ITPUB www.itpub.net

ATAGURU 炼数成金

FSI

SEC'OUG SOUTHEAST CHINA ORACLE USERS GROUP 中国东南地区甲骨文用户组

Broadview®

收获，不止SQL优化

抓住SQL的本质

梁敬彬 梁敬弘 著

◎ 方法意识巧妙融入 ◎ 脑图表格清晰展现

◎ 海量案例完美结合 ◎ 线上线下拓展延伸

资源由 www.eimhe.com 美河学习在线收集提供

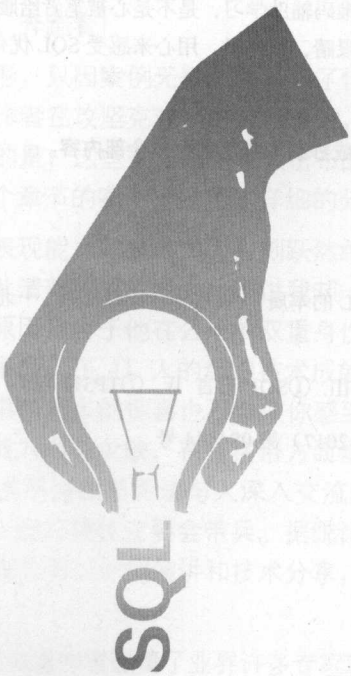
中国工信出版集团

電子工業出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY http://www.phei.com.cn

收获，不止SQL优化

抓住SQL的本质

梁敬彬 梁敬弘 著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

资源由 www.eimhe.com 美河学习在线收集提供

内 容 简 介

有人就有江湖，有江湖就有 IT 系统，有 IT 系统就有数据库，有数据库就有 SQL，SQL 应用可一字概括："广"。加之其简单易学，SQL 实现也可一字概括："乐"。

然而，SQL 虽然实现简单可乐，却极易引发性能问题，那时广大 SQL 使用人员可要"愁"就一个字，心碎无数次了。

缘何有性能问题？原因也一字概括："量"。当系统数据量、并发访问量上去后，不良 SQL 就会拖跨整个系统，我们甚至找不出哪些 SQL 影响了系统。即便找到也不知如何动手优化。此时的心情也可以一字概括："懵"。

现在本书开始带你抛除烦恼，走进优化的可乐世界！

首先教你 SQL 整体优化、快速优化实施、如何读懂执行计划、如何左右执行计划这四大必杀招。整这些干嘛呢？答案是，传授一个先整体后局部的宏观解决思路，走进"道"的世界。

接下来带领大家飞翔在"术"的天空。教你体系结构、逻辑结构、表设计、索引设计、表连接这五大要领。这么多套路，这又是要干嘛？别急，这是教你如何解决问题，准确地说，是如何不改写即完成 SQL 优化。

随后本书指引大家学会等价改写、过程包优化、高级 SQL、分析函数、需求优化这些相关的五大神功。有点头晕，能否少一点套路？淡定，这还是"术"的范畴，依然是教你如何解决问题，只不过这次是如何改写 SQL 完成优化。

最后一个章节没套路了，其中跟随你多年的错误认识是否让你怀疑人生，其中让 SQL 跑得更慢的观点，是否让你三观尽毁？

再多一点真诚吧，本书提供扫二维码辅助学习，是不是心被笔者给暖到了？

读完全书，来，合上书本，闭上眼睛，深呼吸，用心来感受 SQL 优化的世界。

一个字："爽"！

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

收获，不止 SQL 优化：抓住 SQL 的本质 / 梁敬彬，梁敬弘著. —北京：电子工业出版社，2017.6
ISBN 978-7-121-31436-0

Ⅰ. ①收… Ⅱ. ①梁… ②梁… Ⅲ. ①SQL 语言 Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字（2017）第 084434 号

责任编辑：张月萍
特约编辑：梁卫红
印　　刷：北京京科印刷有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：30.5　　字数：760 千字　　彩插：1
版　　次：2017 年 6 月第 1 版
印　　次：2017 年 6 月第 1 次印刷
定　　价：88.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

序

这是自上一本《收获，不止 Oracle》一书后，我第二次为作者写序，我知道这又是一本极不寻常的书。

果然，初翻开此书，就给我带来了惊喜。作者将全书脉络展现得非常清晰，先在前言中通过小故事梳理出 SQL 优化的方法论，接下来将各 SQL 优化的知识点融入到方法论中，形成了全书目录，从而让读者明白为什么要讲解这些知识，学了这些知识对优化有什么帮助。更让人称道的是，这个目录是以一个生动有趣的足迹图展现在读者面前的，不落俗套的同时给人一种视觉上的惊艳感。这是谁的足迹，分明是你自己的足迹！于是，一种强烈的代入感油然而生，来，迈开双腿，学习着，思考着，奔跑着！

足迹所到之处，感动如影随形，只因案例无数。我看到了作者十多年如一日在工作的荆棘之路中勇往直前的精神，看到了作者在攻坚克难后的沉思总结，看到了作者作为感动福富十大人物的一种坚持的精神！更难得的是，这些实战案例背后密布的代码不但没让我迷糊，反倒让我觉得非常亲切，因为本书为每个章节的案例都进行了详细的分类和汇总，让人一目了然。

翻开此书，作者极佳的文字表现能力和技术实力立刻跃然纸上，读者一定会感叹作者怎么具备将晦涩难懂的技术书写得如此清新脱俗的能力！不过我却一点都不感到意外，始终是抱着一种验证的心态来阅读，其中的原因来自于他在公司的双重身份。梁敬彬是福富特级专家，又是公司四星级内训师，前者的荣誉显示了 IT 人的辉煌技术成就，后者的勋章证明了老师的杰出教学能力，两者一完美结合，书中再多的惊喜也不会使你感到意外了。我看到 IT 企业中有很多技术牛人由于在表达沟通交流方面的欠缺，在传帮带方面做得不够好；也看到很多技术人员具备良好的沟通能力却苦于技术不过硬而无法与人深入交流。作者在这方面给我们广大 IT 技术人员树立了一个很好的榜样，会打硬仗还要会带兵。据统计每年接受梁敬彬培训的福富技术人员多达 400 人，加上他每年在公司以外的演讲和技术分享，梁老师可谓桃李满天下，给梁老师点个大大的赞！

随着对此书的进一步了解，我知道作者邀请了业界许多专家对此书进行完善、美化、审核。至此，我又读出了一种精神，叫“团队精神”，此书正是团队协作的结晶！作者把工作中的团队精神带入书籍编写中，值得称道。我在感叹此书的不同凡响之余，更感慨团队的无穷力量！

此书必将成为 IT 书籍的又一个经典传奇，我相信广大读者在翻阅此书时，除了可以学到精妙的 SQL 优化实用技术外，还可以从无数案例中感受到什么叫激情、震撼；从方法论总结上理解什么叫升华、用心；从各种梳理的表格和思维导图中体会什么叫清晰、极致；从书的精妙视觉设计中领悟什么叫求道、协作。我想说的是，从菜鸟到 SQL 大师其实不易，真正的大师不止是技术上精湛，还需要一种精神。这种精神，还请你在阅读本书中感悟吧！

福富软件公司副董事长　杨林

资源由 www.eimhe.com 美河学习在线收集提供

匠心独运　独树一帜

——与梁敬彬先生序

在拿到敬彬新书的稿件时，我的脑海第一时间呈现出来的就是这八个字：匠心独运，独树一帜。

技术书籍的写作也是一个创作过程，平庸者千篇一律，卓越者自出机抒。

写作一本千篇一律的书很容易，而要想自出机抒，形成自己的风格，并且为读者认可，则是难上加难。而敬彬的系列作品，已经形成了自己独特的风格，并且为广大技术爱好者们所喜爱，这不独是匠心所在，更是隐现宗师风范。

如作者所说，有数据库就有 SQL，而 SQL 又因其灵活、复杂，而让众多应用系统饱受性能之苦。我一直认为，在开发环节提高 SQL 质量才是数据库优化的治本良方，SQL 审核也是 DevOps 理念在数据库领域的最佳落地点，云和恩墨也在此保持持续的关注并研发了产品。敬彬的新书从 SQL 入手，以其独特的故事演绎法，让 SQL 优化成为了一种趣味，书中还通过实例打破了以讹传讹的种种法则，让读者获得思想上的自由。

这是一本活的书，活跃的思想，活泼的行文，活动的二维码，活灵活现的音视频，互联网时代，原来书还可以这样写。

快点来一起体验吧！

盖国强

云和恩墨创始人，Oracle ACE 总监，ACOUG 主席

资源由 www.eimhe.com 美河学习在线收集提供

名家力荐

众所周知，数据库应用是 IT 系统极其关键的核心组成部分，而 SQL 是数据库唯一的交互语言，SQL 语句实现难度不大， 但是 SQL 语句优化却比较复杂，需要有人引路，不过这次有了梁老师，广大读者有福了

梁敬彬先生曾参与的大作《剑破冰山——Oracle 开发艺术》一书，直至今日，部分内容在行业里还发挥着重要影响。梁先生的《收获,不止 Oracle》，用生动的故事形式叙述复杂技术，开创数据库技术书籍故事化写作的先河。梁先生技术功底和文字功底同样深厚，更重要的是，具有作为讲师的那种缜密、体系化的思维方式，以及对读者心思的透视力。

此次梁先生的新书更让我吃惊，整本书的 17 个章节结合实战案例，完全被融入到一套完整的方法论中，脉络极其清晰，这是一本有着高度思想性的书，构思思路让人叹为观止。这是一本值得向行业推介的优秀技术书籍！

黄志洪（tigerfish）

炼数成金创始人

SQL 优化并不简单，做好 SQL 优化需要掌握数据库体系结构、表和索引设计、高效 SQL 写法、高级 SQL 语法、多种优化工具等知识，甚至还得分析业务特点，以及了解优化器的缺点。

只有建立 SQL 优化方法论体系，才能够迅速找到最适合的方法来优化 SQL，从而解决由 SQL 引发的性能问题。

在这本书里，梁兄全方位详解了 SQL 性能优化之道，相信读者定会受益良多！

丁俊（dingjun123）

ITPUB Oracle 开发版资深版主

《剑破冰山——Oracle 开发艺术》副主编

继上一本《收获，不止 Oracle》书后，由梁敬彬、梁敬弘兄弟合著的《收获，不止 SQL 优化》再次问世了。感慨两位兄弟在技术之路上孜孜不倦的追求和无私的分享。

梁敬弘是我的学生，学业专精，为人善良热心，是一个非常不错的小伙子。哥哥则精于实战，善于总结，在业内是一个极为知名的数据库专家。两位兄弟联手完成的新书必然是数据库

资源由 www.eimhe.com 美河学习在线收集提供

领域的精品，值得大家去学习和体会。在此，预祝本书的出版获得成功，同时也祝兄弟二人在事业上取得更大的成就。

黄连生

清华大学计算机系教授，博士生导师

据我所知，两兄弟合著的《收获，不止 Oracle》口碑极好，创造了 2 个月内 3 次印刷的销量佳绩，满意率在京东、当当达到了 99%以上，获得了巨大的成功。身边很多清华的学弟学妹们也都购买了此书。我作为作者的老师、挚友、大哥，为他们高兴，得知他们要再次出新书，我更是为他们感到骄傲！

翻阅《收获，不止 SQL 优化》，我发现这确实是一本与众不同的书：清晰的结构、形象的比喻、经典的案例、生动的故事让复杂枯燥的知识瞬间变得简单有趣起来，更难得的还可以扫描二维码导入线上延伸学习，这种责任感让人赞叹不已。我坚信，以敬彬的博学多才和敬弘的扎实严谨，这本新书将会成为数据库书籍的再一个经典传奇！

王道顺

清华大学计算机系教授，博士生导师

《收获，不止 SQL 优化》是市面上我读到的最好的一本 SQL 优化书籍，犹如左右互搏之术，左手原理，右手实战，左右开弓，原理中有实战，实战中有原理，把原理和实战融为一体。本书的精妙之处在于作者的优化思想，一招致胜。

本书适合于 IT 开发者、DBA、应用运维人员、IT 爱好者、计算机专业学生，强烈推荐！

郭一军(guoyjoe)

尖峰在线教育创始人，浙江象行数据技术有限公司 CEO

我对梁敬彬先生的第一感觉是勤奋。作为一双儿女的父亲，在业余时间还能独立完成两本著作，这本身就需要付出巨大的劳动。

我对梁先生的第二感觉是有为。集软件技术专家、培训讲师、围棋业余 5 段于一身，这充分体现了他的才智。

我对梁先生的第三感觉是亲和。我们从他的著作、他发表的文章，以及他的演讲都能体会到，“循循善诱、诲人不倦”这 8 个字。

这本《收获，不止 SQL 优化》，你从章节编排设计就能感受到梁先生的用心，书中的主题也正是数据库开发从业人员在工作学习中必然会遇到的。数据库开发博大精深，这本作者从他

资源由 www.eimhe.com 美河学习在线收集提供

十多年的成功经验总结归纳出的指南，指引我们向正确的方向前进，少走弯路，健康成长。

卢涛

ITPUB Oracle 开发版资深版主

系统分析师

早和梁敬彬先生认识是由于我们长期同在福建省内耕作 Oracle 并且一起经常被叫作“老师”。熟来熟往，因此了解敬彬演绎技术的风格是这样的：从读者的角度出发，在类似小品的故事情节中生活化地展示原先看似复杂的技术。这种风格太好了，尤其是用在深入演绎 SQL 优化这一项他的专长之上。读过书稿之后，我不禁拍案叫绝。像这样去传授 SQL 知识，去展现最佳实践，能让“开卷有益”这四个字实至名归。

长久以来中国东南地区 Oracle 技术交流讨论的气氛都不够浓郁。为了改变本地 Oracle 社区的现状，最近非常有幸我能和他一起作为 SouthEast China Oracle Users Group（SECOUG）的发起人协力去建设我们自己的本地 Oracle 社区。在大量的现场技术培训和技术支持中，我们发现，中国东南地区其实不乏 Oracle 技术热爱者，只是缺乏像用户组这样的分享平台和分享平台上的有益读物。尤其是涉及比较复杂的 SQL 优化项目时，我们的 Oracle 技术热爱者们需要有人去引领和交流。敬彬的这本《收获，不止 SQL 优化》会成为这方面杰出的技术交流媒介，更能帮助 SQL 优化工作者们在个人技术生涯中因为阅读此书而有收获进而变得更为成熟。这本书也会成为 SECOUG 社区分享的重要读物。

唐波

中国科学院 Oracle EBS 最佳技术顾问，福建省知名 Oracle WDP 讲师

中国东南 Oracle 用户组 SECOUG 联合发起人，“DBA+社群”联合发起人

敬彬兄再次出书，依然是脑图逻辑为先，用语通俗易懂，细节深入浅出。我仔细拜读了第 1、2、17 章，敬彬兄不仅将 SQL 优化需要使用的工具做了全面详实的介绍，更结合他在不同行业的实际案例，用诙谐笔法娓娓道来。强烈推荐给还在优化之路上奋斗的 DBA、开发人员们，你定会如书名所言，《收获，不止 SQL 优化》！

杨志洪

“DBA+社群”发起人，新炬网络首席布道师，Oracle ACE，《Oracle 核心技术》译者

与吾兄敬彬相识九载，于剑破冰山始于交心，著述之道甚谨，曾有幸聆听吾兄传道，深入浅出，高屋建瓴，旁征博引，家事国事天下事信手拈来，堂上气氛甚悦，无他“乐”。

乐乃人与生俱来之追求，倘若没有乐，也就丧失了努力的动力。

当然入门之际，首先会“愁”和“懵”。Oracle 发展至今已 40 年，历经若干版本，并得以在大数据、云计算和去 IOE 的大势之下屹立不倒，得益于 Oracle 自身体系架构的严谨和不

资源由 www.eimhe.com 美河学习在线收集提供

断完善，洋洋洒洒数十万页官方文档，即使一辈子也未必能穷就。敬彬之特长就在于化繁就简，由道入术，轻松愉悦中掌握 SQL 优化之技能，一个字“爽”。

弟不才，混迹于各大 IT 论坛，尝闻“术业有专攻”，予则一塌糊涂，得蒙写荐言，不慎惶恐。

王保强

某移动公司首席架构师，IT 畅销书作者

敬彬的新作《收获，不止 SQL 优化》的目标非常聚焦。和某些同类书籍哒哒哒地扫射不同，它是以精准狙击的方式直接锁定数据库领域的难点和痛点，即“SQL 优化”这个话题，宁小不贪大，求透不求全。

难能可贵的是，本书并没有多少高深莫测的理论，内容非常接地气，属于即学即用、一用见效的类型。那是因为，书里所有智慧都是从作者和他的同事们实践中萃取并在实践中得到反复验证的，所有代码都是两位作者一行一行亲自敲出来的，大多数的案例、故事源自真实的工作场景，可以找到事件的原型。

这本书在易学、易用方面，下了很多苦功。但凡有点 SQL 基础的人，看这本书一定不费劲。仿佛有一位优秀的导游，拿着一张详尽的地图，手把手牵着你一路逛过去，你压根不用担心自己迷路。在此预祝读者朋友读书读人，见仁见智，受益多多！

王法松

企友咨询 CEO，知识管理专家，知名课程开发师

和敬彬的第一次相识，是源于 2015 年福建 IT 培训联盟的成立，福富大学校长陈明先生第一个就向我推荐了敬彬。敬彬给我的第一印象是非常谦虚，他一直强调自己并不是什么大师，只是比别人多了一些工作总结，把总结编辑成书籍而已，在我翻看他的第一本数据库专著《收获，不止 Oracle》时，便被他独到的写书风格所吸引，在业界能干会说的工程师难寻，能干会说还能写得一本好书的技术专家更是凤毛麟角，他无疑是后者。

敬彬让我欣赏的另一点是感恩、开放、共享的个性和理念，每当他有机会分享自己的成长经历时，总是用各种方式真诚流露出感恩之情，如今他正以自己的努力和付出回报这个社会。福建 IT 培训联盟成立之初，他开放分享的理念感染了一群怀有技术梦想的年轻人投身到联盟的公益服务。他专精数据库技术，点滴成河，汇聚成海，孜孜不倦，匠心可见，《收获，不止 SQL 优化》一书是福建 IT 培训联盟的优秀代表和骄傲。

黄美龙

福建 IT 培训联盟创始人，福州市软件行业协会副秘书长

资源由 www.eimhe.com 美河学习在线收集提供

作者简介

梁敬彬，福富研究院副理事长、公司唯一四星级内训师。不仅是公司特级专家也是国内一线知名数据库专家，其个人及团队在数据库优化和培训领域有着丰富的经验、过硬的质量和良好的口碑。多次应邀担任国内外数据库大会的演讲嘉宾，在业界有着广泛的影响力。著有多本畅销数据库技术书籍，其代表作《收获，不止 Oracle》已成为数据库领域有口皆碑的经典书籍，《收获，不止 SQL 优化》即将开创一个新的里程碑。

梁敬弘，清华大学计算机系博士毕业，在计算机领域和金融领域皆有建树，拥有多项计算机相关核心专利技术的同时，还拥有金融行业的 CFP 等高级认证。现就职于华夏银行总行。

资源由 www.eimhe.com 美河学习在线收集提供

致谢

我首先要感谢福富软件公司，因为这本书的原型，正是公司的认证资格课程《基于案例学SQL 优化》。公司福富大学专程请来专业的企业内训专家为福富内训师们做内训课程的培训和完善，最终这门课程有幸成为我们公司年度三门精品课程之首。这期间福富研究院的专家们对本课程进行了大量的评审，并提出了各种宝贵的意见。感谢福富公司！感谢福富大学和福富研究院！

我要感谢我们项目组团队的成员，没有黄锏、荣志等公司杰出的技术专家和我并肩作战，我也没有精力写完这本书。我要感谢姚建艺、郑清泉、郑超群等，他们为我们团队的工作提供了最有力的帮助。要特别感谢我们的杨总，她一如既往的支持、她热情洋溢的《序》让我感动不已！

我要感谢我弟梁敬弘在本书内容上倾注精力。感谢林舒楠、张凤为本书在视觉设计上提供的帮助。感谢谢恒忠在线上拓展方面提供的帮助。谢谢你们的参与！

此外，我要感谢苏旭晖、卢涛、丁俊等这几位业内知名的数据库专家对本书的审核校验，感谢博文视点在出版方面的专业性指导意见。感谢大家的帮助！

最后，我要感谢我亲爱的家人，谢谢你们的支持！

梁敬彬

2017 年 4 月

资源由 www.eimhe.com 美河学习在线收集提供

前言与意识：从优化方法到全书脉络

叹 IT 之一入深似海

传说：一入 IT 深似海，从此菜鸟泪成河。

老师，搞 IT 真有传说中的这么惨吗，那我从此要珍爱生命、远离 IT 了。

我们慢慢聊吧。话说这时代啊，应该是最好的时代了。知识的获取相当便利，基本上没有什么知识点是搜索引擎搜不到的；此外，现在的技术书籍、教学视频也非常丰富。除了自学手段外，我们甚至还可以在论坛上提问，或者参加各种线上和线下的培训。当今时代，IT 学习成本越来越低，门槛似乎一点都不高！

对啊，那咋说深似海泪成河呢？

其实这话也是有道理的。我们来说说这个时代的 IT 系统，其和从前也大不相同了，现在对外的 IT 系统大多需要同时支持电脑终端和手机终端（手机终端进一步分为 Android 和 iOS 等操作系统），此外还要考虑各个接口，如关联业务接口、短信接口、微信接口、公安接口、银行接口……系统显然比以前更复杂了。这意味着系统开发在功能实现方面的难度更大了，而系统实现难度大又意味着对 IT 开发人员要求更高了！

嗯，好像是这样。

其实不止是 IT 系统功能实现的难度变大。你想想看，现在几乎人人都有手机，手机端的接入就意味着成千上万的人可以随时随地拿起手机访问系统，这给系统带来了可怕的访问量。此外，不可避免地会出现同一时刻大量用户同时访问某应用的景象，这又带来了巨大的并发量。因此系统如果没有良好的性能规划，很容易垮掉。所以说 IT 开发人员的压力不仅是实现难，还会遭遇性能瓶颈。当然，IT 运维人员的压力更大，因为假如系统有问题，他们首当其冲。

哇，好像还真是如此，听得我手心出汗了。

前面我们谈到了功能实现困难，又提到性能瓶颈压力，现在我再提一点，即定位困难。还记得之前我说的接口吗？随着时代的发展，各种 IT 应用已从孤岛走向关联。比如你的系统是计费系统，当要对用户进行计费时，你可能要从客服系统中获取用户的套餐等资料，或许还要去网厅系统完成……这下问题来了，假如应用有故障，你知道问题出在哪吗？是你自己的系统出问题，还是接口的系统出问题？再比如，你好不容易定位出是自己系统的问题，那请问，到底是数据库、前端应用还是中间件的问题呢？

老师，有没有手帕，我擦擦脸上的汗。

资源由 www.eimhe.com 美河学习在线收集提供

假如你已经知道系统的问题出在数据库。那请问，是 SQL 还是其他问题，你如何定位，如何判断？再假如你通过努力判断出是 SQL 问题，那该如何优化，是动手改写呢，还是不用改写，加加索引啥的……

老师，要考虑的方方面面太多了，看来我是把 IT 系统想简单了。

刚讨论的话题，放在以前，是基本不用担忧的，这是时代高速发展带来的问题。接下来你换一个角度想想，这个时代越来越多的人依赖 IT 系统，你的系统一旦出现问题，多少用户会受到影响？这个时代越来越多的 IT 系统之间有关联，你的系统一旦出现问题，多少别人的系统会受到影响？怎么样，是不是又感受到另一层面的压力了。

哇，看来这时代 IT 人尤其是 IT 菜鸟的日子真的不好过啊！一入 IT 深似海，从此菜鸟泪成河。

结论：

当今时代的 IT 系统复杂度高、数据量和并发量大、关联性强，无论是定位解决故障还是应用开发维护，难度都比较大，并不是一件轻松的事情。

赞 IT 之 SQL 地位高

你也吟上这诗了，别伤感，这不也有好事嘛，我之前就说过如今学习比以前容易很多。

说的也是，我都忘记了，还好还有这点值得安慰。

真是如此吗？好吧，以 IT 学习中的 SQL 优化为例，你知道如何进行学习吗？

这个我知道，SQL 优化主要是看执行计划，比如发现是不合理的全表扫描，就设法转成索引扫描等。

说得有点简单啊。其实 SQL 本不需要优化，就是因为前面我们所说的当今 IT 系统复杂度高、访问量和并发量都大，而数据又是 IT 应用中访问的热点，因此这些压力自然就体现在 IT 系统对应的数据库模块上，所以 SQL 需要优化。

哦，原来如此，明白了。

任何 IT 系统，数据都是核心，同时也是访问和展现的热点，脱离数据库的 IT 项目几乎不存在，甚至可以说几乎没有不需要进行数据库操作的编程人员，而能与数据库进行无缝交互的就只有 SQL 了。此外，SQL 是一种学起来非常容易的“傻瓜语言”，随便一个 where 条件就是一个需求实现，基本上新手级别的开发人员坐下来看看简单语法即可编写 SQL，如果有 3 天时间边做边学，基本上所有 SQL 都会编写了。用我本人的例子来说吧，有人忽然问我学 SQL 开发学了多久，我几乎是本能般从嘴里冒出一句：SQL 开发，我有花时间学吗，写 SQL 难道不是自然而然就会了吗？

正因为 SQL 如此重要，学习成本又如此之低，同时与 IT 系统中不可或缺的数据库交互起来浑然天成，所以几乎所有 Java、C 等开发人员都能较熟练应用数据库 SQL 开发技术。这导致应用 SQL 开发的人在数量上异常庞大，简单地说，就是所有前后端程序开发人员和 IT 运维人

资源由 www.eimhe.com 美河学习在线收集提供

员以及数据库开发人员的总和！

于是在高访问量、高并发的 IT 系统数据库模块中，平均每秒运行成千上万条 SQL 的场景已成常态。在这种情况下，这些 SQL 如果运行较慢，便容易迅速拖垮整个 IT 系统，因此 SQL 优化就变得特别重要了。此外，由于 SQL 过多，不可能仅靠一两个 SQL 专家筋疲力尽地调优，我们便可以高枕无忧了。最有效的方式是，每个 SQL 编写人员自己要有 SQL 优化的意识和本领！

老师，我在项目组里做 Java 开发，确实也涉及大量的 SQL 开发，您说得没错，我也感觉 SQL 开发特简单，不过我的 SQL 经常跑得很慢，您说 SQL 优化好学吗？

结论：

数据是 IT 系统的核心、重中之重，而 SQL 是数据交互的必然手段，所以 SQL 的应用非常广泛，使用人群数量也非常庞大，因此 SQL 的重要性不言而喻！

涌 SQL 之优化泪水

SQL 优化肯定比 SQL 编写本身要难很多，但也存在一些优化的基础知识，如 SQL 执行计划、索引原理，等等。这些都比编写 SQL 本身要复杂得多，因此要成为 SQL 优化高手仅知道一些优化基础知识是远远不够的，还需要经验的沉淀，并且要转化成你的方法论。

老师，您能否举例说明一下需要什么经验吗？

不妨我回答得更有趣点吧，先给你说几个 SQL 优化的小故事，其中奥妙我们后续探讨。

太好了！

嗯，你边听边思考。故事 1：话说某天上午小王被告知某系统的一个菜单访问非常慢，于是他开始介入优化，他跟踪到该菜单调用的具体 SQL，接下来他通过观察该 SQL 的执行计划发现该 SQL 访问某张表时没走索引，觉得该表应该加一个索引，他花了整整 1 个小时发现这个问题后非常兴奋，于是马上动手开始建索引了。建索引大概花费了几分钟时间，随后他发现 SQL 走索引了，并且真的快了许多。于是测试一下该菜单，果然快了不少。故事 1 讲完了。

啥，讲完了，太短了吧，您这说的是优化成功案例吗？

是否成功一会儿再下结论，接下来说故事 2。话说小王优化效果杠杠的，正自鸣得意时，被告知虽然这个菜单变快了，不过刚才持续几分钟时间出现访问该菜单一直报错的情况，随后正常。

老师，这啥意思，为啥应用程序会出错，怎么就恢复了？您故事 2 讲完了吗，怎么您的故事都这么短啊？

嗯，现在开始说第 3 个故事了。当天下午小王又接到电话，被告知那菜单访问又慢了。小王有点吃惊，于是赶紧登录系统运行该 SQL，发现确实变慢了，不过奇怪的是该 SQL 正常走索

资源由 www.eimhe.com 美河学习在线收集提供

引，没什么问题啊，小王很是郁闷。正在他束手无策之时，小王又接到电话，说该菜单又快了。晕，他现在可是啥都没做，这是咋回事？当晚，小王辗转反侧，无心睡眠。第 2 天小王又接到电话……他崩溃了。

好可怜啊。然后呢？

继续第 4 个故事。话说小王崩溃之后无法正常工作，于是领导把这难题交给其他人处理了，小王得知后满血复活再度投入工作中。几天后小王又迎来一个新任务，开发项目组中有一条 SQL 很慢，希望能优化一下。小王看了一眼觉得写得歪瓜裂枣样很不舒服，于是挽起袖子对 SQL 进行重写，改改改！

然后呢？

小王改好后，满怀期待的开发组对新的 SQL 一运行，发现跑得比之前的 SQL 还要慢很多！

可怜，小王又崩溃了吧，接下来呢？

嗯，说故事 5 吧。在小王再度崩溃之前，公司的 SQL 优化大师老丁正好路过，他分析了之后，建议将 SQL 语句涉及的某表的外键加上索引。后来大家照办了，果然性能迅速提升！老丁告诉小王，优化前可通过各种手段先观察观察 SQL 涉及的表结构、索引等，看它们有无不合理之处，急于动手改造 SQL 太盲目了，是没有抓住主要矛盾的体现，而且改代码需要测试、打补丁、上线，也是一件很辛苦的事。小王频频点头，谨记老丁教诲。

看来小王成长了。

是吗？那我们继续说故事 6。

一周后小王又面对一条 SQL 需要优化，这次他不动手改写了，尝试了加索引，又尝试了调整表结构，结果提升效果非常不明显……

然后呢，又崩溃了？

好在老丁又及时出现了，他这次没有修改表结构，而是和开发人员进行了半小时的交谈，然后居然对 SQL 进行重写，改改改！然后 SQL 就变得飞快了。小王傻眼了，不是说尽量不着急先动手改 SQL 吗？怎么老丁一看到这 SQL 就改改改。悲催啊，该怎么做才是对的！让小王悲催的还不止这个，老丁的新 SQL 看上去并不是很复杂，可是小王居然看不懂老丁为什么能这么改写。

没错，看不懂！他看不懂老丁写什么，也完全不理解新旧 SQL 的等价性。

可怜的人！老师，我觉得不是“一入 IT 深似海，从此菜鸟泪成河。”而是“一入 SQL 深似海，从此优化泪成河。”

结论：

SQL 优化不是一件容易的事，明明优化后变快了，结果不一会儿又慢了。有时不改写可以解决问题，有时又必须要改写才能解决问题。让人难以适应。

资源由 www.eimhe.com 美河学习在线收集提供

析 SQL 之悲催故事

你故事听完了，啥感想，就是觉得小王好惨，优化很难吗？

嗯，差不多吧。

好吧，让我来解读一下这些故事吧。故事 1 里小王能定位到具体 SQL，并且能根据 SQL 执行计划进行 SQL 优化，这至少说明了小王还是掌握了一定的 SQL 优化基础技能的，否则估计执行计划是什么都没有听过。不过接下来的故事 2 和故事 3 说明他是失败的。为啥呢？那是因为他经验太少，同时也缺乏由经验转换而成的做事方法论。

我是怎么下的这个结论呢？你注意到没，小王他接到电话后就开始动手优化了。他难道不应该多问一句这问题是一直以来就存在呢，还是今天忽然出现的。

老师，问这有啥用呢？

如果是第一次出现，你可能就会关注一下是否昨晚系统做了啥动作，比如昨晚打了一个补丁，这补丁引发这次故障的可能性就非常大，于是目标就很清晰了。在实在无法解决的情况下，回退补丁也是一个思路。而如果是经常有这故障，那应该有其他同事处理应对过，获取他们之前的分析成果，或者收集之前的日志和现在进行比对，这些难道对小王没帮助吗？

老师，事实上是不是因为晚上打补丁导致的？

你可以认为是，也可以认为不是。我说明一下，这几个故事没有真相，只是解读一下。真相不重要，你猜测的过程就是你进步的过程。另外这里小王是加了一个索引，你觉得这个动作是对的还是错的？

老师，我觉得应该是错的，因为在第 3 个故事里，他加索引后，系统虽然前期变快，但是后来又慢了。我觉得这个加的索引没有用，或者至少是没有解决全部问题。不过我就是奇怪为啥时快时慢？

这没啥奇怪的，我们还是进行假设。你不觉得可能有这么一种情况，此时整个系统都非常地慢，根本不只是这个菜单慢。求助者没有提出其他模块慢或许只是因为他平时仅用这个菜单，那你想想，如果整个平台都瘫痪了，局部能快起来吗？

哦，这我怎么一点都没想到啊！

嗯，我只是说可能性。那为什么整个数据库都慢？可能性更多了，比如数据库主机上某些应用程序耗尽了主机的 CPU、内存等资源，数据库运行在这台主机上，覆巢之下安有完卵？接下来解释为啥忽快忽慢问题，正常啊，如果这些害人的应用程序有时运行，有时结束，结束时数据库不就正常了？

原来如此啊，第 3 个故事里的小王崩溃得好冤啊，如果知道事实的真相，他把数据库主机的程序停了或者优化了或者移到别的地方去，就解决问题了。

你的这个解决方案非常棒，不过别忘了我这里不说真相，一切都是可能性解读。当然这些

资源由 www.eimhe.com 美河学习在线收集提供

可能性猜测是基于经验和推理的，也不是完全没依据的。另外，故事 2 里描述菜单曾经出错了几分钟，你注意到并思考过为什么了吗？

有的，不过这是为啥呢？

你没注意到建索引也是几分钟时间吗？几乎可以肯定地说，是这个建索引动作导致的故障。因为建索引会锁全表，这时候更新数据肯定会失败。小王犯了一个大错，在业务高峰期做 DDL 操作，严重地影响了生产，问题解决者成了麻烦制造者。这故事 2 倒是不需要推测的，真相一定如此。

哇，这么严重的错误原来是小王一手造成的，我还真没想到。

嗯，再说故事 4 吧，小王动手改改改，然后效果差差差，这里我们可以解读到什么呢？那就是解决问题要有目的性，不能没找到真正原因盲目动手。你看他觉得 SQL 写得不好，就改。实际上应该观察慢在什么地方，比如可以通过执行计划看出最大的开销在某全表扫描上，而该全表扫描完全可以通过索引减少访问路径，这时加索引就可以解决问题，改写 SQL 不可能解决问题的。

哦，确实如此。

接下来，在故事 5 里，老丁果然只是增加了索引后就解决了性能问题。优化 SQL 不一定非要改写才可以优化，有时根据数据库的体系逻辑结构不改写 SQL 也可完成优化。改写 SQL 的代价也更高，因为现实中如果你要改 SQL 肯定需要经过测试、打补丁、上线等多个过程，不可能直接就在生产中修改。

最后看故事 6，小王学习老丁只调整参数进行 SQL 优化。接下来剧情反转了，小王学老丁不改，效果差差差。而老丁则动手改改改，效果又是杠杠的。小王欲哭无泪，不知自己该如何做了。其实这里小王生搬硬套了，他没有找到本质原因，比如此时可能是由于冗长的写法导致表访问了多次，而老王改写 SQL 将表访问次数大幅度降低下来。这时不改写是无法优化的。又或者是老丁根据业务需求，砍掉了某些多余的逻辑，这就更需要改写 SQL 了。

哦，老师您总结得很到位啊！

由于不改写通常来说比改写高效，而不改写的优化一般都和数据库的体系逻辑架构有关。因此我们需要认真学习这部分基础知识，这也就是老师的 SQL 优化课程中为什么会涉及这部分知识的原因。不过能不改写优化固然好，有时等价改写也是必需的，而且改写其实分成两个部分：一个是等价改写；一个是根据业务改写。比如小王看某 SQL 写得很不顺眼，然后动手改改改，这显然是等价改写。而老丁和开发人员交谈了半个多小时，改造后的 SQL 连小王都看不明白，这就很可能是根据业务改写的。

明白了，原来如此。

业务改写是优化的最高境界，老丁通过和开发人员交流后发掘出真正的需求，然后写出来的代码表面上看和旧代码逻辑完全不等价，实际却等价。

资源由 www.eimhe.com 美河学习在线收集提供

哦，老师您能否解释一下什么叫真正需求？

好吧，还是讲生动点的例子吧。从前我曾经讲过一个《小余买鱼》的故事。小余妈妈一个劲地让小余买鱼招待客人，条件不允许的时候还坚持如此。后来小余让妈妈用冰箱里的牛肉来代替鱼，妈妈也猛然醒悟了，她忘记了需求的本质是做美味给大家分享。站在做美味这角度上看，去老远的地方买鱼和拿冰箱里的牛肉，又有什么区别呢？

我明白了，没有提炼到这层需求的人，真的很难理解吃牛肉和吃水煮鱼其实是一样的，代码改写前后表面上有这么大差距，也难怪小王会大惑不解。

是的，这里顺便再说一点，小王看不懂老丁写的 SQL，也可能是因为老丁用了某些可能小王没见过的 SQL 语法来改写，我们这里称之为高级 SQL。比如 WITH 子句、树形递归、分析函数，等等。

哦，好想见识一下。

结论：

其实这一节想说的就是，做事要有方法论，要先整体后局部，解决问题要注重效率，先尽量考虑不改写的优化，再考虑改写的优化。而不改写的优化靠的是体系结构知识的沉淀，而改写则要考虑逻辑等价改写和业务改写两大思路，其中业务改写是 SQL 优化的最高境界。另外还是要有一定的知识沉淀，高级 SQL 的语法也要掌握，其在很多场合下能帮上我们大忙。

推规范流程之必要

其实小王的系列故事还暴露了他身上一个非常重要的缺陷，那就是做事缺乏流程、缺乏规范。先不说解决问题的方法论，小王在系统中建了索引后导致系统出现问题，就是犯了一个不规范操作的低级错误。这应该在 DBA 的工作流程中是明确禁止操作的。当然小王有权限做这件事也是值得深究的，这种权限是不是要管控得更严格一些？这里涉及了作业流程规范。

接下来小王手忙脚乱地进行优化，是不是都很有序？他要花一小时时间才定位到某处需要建立索引，如果建索引有依据的规范并且能快速被体检报告诊断出来，他需要花一小时吗？这里涉及了数据库的设计规范。

故事中还涉及 SQL 改写优化的过程，如果性能低下的 SQL 在运行之前能被事先捞取出来，则很多故障就能避免。这里涉及 SQL 写法的规范。

规范不仅重要，我们更要主动去履行。如果我们能一键发现权限不当、数据库设计不良、SQL 写法糟糕的问题，那规范就容易遵守，而不是停留在嘴上。不过放心，我们的数据库整体诊断工具，涵盖了这里大部分的规范检查。

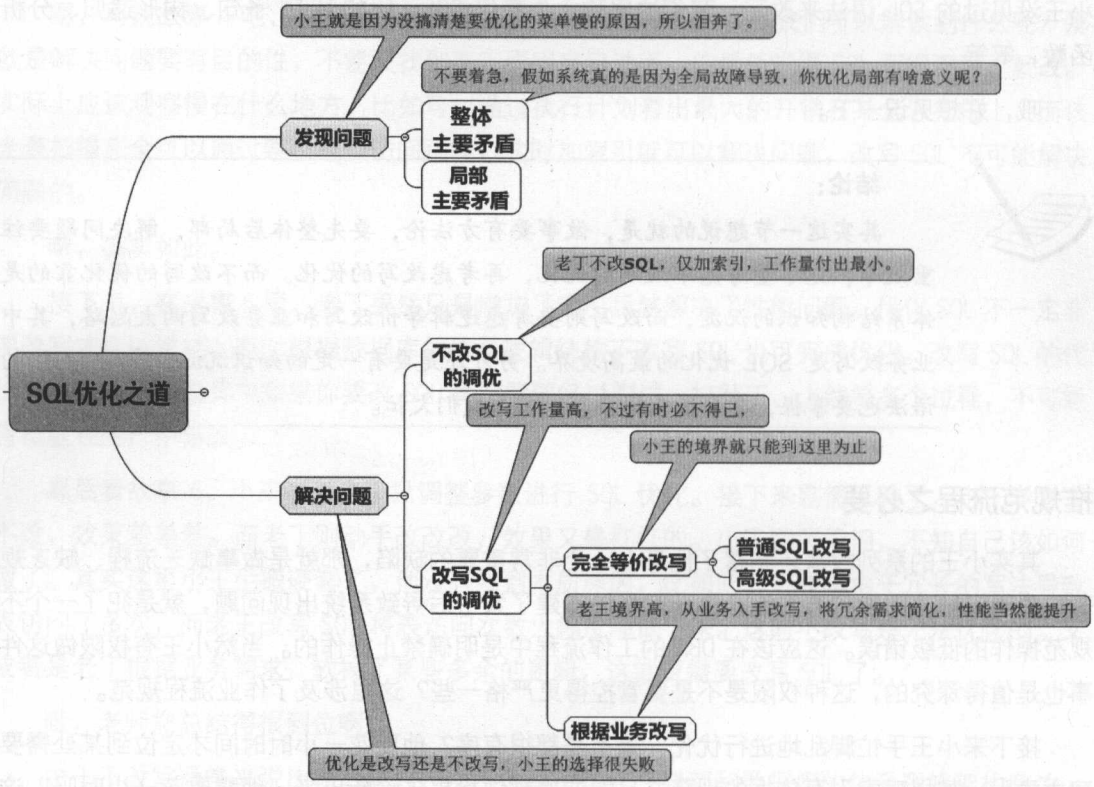
资源由 www.eimhe.com 美河学习在线收集提供

结论：

不成规矩，无以成方圆，如果能制定一定的规范并进行有效的检查，系统的性能问题必然会大幅减少。这里有一个好消息，就是这些要点大多都涵盖进笔者的一键获取数据库整体信息的脚本中了。

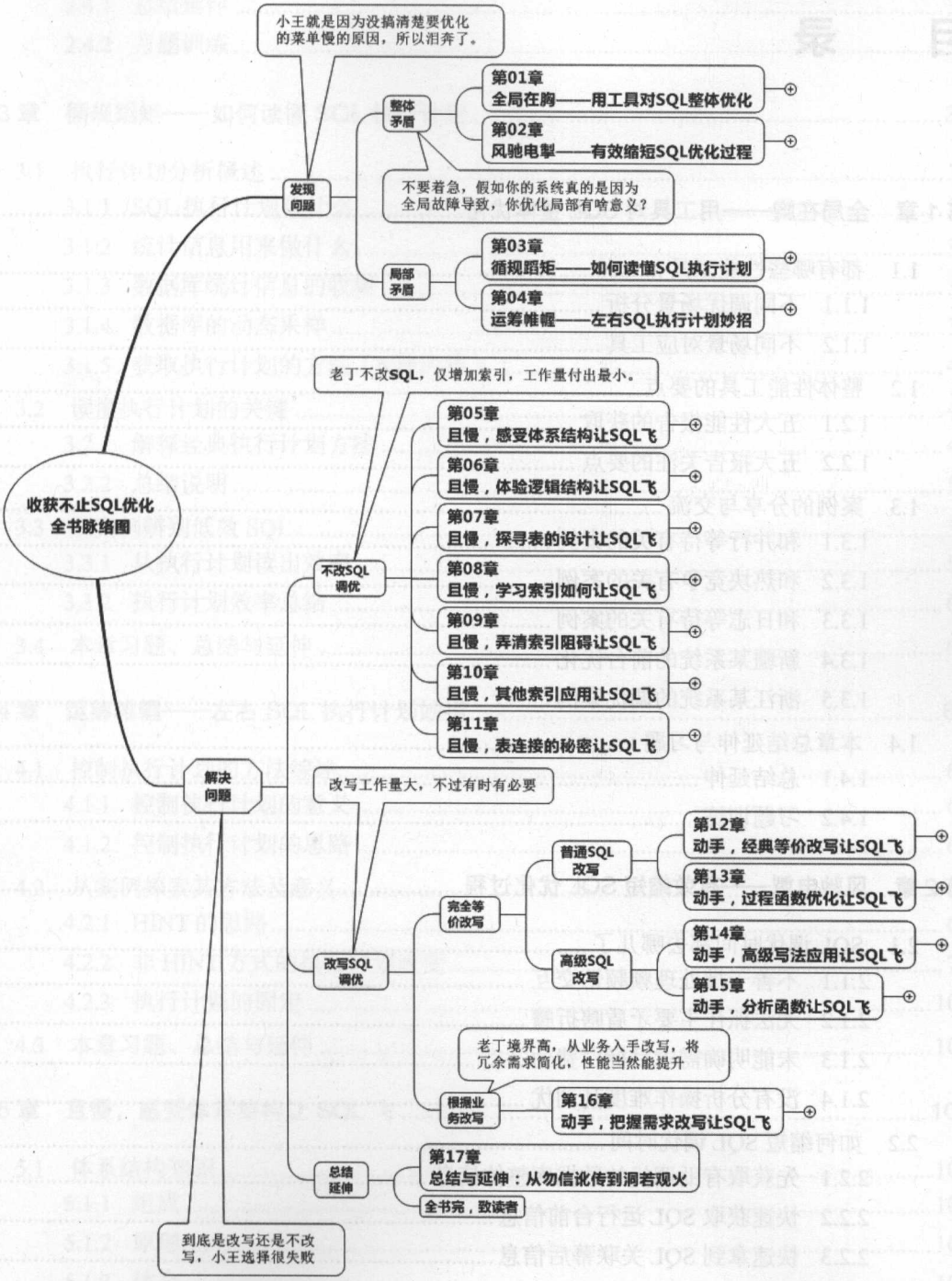
展全书学习之路线

简而言之，小王的悲催故事千言万语化作如下一张学习路线图。限于篇幅，关于流程规划就不在书中详细说明了，我们会将其融进发现问题的整体优化模块中，并写成代码的形式。



接下来，我们就要正式进入全书的学习了，这里同学们先扫一扫二维码，通过一段梁老师精彩的视频来了解 SQL 优化学习的全书之旅，相信本书让大家收获的不止是 SQL 优化。学习路线图最终如下：

资源由 www.eimhe.com 美河学习在线收集提供



资源由 www.eimhe.com 美河学习在线收集提供

目　录

第 1 章　全局在胸——用工具对 SQL 整体优化 1

1.1　都有哪些性能工具 1
1.1.1　不同调优场景分析 2
1.1.2　不同场景对应工具 2
1.2　整体性能工具的要点 4
1.2.1　五大性能报告的获取 5
1.2.2　五大报告关注的要点 10
1.3　案例的分享与交流 18
1.3.1　和并行等待有关的案例 18
1.3.2　和热块竞争有关的案例 19
1.3.3　和日志等待有关的案例 20
1.3.4　新疆某系统的前台优化 20
1.3.5　浙江某系统的调优案例 21
1.4　本章总结延伸与习题 21
1.4.1　总结延伸 21
1.4.2　习题训练 23

第 2 章　风驰电掣——有效缩短 SQL 优化过程 24

2.1　SQL 调优时间都去哪儿了 25
2.1.1　不善于批处理频频忙交互 25
2.1.2　无法抓住主要矛盾瞎折腾 25
2.1.3　未能明确需求目标白费劲 26
2.1.4　没有分析操作难度乱调优 26
2.2　如何缩短 SQL 调优时间 27
2.2.1　先获取有助调优的数据库整体信息 27
2.2.2　快速获取 SQL 运行台前信息 27
2.2.3　快速拿到 SQL 关联幕后信息 28
2.3　从案例看快速 SQL 调优 29
2.3.1　获取数据库整体的运行情况 29
2.3.2　获取 SQL 的各种详细信息 29

资源由 www.eimhe.com 美河学习在线收集提供

2.4 本章总结延伸与习题 32
2.4.1 总结延伸 32
2.4.2 习题训练 33

第 3 章 循规蹈矩——如何读懂 SQL 执行计划 34

3.1 执行计划分析概述 35
3.1.1 SQL 执行计划是什么 35
3.1.2 统计信息用来做什么 36
3.1.3 数据库统计信息的收集 37
3.1.4 数据库的动态采样 37
3.1.5 获取执行计划的方法（6 种武器） 40
3.2 读懂执行计划的关键 48
3.2.1 解释经典执行计划方法 49
3.2.2 总结说明 55
3.3 从案例辨别低效 SQL 55
3.3.1 从执行计划读出效率 56
3.3.2 执行计划效率总结 60
3.4 本章习题、总结与延伸 60

第 4 章 运筹帷幄——左右 SQL 执行计划妙招 62

4.1 控制执行计划的方法综述 63
4.1.1 控制执行计划的意义 63
4.1.2 控制执行计划的思路 64
4.2 从案例探索其方法及意义 65
4.2.1 HINT 的思路 65
4.2.2 非 HINT 方式的执行计划改变 72
4.2.3 执行计划的固定 100
4.3 本章习题、总结与延伸 102

第 5 章 且慢，感受体系结构让 SQL 飞 103

5.1 体系结构知识 104
5.1.1 组成 104
5.1.2 原理 104
5.1.3 体会 105
5.2 体系与 SQL 优化 106
5.2.1 与共享池相关 107

资源由 www.eimhe.com 美河学习在线收集提供

5.2.2 数据缓冲相关 111
5.2.3 日志归档相关 116
5.3 扩展优化案例 118
5.3.1 与共享池相关 118
5.3.2 数据缓冲相关 122
5.3.3 日志归档相关 126
5.4 本章习题、总结与延伸 130

第 6 章 且慢，体验逻辑结构让 SQL 飞 132

6.1 逻辑结构 132
6.2 体系细节与 SQL 优化 133
6.2.1 BLOCK 133
6.2.2 SEGMENT 与 EXTENT 137
6.2.3 TABLESPACE 139
6.2.4 ROWID 139
6.3 相关优化案例分析 140
6.3.1 块的相关案例 141
6.3.2 段的相关案例 144
6.3.3 表空间的案例 148
6.3.4 ROWID 151
6.4 本章习题、总结与延伸 153

第 7 章 且慢，探寻表的设计让 SQL 飞 154

7.1 表设计 154
7.1.1 表的设计 155
7.1.2 其他补充 155
7.2 表设计与 SQL 优化 156
7.2.1 表的设计 156
7.2.2 其他补充 179
7.3 相关优化案例分析 184
7.3.1 分区表相关案例 185
7.3.2 全局临时表案例 190
7.3.3 监控异常的表设计 195
7.3.4 表设计优化相关案例总结 199
7.4 本章习题、总结与延伸 199

资源由 www.eimhe.com 美河学习在线收集提供

第 8 章 且慢，学习索引如何让 SQL 飞 200

8.1 索引知识要点概述 201
8.1.1 索引结构的推理 201
8.1.2 索引特性的提炼 204
8.2 索引的 SQL 优化 206
8.2.1 经典三大特性 207
8.2.2 组合索引选用 217
8.2.3 索引扫描类型的分类与构造 219
8.3 索引相关优化案例 225
8.3.1 三大特性的相关案例 225
8.3.2 组合索引的经典案例 231
8.4 本章习题、总结与延伸 234

第 9 章 且慢，弄清索引之阻碍让 SQL 飞 235

9.1 索引的不足之处 235
9.1.1 索引的各种开销 236
9.1.2 索引使用失效 236
9.2 感受美好索引另一面 237
9.2.1 索引各种开销 237
9.2.2 索引使用失效 243
9.2.3 索引取舍控制 246
9.3 从案例看索引各种恨 248
9.3.1 索引的开销 248
9.3.2 索引去哪儿了 253
9.3.3 索引的取舍 267
9.4 本章习题、总结与延伸 269

第 10 章 且慢，其他索引应用让 SQL 飞 270

10.1 其他索引的总体概述 270
10.1.1 位图索引 271
10.1.2 函数索引 271
10.1.3 反向键索引 272
10.1.4 全文索引 272
10.2 走进其他索引的世界 272
10.2.1 位图索引 273

资源由 www.eimhe.com 美河学习在线收集提供

10.2.2 函数索引 278
10.2.3 反向键索引 282
10.2.4 全文索引 282
10.3 其他索引的相关案例 285
10.3.1 位图索引 286
10.3.2 函数索引 288
10.3.3 反向键索引 297
10.3.4 全文索引 299
10.4 本章习题、总结与延伸 300

第 11 章 且慢，表连接的秘密让 SQL 飞 302

11.1 三大经典表连接概要说明 302
11.2 各类型表连接的知识要点 303
11.2.1 从表的访问次数探索 304
11.2.2 表驱动顺序与性能 308
11.2.3 表连接是否有排序 311
11.2.4 各连接的使用限制 314
11.2.5 三大表连接的特性总结 317
11.3 从案例学表连接优化要点 （三刀三斧四式走天下） 317
11.3.1 一次 NESTED LOOPS JOIN 的优化全过程 318
11.3.2 一次 HASH JOIN 的 优化全过程 320
11.3.3 一次 MERGE SORT JOIN 的优化全过程 324
11.3.4 一次统计信息收集不准确引发的 NL 性能瓶颈 329
11.4 本章习题、总结与延伸 332

第 12 章 动手，经典等价改写让 SQL 飞 333

12.1 设法减少访问路径 333
12.1.1 CASE WHEN 改造 334
12.1.2 ROWNUM 分页改写 337
12.1.3 HINT 直接路径改造 338
12.1.4 只取你所需的列 339
12.1.5 避免或者减少递归调用 341
12.1.6 ROWID 优化应用 347
12.2 设法避免外因影响 350
12.2.1 HINT 改写确保执行计划正确 350
12.2.2 避免子查询的错误执行计划 350

资源由 www.eimhe.com 美河学习在线收集提供

12.2.3 所在环境的资源不足等问题 351
12.3 本章习题、总结与延伸 351

第 13 章 动手，过程函数优化让 SQL 飞 352

13.1 PL/SQL 优化重点 353
13.1.1 定义类型的优化 353
13.1.2 PL/SQL 的集合优化 355
13.1.3 PL/SQL 的游标合并 361
13.1.4 动态 SQL 364
13.1.5 使用 10046 TRACE 跟踪 PL/SQL 368
13.2 PL/SQL 优化其他相关扩展 369
13.2.1 编译无法成功 369
13.2.2 通用脚本分享 370
13.3 本章习题、总结与延伸 380

第 14 章 动手，高级写法应用让 SQL 飞 381

14.1 具体 SQL 调优思路 381
14.1.1 改写 SQL 调优 382
14.1.2 不改写 SQL 调优 382
14.2 高级 SQL 介绍与案例 383
14.2.1 GOURP BY 的扩展 383
14.2.2 INSERT ALL 389
14.2.3 MERGE 392
14.2.4 WITH 子句 402
14.3 本章习题、总结与延伸 404

第 15 章 动手，分析函数让 SQL 飞 406

15.1 高级 SQL 之分析函数 407
15.1.1 语法概述 407
15.1.2 特别之处 407
15.2 分析函数详解与案例 409
15.2.1 学习详解 410
15.2.2 案例分享 417
15.3 本章习题、总结与延伸 432

资源由 www.eimhe.com 美河学习在线收集提供

第 16 章　动手，把握需求改写让 SQL 飞 433

16.1　考虑需求最小化 434
16.2　千万弄清 SQL 改造的等价性 434
16.2.1　看似等价的写法，其实不等价 435
16.2.2　看似不等价的写法，其实等价 438
16.3　开发设计应用中的需求 439
16.3.1　界面权限设计优化 439
16.3.2　界面汇总与展现 439
16.3.3　界面实时刷新改良 439
16.3.4　目录树菜单的优化 440
16.4　场景选择的经典案例之谁是 COUNT(*)之王 440
16.4.1　优化过程 440
16.4.2　优化总结 445
16.5　本章习题、总结与延伸 446

第 17 章　总结与延伸：从勿信讹传到洞若观火 447

17.1　SQL 优化的各个误区 447
17.1.1　COUNT(*)与 COUNT(列)的传言 447
17.1.2　谈 SQL 编写顺序之流言蜚语 451
17.1.3　IN 与 EXISTS 之争 455
17.1.4　总结探讨 457
17.2　误区背后的话题扩展 457
17.2.1　话题扩展之等价与否优先 457
17.2.2　话题扩展之颠覆误区观点 458
17.3　全书完，致读者 461

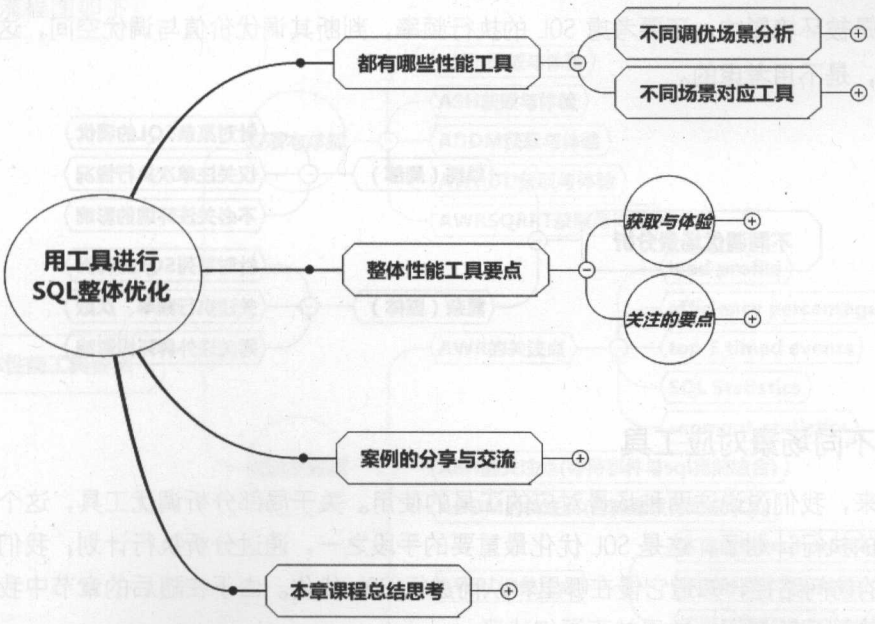
资源由 www.eimhe.com 美河学习在线收集提供

第1章　全局在胸——用工具对 SQL 整体优化

只有站得高，你才看得远

01 全局在胸——
用工具对 SQL 整体优化

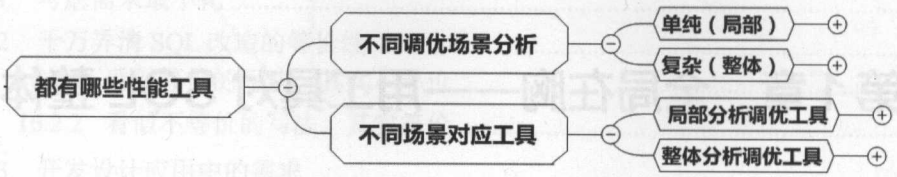
从前言故事中，大家可以明白一个道理：SQL 优化是一个复杂的工程，首先要讲究从整体到局部。嗯，那我们就从整体开始吧。首先，我们学习关于数据库整体优化都有哪些性能工具；接着分析这些工具的特点，并结合案例进行探索；最后再进行本章课程的总结和思考。总体学习思路如下图所示：



1.1　都有哪些性能工具

都有哪些性能工具呢？这里首先要分成两部分：一种是不同调优场景的分析，可分为单纯

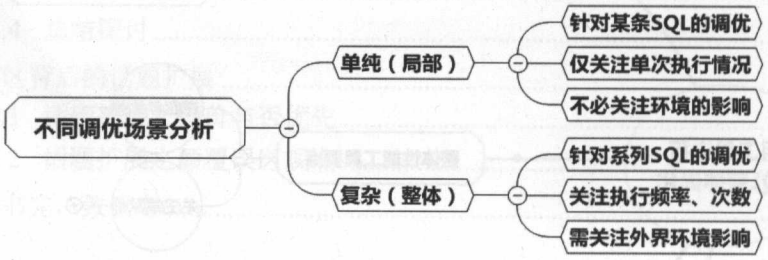
场景的优化和复杂场景的优化；而另一种是基于这些场景的工具应用，就是针对单纯场景的优化手段和复杂场景的优化手段。



1.1.1 不同调优场景分析

我们继续探讨，单纯是有多单纯呢？哦，其实可以理解为无菌真空实验室里的实验。比如一条 SQL 很慢，原因是未走高效的索引查询而走全表扫描，加个索引就快了，执行速度从 10s 变成了 0.1s；或者一条 SQL 执行速度被优化到 1s 左右，逻辑读控制在 50 个左右，应该就已经 OK。这就是单纯的环境，我们差不多无须再考虑优化了。

那啥是复杂呢？那就是，刚才那个语句加了索引后，本应该从 10s 变成 0.1s，结果还是 10s，甚至变成 30s 了，这是咋回事呢？原来，现在系统是整体出问题了，数据库主机资源耗尽，啥语句都跑不快的。还有那个逻辑读在 50 左右的 SQL，如果一天执行几百几千万次，这要是能将逻辑读降低一点，得省多少的逻辑读啊。原来复杂环境真的很复杂，要考虑 SQL 本身没问题而是被环境影响，还要考虑 SQL 的执行频率，判断其调优价值与调优空间，这些在单纯的环境里，是不用考虑的。

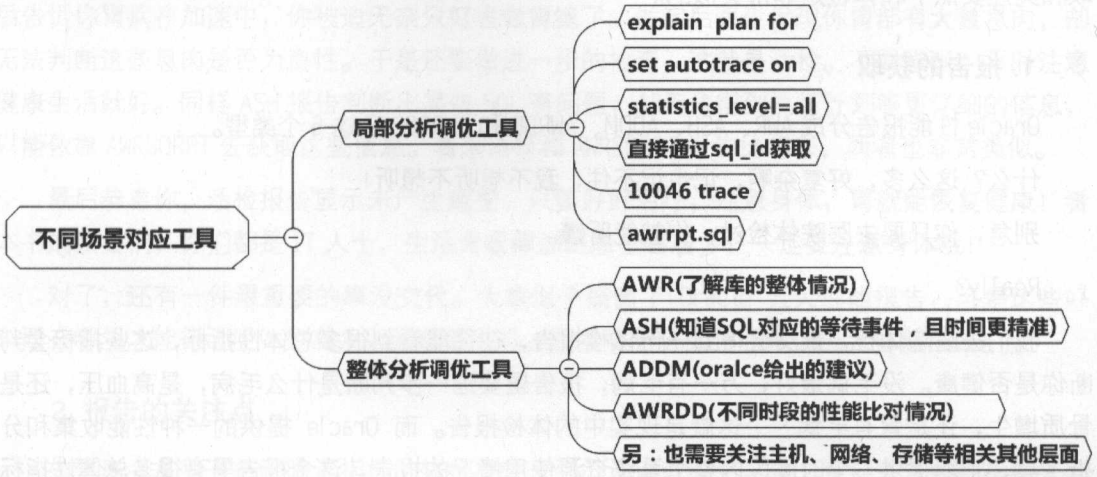


1.1.2 不同场景对应工具

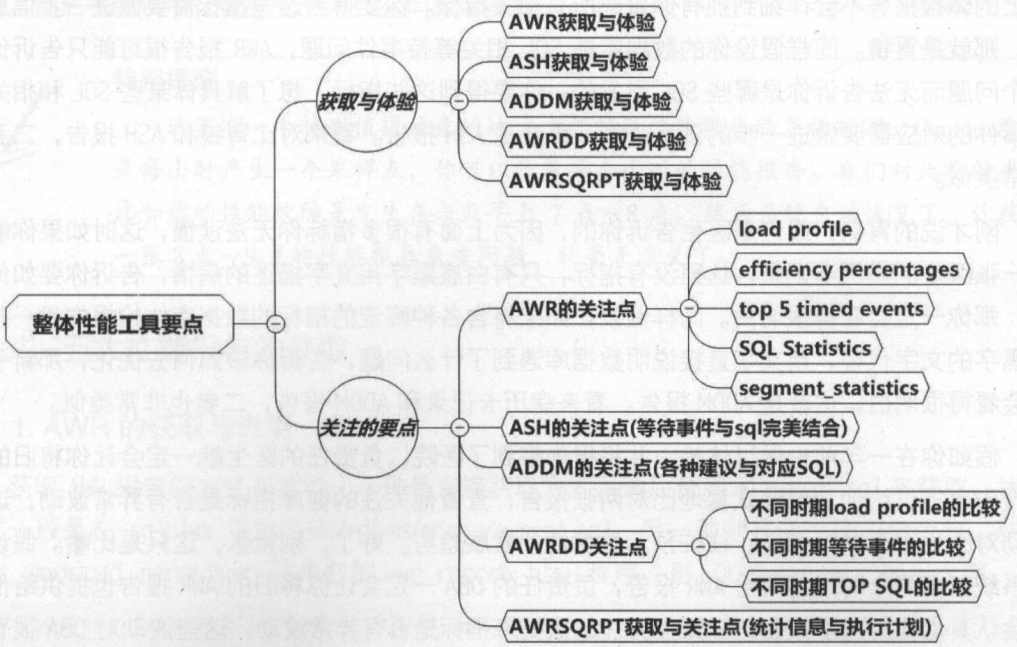
接下来，我们说说这两种场景对应的工具的使用。关于局部分析调优工具，这个其实就是在说 SQL 的执行计划了，这是 SQL 优化最重要的手段之一，通过分析执行计划，我们可以知道 SQL 语句的访问路径，知道它慢在哪里，从而进行 SQL 优化。由于在随后的章节中我们会详细介绍执行计划相关知识，这里就不再细述了。

关于整体的调优工具，这里我们先撇开主机、网络、存储等层面的因素，暂时从数据库的整体层面入手。主要工具有 AWR、ASH、ADDM、AWRDD 这四个工具。其中 AWR 是关注数据库的整体性能的报告；ASH 是数据库中的等待事件与哪些 SQL 具体对应的报告；ADDM 是 Oracle 给出

的一些建议；而 AWRDD 是 Oracle 针对不同时段的性能的一个比对报告，比如今天早上 9 点系统很慢，而昨天这个时候很正常，很多人就想知道今天早上 9 点和昨天早上 9 点有什么不同，于是就有了这个报告。



整体分析调优是必需的，那么我们对此的学习也有规律可循。首先是获取系统整体信息的手段，一般通过报告和日志获取。好比破案一样，这就是收集证据的阶段。接下来要找到蛛丝马迹，那就是如何发现问题。在本书中就是需要关注提取到的这些报告的哪些要点、哪些关键字，具体流程图如下：



1.2 整体性能工具的要点

现代人对健康都比较重视，每年都会进行健康体检。其实数据库性能工具的应用（报告获取和关注要点）和体检是非常类似的。

1. 报告的获取

Oracle 性能报告分成 AWR、ASH、ADDM、AWRDD 和 AWRSQRPT 这 5 个类型。

什么？这么多，好复杂啊，记也记不住，我不想听不想听！

别急，你只要去医院体检过，你就能听懂。

Really?

我们去医院体检，最终会得到一份体检报告，往往能看到很多总体性指标，这些指标会判断你是否健康。没毛病最好，万一有毛病，报告里要进一步判断是什么毛病，是高血压，还是骨质增生，还是胃有毛病……这就是现实中的体检报告。而 Oracle 提供的一种性能收集和分析工具，它能提供一个时间段内整个系统资源使用情况的报告，这个报告里有很多总体性指标来判断系统是否健康。没毛病最好，万一有毛病，问题出在什么模块，是日志切换过于频繁，还是硬解析过大，还是某些 SQL 相关等待事件在耗资源……这就是 AWR 报告。这样看来，体检报告和 AWR 报告非常类似。

假设体检报告说你有胃病，很可能只告诉你胃有问题，却无法告诉你具体啥毛病，因为你手上的体检报告不会详细到拥有你胃部所有相关指标。你要得到这些指标需要做进一步信息收集，那就是胃镜。同样假设你的数据库是 SQL 相关等待事件问题，AWR 报告很可能只告诉你有这个问题而无法告诉你是哪些 SQL 引发的。你要得到这些指标，想了解具体某些 SQL 和相关等待事件的对应需要做进一步的信息收集，那就是 ASH 报告。看来对比胃镜和 ASH 报告，二者也非常类似。

刚才说的胃病，或许是医生告诉你的，因为上面有很多指标你无法读懂，这时如果你能拿到一张医生的病历卡记录，这里没有指标，只有白底黑字用文字描述的病情，告诉你要如何治疗，那你一定会看得很明白。同样假设，如果将含各种晦涩的指标的数据库体检报告用一些白底黑字的文字代替，用文字直接说明数据库遇到了什么问题，告诉你该如何去优化，那新手一定会看得很明白，这就是 ADDM 报告。看来病历卡记录和 ADDM 报告，二者也非常类似。

假如你在一年前也做过体检，并将报告带到了医院，负责任的医生就一定会让你将旧的体检报告也提供给他。他会认真地比对两张报告，查看他关注的健康指标是否有异常波动，这些波动对医生很有参考意义，往往预示着病情的发展趋势。好了，别紧张，这只是比喻。假设你有系统新旧两个时段的两份 AWR 报告，负责任的 DBA 一定会让你将旧的 AWR 报告也提供给他。他会认真地比对两份报告，查看他关注的数据库指标是否有异常波动，这些波动对 DBA 很有参考意义，往往预示着数据库性能瓶颈的发展趋势。Oracle 提供了一个工具能够将两个时段的 AWR 报告合并，并能方便地显示出比对信息，这个工具就是 AWRDD。看来医生分析前后两次体

检报告的动作和 AWRDD 报告比起来，两者也非常类似。

大家知道做胃镜是一件很麻烦的事(类似 ASH 报告)，如果没毛病就没必要让我们遭这罪。可万一体检报告无情地告诉你胃有毛病，甚至是医生分析你前后两次体检报告（类似 ADDM）后告诉你胃病在加速中，你被迫无奈只好去做胃镜了。做完后医生发现你胃部有大量息肉，却无法判断这些息肉是否为良性。于是还要做进一步的检查，这就是活检。不要紧张，平时注意健康生活就好。同样 ASH 报告判断出某些 SQL 有问题，却无法得到执行计划等更详细的信息，只能依靠 AWRSQRPT 去获取这些信息。看来活检和 AWRSQRPT 报告比起来，两者也非常类似。

最后恭喜你，活检报告显示未产生癌变，只要好好治疗，注意身体，胃就能恢复健康！看本书的读者们，你们都是 IT 人士，生活无规律加班熬夜者居多，一定要注意身体哦！

对了，还有一件最重要的事没交代。大家似乎搞懂了 Oracle 五大性能报告，可是这些好东西在哪里才能得到呢？别着急，后续章节马上就会告诉你如何获取这五大性能报告。

2. 报告的关注点

如果患者拿着有各种晦涩指标的体检报告来到门诊请教医生，他一定会关注各种指标来判断患者具体是什么毛病。同样你也会对 Oracle 的性能报告中的各种指标进行关注来判断数据库出了什么毛病。两者非常类似，关注不同的指标，都是为了施救，前者救人，后者救数据库。

听起来是不是很激动，恨不得马上就要开始当救库英雄了！别急，接下来还要告诉你关注什么，然后在案例中让你感受一下什么叫救库英雄。

特别提醒

这里有一个特别值得注意的地方，那就是性能报告的采样时间。Oracle 默认是每小时产生一个采样点，你可以收集每个小时的性能报告。我们对此要敏感，比如你的性能故障是发生在今天早上 7 点～8 点。然后系统自动恢复了，你获取一张 8 点～9 点的性能报告来查问题，就毫无意义了。

1.2.1　五大性能报告的获取

1. AWR 的获取与说明

获取 AWR 报告的方式有两种：一种是直接获取方式，调后台脚本 awrrpt.sql 来获取，执行方式一般是在 sqlplus 下执行@?/rdbms/admin/awrrpt.sql；另一种则是通过调用命令包，获取 dbms_workload_repository 这个包的 awr_report_html 程序，用 SQL 命令的形式输出内容。

```
Select output from table(dbms_workload_repository.awr_report_html
(v_dbid, v_instance_number,v_min_snap_id,v_max_snap_id))
```

（1）直接获取

试验 1（未使用批量提交）：

```
drop table t purge;
create table t ( x int );

exec dbms_workload_repository.create_snapshot();
set timing on

begin
    for i in 1 .. 100000
    loop
       insert into t values (i);
       commit;
    end loop;
end;
/
---接下来手工生成断点，命令如下：
exec dbms_workload_repository.create_snapshot();
--然后在 sqlplus 下执行如下命令
@?/rdbms/admin/awrrpt.sql
```

接下来通过提示就可以生成 awr 报告了，具体步骤略去，详情请扫本章最后的二维码。

试验 2（单机下，正确使用批量提交）：

```
drop table t purge;
create table t ( x int );

exec dbms_workload_repository.create_snapshot();
set timing on
begin
    for i in 1 .. 100000
    loop
       insert into t values (i);
    end loop;
    commit;
end;
/
exec dbms_workload_repository.create_snapshot();
@?/rdbms/admin/awrrpt.sql
```

接下来通过提示就可以生成 awr 报告了，具体步骤略去。

（2）通过调用命令包获取

直接调用工具包的方式，特别适合用在程序自动获取报告的场景。

```
set pagesize 0
set linesize 121
spool d:\awr_commit_frequently.html
```

```
select output from table(dbms_workload_repository.awr_report_html(977587123,1,1920,1921));
spool off
```

注：其中 977587123 是数据库的主机标识，可以在数据库的数据字典中查到，1 是标识实例，如果是 RAC，就有 1 和 2 两个，单机就只有 1。1920 和 1921 是两个断点时间，比如 9 点和 10 点之间。

2. ASH 的获取与说明

获取 ASH 报告的方式也有两种：一种是直接获取方式，调后台脚本 ashrpt.sql 来获取，执行方式一般是在 sqlplus 下执行@?/rdbms/admin/ashrpt.sql；另一种则是通过调用命令包，获取 dbms_workload_repository 这个包的 ash_report_html 程序。用 SQL 命令的形式输出内容。

```
select output from table(dbms_workload_
repository.ash_report_html( dbid,inst_num,l_btime,l_etime)
```

（1）直接获取

```
sqlplus as/ sysdba
--有的环境需要如下操作后才可以正常完成报告输出
alter session set nls_date_language='american' ;
@?/rdbms/admin/ashrpt.sql

Current Instance
~~~~~~~~~~~~~~~~

   DB Id    DB Name      Inst Num Instance
------------------------------------------
  977587123 TEST11G             1 test11g


Specify the Report Type
~~~~~~~~~~~~~~~~~~~~~~~
Enter 'html' for an HTML report, or 'text' for plain text
Defaults to 'html'
输入 report_type 的值:

Type Specified:  html


ASH Samples in this Workload Repository schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Oldest ASH sample available:  10-2月 -14 13:58:14  [  11301 mins in the past]
Latest ASH sample available:  18-2月 -14 10:10:50  [      8 mins in the past]
```

```
Specify the timeframe to generate the ASH report
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Enter begin time for report:

--    Valid input formats:
--      To specify absolute begin time:
--        [MM/DD[/YY]] HH24:MI[:SS]
--        Examples: 02/23/03 14:30:15
--                  02/23 14:30:15
--                  14:30:15
--                  14:30
--      To specify relative begin time: (start with '-' sign)
--        -[HH24:]MI
--        Examples: -1:15  (SYSDATE - 1 Hr 15 Mins)
--                  -25    (SYSDATE - 25 Mins)

Defaults to -15 mins
输入 begin_time 的值:
```

说明：

1. 如果你是一路回车，就是获取最近 5 分钟的 ASH 报告。
2. 如果你根据 Oldest ASH sample available 时间，然后回车，选择的是目前可收集的最长 ASH 运行情况。
3. 你可以选择 Oldest ASH sample available 和 Latest ASH sample available 之间时间，然后输入时长，比如 30 表示 30 分钟，取你要取的任何时段的 ASH 报告。
4. ASH 报告的获取不同于 AWR 的地方在于，快照之间有无重启动作不影响报告的获取。
5. ASH 报告可以直接手工获取，比如 select output from table(dbms_workload_repository.ash_report_html(dbid,inst_num,l_btime,l_etime)。

（2）通过调用命令包获取

直接调用工具包的方式，特别适合用在程序自动获取报告的场景。

```
set pagesize 0
set linesize 121
spool d:\ash_rpt.html
select output from table(dbms_workload_repository.ash_report_html(977587123,1,
SYSDATE-30/1440,SYSDATE-1/1440));
spool off
```

注：其中 977587123 是数据库的主机标识，可以在数据库的数据字典中查到，1 是标识实

例，如果是 RAC，就有 1 和 2 两个，单机就只有 1。SYSDATE-30/1440,SYSDATE-1/1440 分别是开始时间和结束时间。

3. ADDM 的获取与说明

获取 ADDM 报告的方式也有两种，一种是直接获取方式：调后台脚本 addmrpt.sql 来获取，执行方式一般是在 sqlplus 下执行@?/rdbms/admin/addmrpt.sql。另一种则是通过调用命令包的方式获取：调用 dbms_workload_repository 这个包的 addm_report_html 程序。用 SQL 命令的形式输出内容。

```
-- Create an ADDM task.
  DBMS_ADVISOR.create_task (
    advisor_name      => 'ADDM',
    task_name         => 'MYADDM',
    task_desc         => 'MYADDM');
```

（1）直接获取

```
@?/rdbms/admin/addmrpt.sql
```

具体执行过程略去。

（2）通过调用命令包获取

注：直接调用工具包的方式，适合用在自动获取报告的场景。

```
set pagesize 0
set linesize 121
spool d:\addm_rpt.html
SET LONG 1000000 PAGESIZE 0 LONGCHUNKSIZE 1000
COLUMN get_clob FORMAT a80
SELECT dbms_advisor.get_task_report('ADDM_02', 'TEXT', 'ALL') FROM DUAL;
spool off
```

4. AWRDD 的获取与说明

获取 AWRDD 报告一般是用直接获取的方式，这个脚本的交互部分需要输入要进行对比的两个 awr 报告的 begin snap_id 与 end snap_id，然后输入对比结果报告的名称，这里就不详细介绍了，请读者自行试验完成。

直接获取：

```
@?/rdbms/admin/awrddrpt.sql
```

具体略去。

5. AWRSQ 获取与说明

获取 AWRSQRPT 报告的关键之处在于，交互部分要输入所要分析的 SQL 的 SQL_ID，这是关

键之处。而这个 SQL_ID 可以从 AWR 报告中获取。

```
--第一步：创建测试用的表
drop table t1 cascade constraints purge;
drop table t2 cascade constraints purge;
create table t1 AS SELECT * FROM dba_objects ;
create table t2 AS SELECT * FROM dba_objects where rownum<=100 ;

--第二步：快照
exec dbms_workload_repository.create_snapshot();

--第三步：模拟操作
DECLARE
    v_var number;
BEGIN
    FOR n IN 1..1000
    LOOP
        select count(*) into v_var from t1,t2 where t1.object_id=t2.object_id;
    END LOOP;
END;
/


---第四步：再次快照
exec dbms_workload_repository.create_snapshot();

@?/rdbms/admin/awrrpt.sql


---获取 AWR 的过程略去，主要是从报告中找到需要跟踪的 SQL 的 SQL_ID=1rrtf60fmhxkj

@?/rdbms/admin/awrsqrpt.sql
```

以上 5 个报告的获取本身并不难，操作一遍就会了，笔者也会再提供在线操作视频，让大家实际体会一遍。现在关键在于，要明白这 5 个报告的作用和相互之间的区别，搞懂这些，调优之路就算完成过半了。当然，接下来如何分析读懂这五大报告的关键指标就非常重要了，有一些指标你必须关注，否则你就当不了“医生”了。

1.2.2 五大报告关注的要点

1. AWR 的关注点

AWR 报告是五大报告中最全面最重要的一个报告，它的相关指标也显得格外重要。这里我们列出 DB Time、load_profile、efficiency percentages、top 5 events、SQL Statistics、Segment_statistics 这 6 个指标入手分析。

（1）AWR 关注点 1 之 DB Time

DB Time 这个指标主要用来判断当前系统有没有遇到相关瓶颈，是否较为繁忙导致等待时长很长。一般来说，Elapsed 时间乘以 CPU 个数的时间如果结果大于 DB Time，我们认为系统压力不大，反之则压力较大。如下例子中，60.11×64=3847.04<5990.6，说明系统现在还是比较繁忙的。

Host Name	Platform	CPUs	Cores	Sockets	Memory (GB)
ndbisdb1.atlbattery.com	Linux x86 64-bit	64	32	4	125.97

	Snap Id	Snap Time	Sessions	Cursors/Session
Begin Snap:	43943	02-Aug-16 08:00:30	473	29.7
End Snap:	43944	02-Aug-16 09:00:37	811	18.1
Elapsed:		60.11 (mins)		
DB Time:		5,990.06 (mins)		

（2）AWR 关注点 2 之 load_profile

load_profile 这个指标主要用来展现当前系统的一些指示性能的总体参数，比如经典的 Redo size 就是用来显示平均每秒的日志尺寸和平均每个事务的日志尺寸，结合 Transactions 这个每秒事务数的指标，就可以分析出当前事务的繁忙程度。

下图中显示每秒有 6777.1 个事务数，这在现实中几乎不可能，现实中的运营商系统一般在 200 上下比较正常，超过 1000 就属于非常繁忙了。

Load Profile

	Per Second	Per Transaction	Per Exec	Per Call
DB Time(s):	1.0	0.0	0.00	0.68
DB CPU(s):	0.9	0.0	0.00	0.57
Redo size:	3,527,300.9	520.5		
Logical reads:	27,325.2	4.0		
Block changes:	27,351.0	4.0		
Physical reads:	1.9	0.0		
Physical writes:	148.3	0.0		
User calls:	1.5	0.0		
Parses:	31.6	0.0		
Hard parses:	0.3	0.0		
W/A MB processed:	1.7	0.0		
Logons:	0.5	0.0		
Executes:	6,845.1	1.0		
Rollbacks:	0.0	0.0		
Transactions:	6,777.1			

把上图和下面的图进行比较，就非常明显了，下图显示每秒有 0.6 个事务，平均每个事务产生的日志尺寸是 7 位数。这说明系统是一个提交不频繁的处理大任务事件的系统。而上图的尺寸是 3 位数。这里非常容易看出，这是一个提交非常频繁且每个事务都非常小的密集提交系统。

Load Profile

	Per Second	Per Transaction	Per Exec	Per Call
DB Time(s):	0.2	0.4	0.00	1.31
DB CPU(s):	0.2	0.4	0.00	1.28
Redo size:	1,099,405.6	1,883,789.2		
Logical reads:	4,891.1	8,380.8		
Block changes:	9,140.0	15,661.0		
Physical reads:	0.2	0.3		
Physical writes:	0.1	0.2		
User calls:	0.2	0.3		
Parses:	10.2	17.5		
Hard parses:	0.4	0.6		
W/A MB processed:	1.1	1.9		
Logons:	0.0	0.0		
Executes:	4,503.9	7,717.3		
Rollbacks:	0.0	0.0		
Transactions:	0.6			

（3）AWR 关注点 3 之 efficiency percentages

efficiency percentages 是一些命中率指标，其中 Buffer Hit、Library Hit 等都表示 SGA（System global area）的命中率。在下图中 Soft Parse 指标表示共享池的软解析率，在 OLTP 系统中如果该指标低于 90%应当引起你的注意，这表示存在未使用绑定变量的情况。我们通过比对两个报告，可以看出明显差异，如下面系列图所示。

报告 1（未有效地使用绑定变量，产生大量硬解析的场景）。

Load Profile

	Per Second	Per Transaction	Per Exec	Per Call
DB Time(s):	1.0	3.1	0.00	3.09
DB CPU(s):	1.0	3.1	0.00	3.07
Redo size:	532,349.7	1,642,263.2		
Logical reads:	8,866.5	27,352.4		
Block changes:	4,411.6	13,609.3		
Physical reads:	0.3	0.9		
Physical writes:	22.3	68.8		
User calls:	0.3	1.0		
Parses:	2,168.6	6,689.9		
Hard parses:	2,161.6	6,668.3		
W/A MB processed:	0.6	1.7		
Logons:	0.1	0.2		
Executes:	2,170.8	6,696.8		
Rollbacks:	0.0	0.0		
Transactions:	0.3			

Instance Efficiency Percentages (Target 100%)

Buffer Nowait %:	100.00	Redo NoWait %:	100.00
Buffer Hit %:	100.00	In-memory Sort %:	100.00
Library Hit %:	62.53	Soft Parse %:	0.32
Execute to Parse %:	0.10	Latch Hit %:	100.00
Parse CPU to Parse Elapsd %:	100.34	% Non-Parse CPU:	23.34

报告 2（有效地使用绑定变量，进行绑定变量优化后的场景）。

Load Profile

	Per Second	Per Transaction	Per Exec	Per Call
DB Time(s):	1.0	0.5	0.00	0.96
DB CPU(s):	1.0	0.5	0.00	0.95
Redo size:	4,253,662.6	2,038,213.3		
Logical reads:	18,935.5	9,073.3		
Block changes:	35,400.0	16,962.5		
Physical reads:	0.9	0.4		
Physical writes:	0.4	0.2		
User calls:	1.0	0.5		
Parses:	39.7	19.0		
Hard parses:	0.9	0.4		
W/A MB processed:	4.3	2.1		
Logons:	0.0	0.0		
Executes:	17,447.0	8,360.0		
Rollbacks:	0.0	0.0		
Transactions:	2.1			

Instance Efficiency Percentages (Target 100%)

Buffer Nowait %:	100.00	Redo NoWait %:	100.00
Buffer Hit %:	100.00	In-memory Sort %:	100.00
Library Hit %:	99.99	Soft Parse %:	97.81
Execute to Parse %:	99.77	Latch Hit %:	100.00
Parse CPU to Parse Elapsd %:	100.00	% Non-Parse CPU:	99.30

（4）AWR 关注点 4 之 top 5 events

等待事件是衡量数据库整体优化情况的重要指标，通过观察 Top 5 Timed Foreground Events 模块的 Event 和%DB time 两列，可以非常直观地看出当前数据库面临的主要等待事件是什么。下图两个例子分别告诉我们数据库面临锁等待和日志切换等待的情形。

Top 5 Timed Foreground Events

Event	Waits	Time(s)	Avg wait (ms)	% DB time	Wait Class
enq: TX - row lock contention	3	279	93010	99.01	Application
DB CPU		3		0.97	
control file sequential read	141	0	0	0.01	System I/O
db file scattered read	20	0	1	0.01	User I/O
db file sequential read	31	0	0	0.00	User I/O

Top 5 Timed Foreground Events

Event	Waits	Time(s)	Avg wait (ms)	% DB time	Wait Class
log file switch (checkpoint incomplete)	79	75	952	61.81	Configuration
DB CPU		42		34.71	
log file switch completion	39	4	100	3.20	Configuration
db file sequential read	142	0	0	0.03	User I/O
control file sequential read	141	0	0	0.03	System I/O

（5）AWR 关注点 5 之 SQL Statistics

SQL Statistics 分别从几个维度来罗列出 TOP 的 SQL，这是一种简单粗暴但有效的方法。看看执行时长，直接拿出来优化一般都是对的做法。

SQL Statistics

- SQL ordered by Elapsed Time
- SQL ordered by CPU Time
- SQL ordered by User I/O Wait Time
- SQL ordered by Gets
- SQL ordered by Reads
- SQL ordered by Physical Reads (UnOptimized)
- SQL ordered by Executions
- SQL ordered by Parse Calls
- SQL ordered by Sharable Memory
- SQL ordered by Version Count
- Complete List of SQL Text

SQL ordered by Elapsed Time

- Resources reported for PL/SQL code includes the resources used by all SQL statements called by the code.
- % Total DB Time is the Elapsed Time of the SQL statement divided into the Total Database Time multiplied by 100
- %Total - Elapsed Time as a percentage of Total DB time
- %CPU - CPU Time as a percentage of Elapsed Time
- %IO - User I/O Time as a percentage of Elapsed Time
- Captured SQL account for 37.2% of Total DB Time (s): 15
- Captured PL/SQL account for 8.2% of Total DB Time (s): 15

Elapsed Time (s)	Executions	Elapsed Time per Exec (s)	%Total	%CPU	%IO	SQL Id	SQL Module	SQL Text
4.33	100,000	0.00	28.95	92.27	0.00	2n9c7yuww4dx4	SQL*Plus	INSERT INTO T VALUES (:B1)
1.01	1	1.01	6.78	98.48	0.73	bc7gjv3ppdtbz	SQL*Plus	BEGIN dbms_workload_repository...
0.35	1	0.35	2.33	98.62	0.00	dayq182sk41ks		insert into wrh$_memory_target...
0.35	1	0.35	2.33	98.67	0.00	bunssq950snhf		insert into wrh$_sga_target_ad...
0.35	1	0.35	2.32	98.78	0.00	bm2pwrncr8ru6		select sga_size,s_sga_size_fa

（6）AWR 关注点 6 之 Segment Statistics

使用 Segment Statistics 指标进行寻找和判断，也是一个非常直接的优化手段。当我们知道繁忙落在数据库的那个表段是索引段时，优化就变得相对简单了，比如最简单粗暴的方法就是对表和索引进行数据清理和瘦身。

Segment Statistics

- Segments by Logical Reads
- Segments by Physical Reads
- Segments by Physical Read Requests
- Segments by UnOptimized Reads
- Segments by Optimized Reads
- Segments by Direct Physical Reads
- Segments by Physical Writes
- Segments by Physical Write Requests
- Segments by Direct Physical Writes
- Segments by Table Scans
- Segments by DB Blocks Changes
- Segments by Row Lock Waits
- Segments by ITL Waits
- Segments by Buffer Busy Waits

Back to Top

Segments by Row Lock Waits

- % of Capture shows % of row lock waits for each top segment compared
- with total row lock waits for all segments captured by the Snapshot

Owner	Tablespace Name	Object Name	Subobject Name	Obj. Type	Row Lock Waits	% of Capture
LJB	TBS_LJB	T		TABLE	1	100.00

Segments by Logical Reads

- Total Logical Reads: 410,286
- Captured Segments account for 99.4% of Total

Owner	Tablespace Name	Object Name	Subobject Name	Obj. Type	Logical Reads	%Total
LJB	TBS_LJB	T		TABLE	401,328	97.82
SYS	SYSAUX	WRH$_SEG_STAT_OBJ_PK		INDEX	1,648	0.40
SYS	SYSAUX	WRH$_SQL_PLAN_PK		INDEX	576	0.14
SYS	SYSAUX	SCHEDULER$_EVENT_LOG		TABLE	512	0.12
SYS	SYSAUX	WRH$_SQLSTAT	WRH$_SQLSTA_977587123_1897	TABLE PARTITION	272	0.07

2. ASH 的关注点

ASH 是啥？哦，有人想起来了，胃镜。

完成了 ASH 报告的获取后，打开获得的 ASH 报告，其实对于该报告可关注的东西非常直接，就是看看哪些 SQL 和哪些等待事件是相关联的。

如下图所示：

Top SQL with Top Events

SQL ID	Planhash	Sampled # of Executions	% Activity	Event	% Event	Top Row Source	% RwSrc	SQL Text
cafv93454t4jv		1	57.69	enq: TX - row lock contention	57.69	** Row Source Not Available **	57.69	insert into t values ('M', 78,...
aycghy7dbzja1	3650850293	1	20.42	enq: TX - row lock contention	20.42	DELETE	20.42	delete from T WHERE GENDER='M'

3. ADDM 的关注点

ADDM 是啥？哦，是医生的门诊报告。

由于这是 Oracle 的一些分析建议，所以 ADDM 的阅读非常简单，基本上从 FINDING 1、FINDING 2 顺序往下看就可以了。一般是从数据库整体配置和局部 SQL 两方面给出建议。我们看看都能明白，如下图所示：

整体性的建议

```
FINDING 1: 90% impact (133915 seconds)
-------------------------------------
Host CPU was a bottleneck and the instance was consuming 97% of the host CPU.
All wait times will be inflated by wait for CPU.

   RECOMMENDATION 1: Host Configuration, 72% benefit (106700 seconds)
      ACTION: Consider adding more CPUs to the host or adding instances
         serving the database on other hosts.
      ACTION: Also consider using Oracle Database Resource Manager to
         prioritize the workload from various consumer groups.

FINDING 1: 100% impact (107131 seconds)
---------------------------------------
Significant virtual memory paging was detected on the host operating system.

   RECOMMENDATION 1: Host Configuration, 100% benefit (107131 seconds)
      ACTION: Host operating system was experiencing significant paging but no
         particular root cause could be detected. Investigate processes that
         do not belong to this instance running on the host that are consuming
         significant amount of virtual memory. Also consider adding more
         physical memory to the host.
```

局部 SQL 建议

```
---------------------------------
发现 SQL 语句正处于行锁定等待。

   建议案 1: 应用程序分析
   估计的收益为 .39 个活动会话，占总活动的 72.36%。
   ---------------------------------
   操作
      在 INDEX "LJB.GENDER_IDX" (对象 ID 为 110057) 中检测到了严重的行争用。使用
   指定的阻塞 SQL
      语句在应用程序逻辑中跟踪行争用的起因。
      相关对象
         ID 为 110057 的数据库对象。
   原理
      SQL ID 为 "cafv93454t4jv" 的 SQL 语句在行锁上被阻塞。
      相关对象
         SQL_ID 为 cafv93454t4jv 的 SQL 语句。
         insert  into t values ('M',78, 'young','TTT')
   原理
      具有 ID 130 和序列号 423 (在实例号 1 中) 的会话是构成此建议案中的优化建议
   的 98% 的阻塞会话。

   建议案 2: 应用程序分析
```

4. AWRDD 的关注点

AWRDD 是啥？哦，是医生在看你前后两次体检报告，在比较指标的变化。其实这个关注点很简单，基本上就是 AWR 关注什么，AWRDD 就关注什么，没什么特别的，简单列举如下。

（1）AWRDD 关注点 1 之不同时期 load profile 的比较

Load Profile

	1st per sec	2nd per sec	%Diff	1st per txn	2nd per txn	%Diff
DB time:	0.06	0.01	-83.33	0.49	0.07	-85.71
CPU time:	0.06	0.01	-83.33	0.46	0.07	-84.78
Redo size:	3,795.59	1,752.67	-53.82	29,419.68	14,621.05	-50.30
Logical reads:	6,611.66	314.35	-95.25	51,247.14	2,622.38	-94.88
Block changes:	26.72	23.71	-11.26	207.09	197.81	-4.48
Physical reads:	18.55	0.82	-95.58	143.82	6.84	-95.24
Physical writes:	0.94	0.66	-29.79	7.32	5.52	-24.59
User calls:	0.30	0.26	-13.33	2.29	2.14	-6.55
Parses:	3.45	2.55	-26.09	26.71	21.24	-20.48
Hard parses:	0.17	0.08	-52.94	1.30	0.69	-46.92
W/A MB processed:	41,121.71	31,591.71	-23.18	318,735.20	263,543.74	-23.18
Logons:	0.06	0.06	0.00	0.47	0.51	8.51
Executes:	17.04	7.87	-53.81	132.05	65.69	-50.25
Transactions:	0.13	0.12	-7.69			

（2）AWRDD 关注点 2 之不同时期等待事件的比较

Top Timed Events

- Events with a "-" did not make the Top list in this set of snapshots, but are displayed for comparison purposes

1st						2nd					
Event	Wait Class	Waits	Time(s)	Avg Time(ms)	%DB time	Event	Wait Class	Waits	Time(s)	Avg Time(ms)	%DB time
CPU time			953.43		93.45	CPU time			90.20		104.02
db file scattered read	User I/O	45,907	29.23	0.64	2.87	os thread startup	Concurrency	430	8.27	19.23	9.54
os thread startup	Concurrency	653	10.70	16.38	1.05	control file sequential read	System I/O	12,008	4.90	0.41	5.65
control file sequential read	System I/O	25,823	9.97	0.39	0.98	control file parallel write	System I/O	3,816	2.48	0.65	2.86
db file sequential read	User I/O	15,634	5.81	0.37	0.57	log file parallel write	System I/O	2,423	0.74	0.30	0.85
-control file parallel write	System I/O	5,785	3.67	0.63	0.36	-db file sequential read	User I/O	2,571	0.71	0.28	0.82
-log file parallel write	System I/O	5,484	1.97	0.36	0.19	-db file scattered read	User I/O	626	0.37	0.59	0.42

（3）AWRDD 关注点 3 之不同时期 TOP SQL 的比较

SQL Statistics

- Top SQL Comparison by Elapsed Time
- Top SQL Comparison by CPU Time
- Top SQL Comparison by I/O Time
- Top SQL Comparison by Buffer Gets
- Top SQL Comparison by Physical Reads
- Top SQL Comparison by UnOptimized Read Requests
- Top SQL Comparison by Executions
- Top SQL Comparison by Parse Calls
- Top SQL Comparison by Sharable Memory
- Top SQL Comparison by Version Count
- Top SQL Comparison by Cluster Wait Time

Top SQL Comparison by Elapsed Time

- Ordered by absolute value of 'Diff' column of 'Elapsed Time % of DB time' descending
- '#Plans' column indicates the number of distinct execution plans for the statement in 1st and 2nd periods and in Both periods combined
- '1st Total' and '2nd Total' show respective running totals for '1st' and '2nd' columns of 'Elapsed Time % of DB time'
- DB time First: 1,020.23 seconds, Second: 86.71 seconds
- Captured SQL Elapsed Time First: 891.60 seconds, Second: 62.11 seconds
- Captured SQL Elapsed Time as a % of DB time First: 87.39%, Second: 71.63%
- Captured PL/SQL Elapsed Time as a % of DB time First: 83.04%, Second: 63.7%
- Common SQL Elapsed Time as a % of DB time First: 14.3%, Second: 58.35%

	Elapsed Time % of DB time					Elapsed Time (ms) per Exec		#Exec/sec (DB time)		CPU Time (ms) per Exec		I/O Time (ms) per Exec		Physical Reads per Exec		#Rows Processed per Exec		#Executions		#Plans	
SQL Id	1st	1st Total	2nd	2nd Total	Diff	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st/2nd/Both	SQL Text
9kyv274yjgdtf	58.70	58.70			-58.70	598,899		0.00		594,114		50		1,042.00		1.00		1		1/	DECLARE v_var number; BEGIN FO...
48k1njhd4ras3	57.57	116.27			-57.57	6		98.02		6		0		0.01		1.00		100,000		1// 1	SELECT COUNT(*) FROM T
98vyc4xxkw38t			31.60	31.60	31.60		13,699		0.02		13,556		57		1,044.00		1.00		2		DECLARE v_var number; BEGIN FO...
1rrtf60fmhxkj			31.33	62.92	31.33		14		23.06		13		0		1.04		1.00		2,000	/ 1/ 1	SELECT COUNT(*) FROM T1 T2 WH...

5. AWRSQRPT 的关注点

AWRSQRPT 是啥？哦，有人想起来了，活检。别打颤！

其实没啥，就是看看 AWR 和 ASH 里看不到的东西。都有啥呢？比如执行计划的相关细节，关于执行计划我们会在后面详细说明。这里要特别注意一点，Oracle 的执行计划可能会随着环境的变化而变化，会随着数据的变化而变化，因此可能会产生多个执行计划，这个 AWRSQRPT 就会出现多个执行计划。具体详见下面系列图。

（1）Plan statistics

Plan Statistics

- % Total DB Time is the Elapsed Time of the SQL statement divided into the Total Database Time multiplied by 100

Stat Name	Statement Total	Per Execution	% Snap Total
Elapsed Time (ms)	13,564	13.56	92.27
CPU Time (ms)	13,385	13.38	91.76
Executions	1,000		
Buffer Gets	1,051,075	1,051.08	99.48
Disk Reads	1,044	1.04	99.90
Parse Calls	1	0.00	0.36
Rows	1,000	1.00	
User I/O Wait Time (ms)	55		
Cluster Wait Time (ms)	0		
Application Wait Time (ms)	0		
Concurrency Wait Time (ms)	0		
Invalidations	0		
Version Count	1		
Sharable Mem(KB)	14		

Back to Plan 1(PHV: 4274056747)
Back to Top

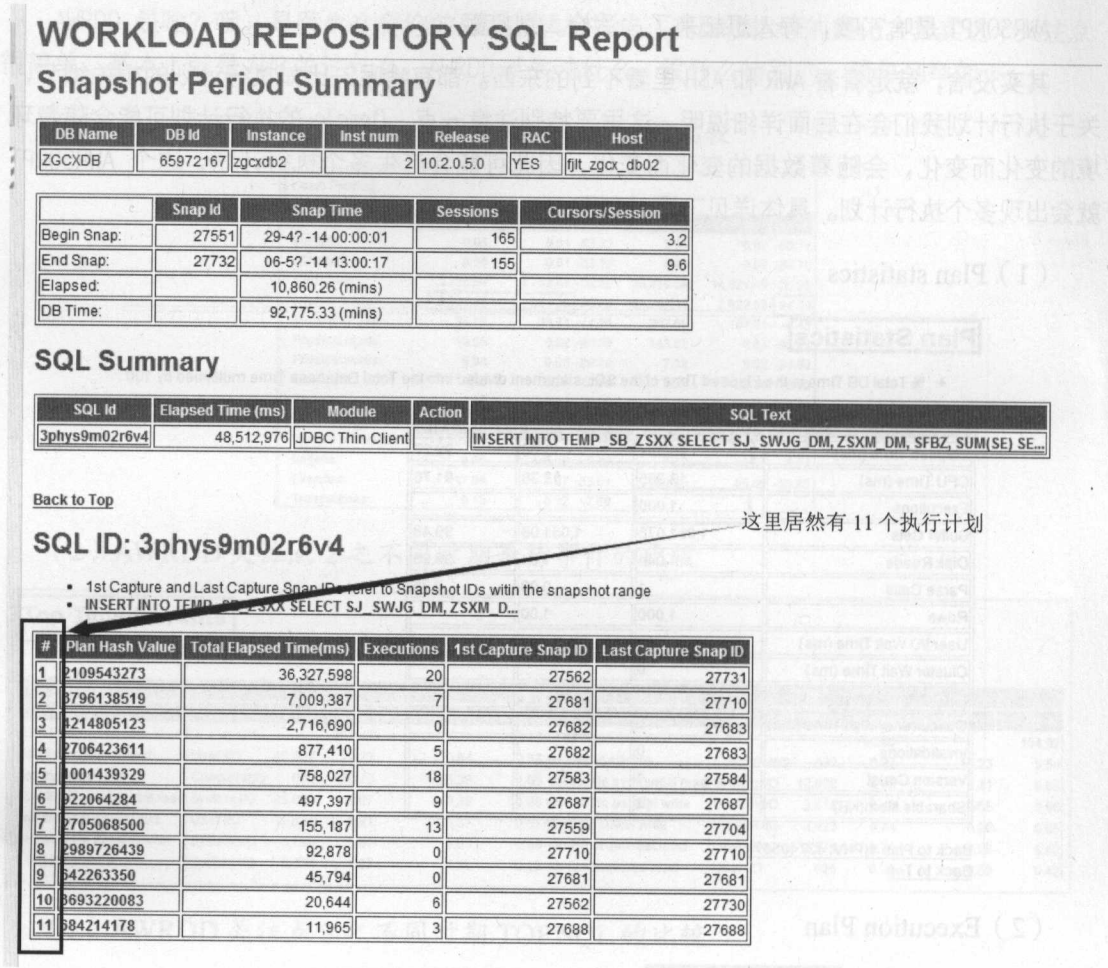
（2）Execution Plan

Execution Plan

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				296 (100)	
1	SORT AGGREGATE		1	26		
2	HASH JOIN		100	2600	296 (1)	00:00:04
3	TABLE ACCESS FULL	T2	100	1300	3 (0)	00:00:01
4	TABLE ACCESS FULL	T1	69217	878K	292 (1)	00:00:04

- dynamic sampling used for this statement (level=2)

（3）是否有多个执行计划



WORKLOAD REPOSITORY SQL Report

Snapshot Period Summary

DB Name	DB Id	Instance	Inst num	Release	RAC	Host
ZGCXDB	65972167	zgcxdb2	2	10.2.0.5.0	YES	fjlt_zgcx_db02

	Snap Id	Snap Time	Sessions	Cursors/Session
Begin Snap:	27551	29-4? -14 00:00:01	165	3.2
End Snap:	27732	06-5? -14 13:00:17	155	9.6
Elapsed:		10,860.26 (mins)		
DB Time:		92,775.33 (mins)		

SQL Summary

SQL Id	Elapsed Time (ms)	Module	Action	SQL Text
3phys9m02r6v4	48,512,976	JDBC Thin Client		INSERT INTO TEMP_SB_ZSXX SELECT SJ_SWJG_DM, ZSXM_DM, SFBZ, SUM(SE) SE...

Back to Top

这里居然有 11 个执行计划

SQL ID: 3phys9m02r6v4

- 1st Capture and Last Capture Snap IDs refer to Snapshot IDs witin the snapshot range
- INSERT INTO TEMP_SB_ZSXX SELECT SJ_SWJG_DM, ZSXM_D...

#	Plan Hash Value	Total Elapsed Time(ms)	Executions	1st Capture Snap ID	Last Capture Snap ID
1	2109543273	36,327,598	20	27562	27731
2	3796138519	7,009,387	7	27681	27710
3	4214805123	2,716,690	0	27682	27683
4	2706423611	877,410	5	27682	27683
5	1001439329	758,027	18	27583	27584
6	922064284	497,397	9	27687	27687
7	2705068500	155,187	13	27559	27704
8	2989726439	92,878	0	27710	27710
9	542263350	45,794	0	27681	27681
10	3693220083	20,644	6	27562	27730
11	584214178	11,965	3	27688	27688

1.3 案例的分享与交流

说了这么多，我们来看几个相关案例，体会使用工具进行整体优化的重要性。

1.3.1 和并行等待有关的案例

这是来自某政府系统的一个平台的案例，请看下图，这是 AWR 报告的 Top 5 Timed Events 的展现，可以看出当前数据库的等待事件主要是 PX Deq 相关的等待，这属于滥用并行等待导致系统资源紧张的一个案例。

Top 5 Timed Events

Event	Waits	Time(s)	Avg Wait(ms)	% Total Call Time	Wait Class
PX Deq Credit: send blkd	21,758	2,954	136	73.6	Other
CPU time		714		17.8	
log file sync	201,213	202	1	5.0	Commit
db file scattered read	33,757	163	5	4.1	User I/O
log file parallel write	182,646	123	1	3.1	System I/O

该案例暴露出的问题比想象中更严重，因为该系统的不少表和索引的属性被设置了并行度，这导致所有对这些表和索引的访问都成了并行访问。后续解决思路就是将表和索引的并行属性去掉。将一些需要并行处理的大任务进行时间切割，确认部分大任务是可以放在凌晨业务低峰期执行的，就设置了并行的 Hint 任务，让部分 SQL 在夜间并行执行，大部分 SQL 在白天正常执行，从而系统恢复正常，业务也能顺利开展。

1.3.2　和热块竞争有关的案例

接下来我们再看一个案例，这是某运营商的系统，从 AWR 报告的 Top 5 Timed Events 等待事件主要是 gc buffer busy 来看，当前系统主要等待事件是热块竞争的等待。

Top 5 Timed Events

Event	Waits	Time(s)	Avg Wait(ms)	% Total Call Time	Wait Class
gc buffer busy	78,233,089	80,322	1	53.2	Cluster
latch: cache buffers chains	18,383,189	21,595	1	14.3	Concurrency
CPU time		13,883		9.2	
gc current block 2-way	3,123,748	4,076	1	2.7	Cluster
gc cr multi block request	4,928,809	3,641	1	2.4	Cluster

等待事件对应的 SQL 主要有哪些，我们其实可以通过对应时间段的 ASH 报告分析出来，比如下图就是和 AWR 的对应。

Top SQL Statements

SQL ID	Planhash	% Activity	Event	% Event	SQL Text
c9whwr7j93m5g	2144253418	36.75	CPU + Wait for CPU	22.63	select * from (select count(0)...
			gc buffer busy	10.99	
			latch: cache buffers chains	1.43	
a5nr3x00d0kkh	3496244507	17.25	CPU + Wait for CPU	8.79	select * from (select rownum s...
			gc buffer busy	5.39	
			read by other session	1.96	
ghyfq5q1jqp4a	4021807939	10.13	gc buffer busy	7.56	SELECT * FROM (SELECT ROWNUM S...
			CPU + Wait for CPU	1.34	
6w4y6hbp3u1w2	3694320015	9.49	gc buffer busy	5.60	SELECT COUNT(*) COUNT FROM (se...
			gc current block 2-way	1.26	
			CPU + Wait for CPU	1.12	
fx8wpaf56xk4v	1030238553	3.81	CPU + Wait for CPU	3.67	SELECT COUNT(*) COUNT FROM (se...

将 AWR 报告和 ASH 报告结合起来看，往往可以找出具体需要优化的 SQL。在本案例中，我们发现两个节点共同访问一些对象导致热块竞争。后续通过一系列改造，让不同的业务跑在不

同的节点上，从而避免了两个节点访问同一个对象，问题得以缓解。

1.3.3 和日志等待有关的案例

这是一个典型的案例，从 Transactions 达到 800 多，可以看出事务非常繁忙，再从 Per Transaction 才 1000 左右，可以看出每个事务非常小。这说明了系统存在事务未批量提交的情况。这种情况一般出现在循环中，把提交写到循环里面的情况。后续通过排查，发现果真是如此原因。

接下来的 log file switch(checkpoint incomplete) 和 log file sync 的相关等待正是由于日志切换过于频繁导致的等待，这正是如前所述，未批量提交导致。

Load Profile

	Per Second	Per Transaction
Redo size:	869,060.90	1,035.92
Logical reads:	44,965.46	53.60
Block changes:	4,835.56	5.76
Physical reads:	281.46	0.34
Physical writes:	426.80	0.51
User calls:	7,847.72	9.35
Parses:	5,898.96	7.03
Hard parses:	0.32	0.00
Sorts:	0.89	0.00
Logons:	0.04	0.00
Executes:	5,903.07	7.04
Transactions:	838.92	

Top 5 Timed Events

Event	Waits	Time(s)	Avg Wait(ms)	% Total Call Time	Wait Class
db file sequential read	457,750	65,306	143	48.4	User I/O
log file switch (checkpoint incomplete)	46,779	28,906	618	21.4	Configuration
log file sync	1,537,513	22,539	15	16.7	Commit
enq: SQ - contention	34,250	11,964	349	8.9	Configuration
db file parallel write	16,692	8,331	499	6.2	System I/O

1.3.4 新疆某系统的前台优化

如下是新疆某运营商的优化案例，我们通过 Top 5 Timed Events 等待事件发现了瓶颈主要在 IO。接下来我们迅速到 Tablespace IO Stats 模块去查看，如下图所示：

Tablespace IO Stats

- ordered by IOs (Reads + Writes) desc

Tablespace	Reads	Av Reads/s	Av Rd(ms)	Av Blks/Rd	Writes	Av Writes/s	Buffer Waits	Av Buf Wt(ms)
BOSSWG_CFG	1,785,153	490	13.90	10.09	11,680	3	393,996	10.93
TEMP	77,812	21	2.34	4.00	24,267	7	0	0.00
BOSSWG_INDEX	471	0	18.45	1.00	36,375	10	8,187	14.00
SYSAUX	4,232	1	12.54	1.44	330	0	0	0.00
BOSSWG_TECH_DATA	2,269	1	20.67	3.56	1,752	0	112,566	7.86
UNDOTBS02	0	0	0.00	0.00	3,308	1	1,299	0.48
BOSSWG_PERF_DATA	209	0	47.61	1.00	1,791	0	306	6.96
SYSTEM	200	0	13.80	1.00	182	0	0	0.00

果然是有点问题。这个 Av Rd(ms) 项表示平均一次物理读花费的时间（单位为 ms）。有一种说法是， AV RD(MS)大于 7 就说明系统有严重的 IO 问题，其中 BOSSWG_PERF_DATA 居然达到了 47，这说明当前的存储 IO 存在瓶颈。后续通过改善存储解决了问题。

1.3.5　浙江某系统的调优案例

这个案例来自浙江某生产系统，我们通过 Top 5 Timed Events 等待事件发现了瓶颈主要在 gc buffer busy 等待事件，这和 1.3.4 节的案例类似。不过 AWR 报告非常强大，你通过各个细节都可以很有收获，从而找到解决问题的方法，比如你此时直接定位到 Segments by Global Cache Buffer Busy 模块，如下图所示：

Segments by Global Cache Buffer Busy

- % of Capture shows % of GC Buffer Busy for each top segment compared
- with GC Buffer Busy for all segments captured by the Snapshot

Owner	Tablespace Name	Object Name	Subobject Name	Obj. Type	GC Buffer Busy	% of Capture
BOSSWG	BOSSWG_CFG	CUST_ZJ_DECLARATION		TABLE	376,463,168	97.49
BOSSWG	BOSSWG_INDEX	TG_OPERATE_LOG		TABLE	7,014,883	1.82
BASEDBA	BOSSWG_TECH_DATA	TACHE_HIS		TABLE	1,418,268	0.37
BOSSWG	BOSSWG_CFG	NE_ALARM_LIST		TABLE	315,800	0.08
BASEDBA	BOSSWG_TECH_DATA	STAFF_EVENT_HIS		TABLE	245,348	0.06

通过观察 segments by global cache buffer busy 的对象，我们找到了相关需要优化的表。最后我们结合业务，通过对该表瘦身、增加分区、避免两个节点同时访问的方案，优化了对应 SQL 的性能。

1.4　本章总结延伸与习题

1.4.1　总结延伸

场景再现

整体性能工具获取					
工具	作用	类比	获取报告的方式		关注的要点
			直接调用	通过命令获取	
AWR	关注数据库的整体性能报告	体检报告	"@?/rdbms/admin/awrrpt.sql"	select output from table(dbms_workload_repository.awr_report_html (v_dbid, v_instance_number,v_min_snap_id, v_max_snap_id));	load_profile efficiency percentages top 5 events SQL Statistics segment_statistics

续表

整体性能工具获取					
工具	作用	类比	获取报告的方式		关注的要点
			直接调用	通过命令获取	
ASH	ASH 关注数据库中的等待事件与哪些 SQL 具体对应	胃镜	"@?/rdbms/admin/ashrpt.sql"	select output from table(dbms_workload_repository.ash_report_html(dbid,inst_num,l_btime,l_etime));	等待事件与具体的 SQL 完美结合
ADDM	Oracle 给出的一些建议	病历卡记录	"@?/rdbms/admin/addmrpt.sql"	SELECT dbms_advisor.get_task_report('ADDM_02', 'TEXT', 'ALL') FROM DUAL;	整体性的建议 局部 SQL 建议
AWRDD	针对不同时段的性能的一个比对报告	医生分析前后两次体检报告的动作	"@?/rdbms/admin/awrddrpt.sql"	略去	不同时期 load profile 不同时期等待事件 不同时期 TOP SQL
AWRSQ	具体某个 SQL 的执行计划，可以保存多个执行计划	活检	"@?/rdbms/admin/awrsqrpt.sql"	略去	Plan statistics Execution Plan

我们来假想一下，学习完这章内容的小王，再回到从前，会有怎样的经历呢？

小王知道解决问题要先整体再局部，从此不再盲目解决问题了。关于整体性方面，他不仅知道了五大性能报告的具体用途，还明白了如何获取到这些报告，甚至还知道如何解读这些报告。如今的他会对着获取到的这几张报告眉头紧锁，认真研读，自言自语，不住点头，指点江山，激扬文字……仿佛一位住在地下室的青年，正在思考国家下一步该怎么走？如何突破美国封锁？如何收复台湾？如何保住南沙钓鱼岛……

他站得高，看得远，这下，应该成功了。

脑洞大开

停，住在地下室的青年，画风有些不对，这是什么比喻？不够高大上嘛。

来，提升一下。

虽然小王可以获取到这些报告，但是毕竟是手工获取，一张张得到，速度慢，咋办？

可以这样，手指头一点，执行一个命令，就自动获取到这几大性能报告。

如果你不知道这些性能报告获取的时段，对方又不能明确告诉你，咋办？

可以这样，自动去系统里判断资源消耗比较大的时段，获取到这个时段的报告。

数据库告警日志、主机信息，这些 OS 层面的东西，能一并获取到吗？

如果你想要，我就一并给你。

从 AWRSQRPT 报告里得到了执行计划等更详细的信息，我想看到更细致的，如对应的表和索引等信息，可以吗？

如果你想要，我就一并给你。

这么神奇，我想马上就提升！

别着急，请继续阅读下一章吧。

1.4.2　习题训练

习题 1：SQL 整体调优工具有哪些？

习题 2：请说说 SQL 优化方法论。

习题及疑问的邮件发送地址与本章总结及更多内容，可扫二维码获得。

第 2 章　风驰电掣——有效缩短 SQL 优化过程

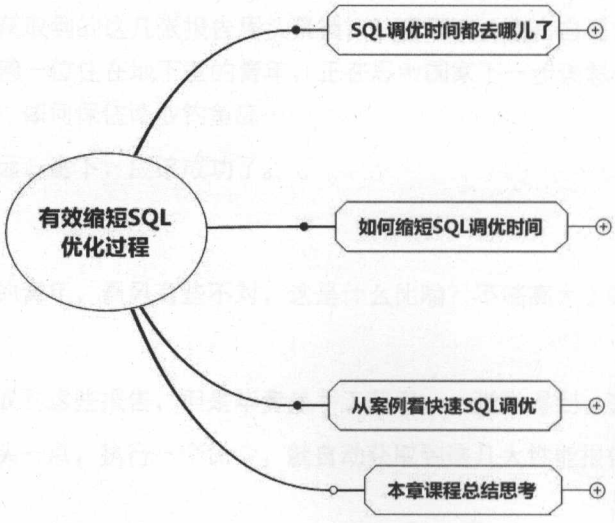
02 风驰电掣——
有效缩短 SQL 优化过程

时间都去哪儿了？

第 1 章似乎让我们有些热血澎湃，不过提升效率的重点内容会出现在本章，能否从地下室搬出来，就在此一举了。

不知大家还记不记得前言故事 1 中的一个细节，小王发现问题整整花了 1 个小时，这时间可不短啊。如果系统故障要花整整 1 小时才能找到，那这 1 小时时间对客户来说，是多么难熬的 1 小时啊。而且，更可悲的是，这 1 小时小王并没有解决问题，故障依旧。

有人问，为啥会花费 1 小时时间呢？嗯，本章就开始探讨这个问题。首先说明调优时间都花在哪儿，接下来分析如何缩短，然后结合案例来巩固知识，最后大家思考回顾。总体学习思路如下图所示：



2.1　SQL 调优时间都去哪儿了

我们先来分析调优时间都去哪儿了，分析 1 小时方得到解决方案固然让人接受不了，事实其实更糟糕，小王是只花费 1 小时吗?

显然不止 1 小时!

他到下午还在解决这个故障，甚至第 2 天他还在解决这个问题，这已经 1 天了。小王只花了 1 天时间吗?

显然不止 1 天!

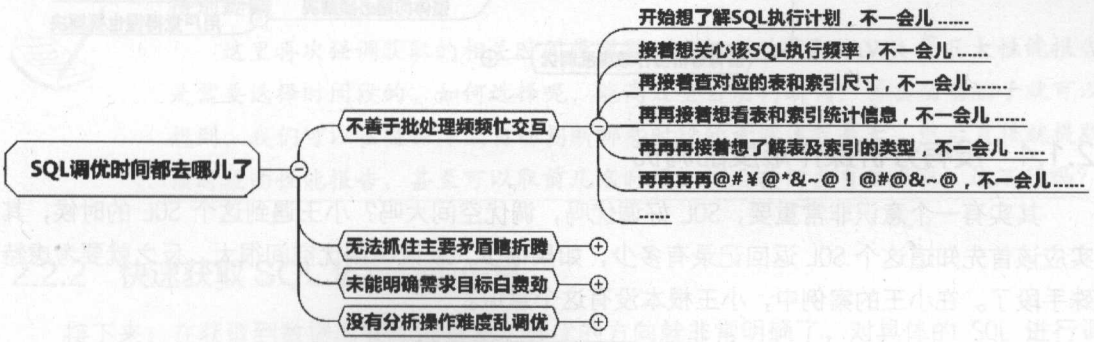
实际情况是小王最后崩溃了无法解决问题。

显然这是无限期!

面对这么凄惨的事情，我这里总结出几点经验，来分析这个问题。

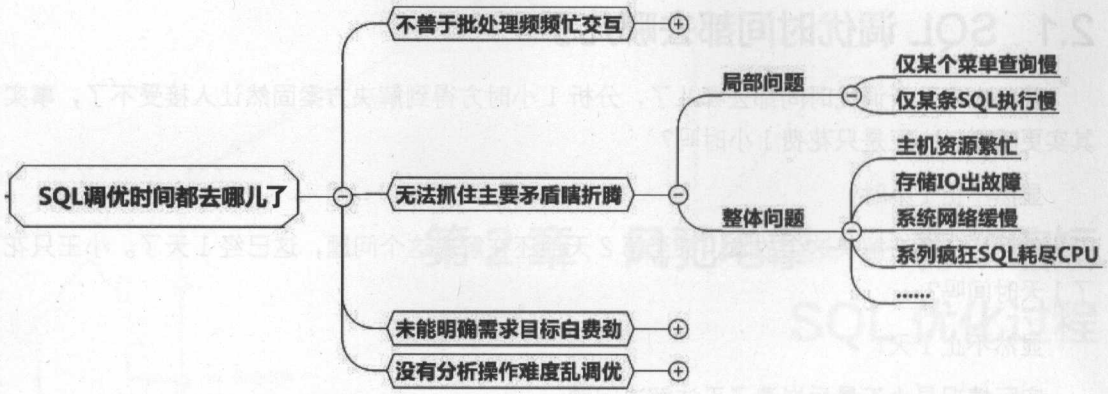
2.1.1　不善于批处理频频忙交互

故事中没有细说小王和求助者之间的交互细节，但是从花了整整 1 小时才发现需要加索引来看，小王的工作效率肯定不高。我们可以猜测他发生了什么事。也许小王本身无法访问这个系统，需要通过对方执行自己的脚本来反馈结果。于是他想了解这个语句的返回记录的量，又想要了解这个语句的执行计划，又想了解这个语句对应的表和索引的信息……电话、QQ、邮件不断地交互，于是 1 小时就这么过去了。就好像如下图所示的这样:



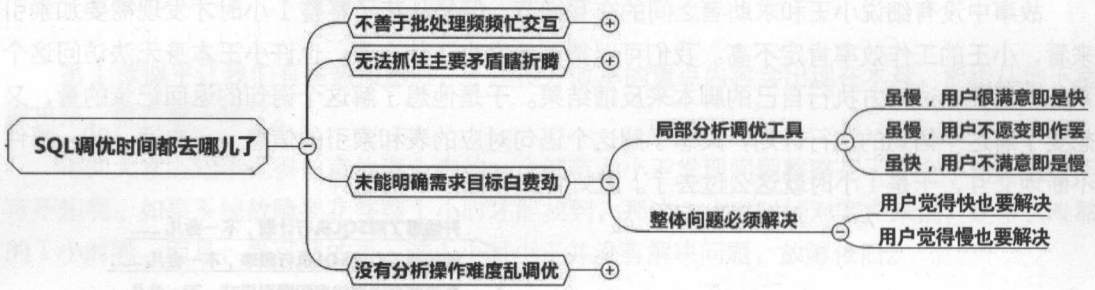
2.1.2　无法抓住主要矛盾瞎折腾

小王没有从整体出发来考虑问题，没有想明白是整体问题还是局部问题，一路抓瞎，无端耗费时间却徒劳无功。



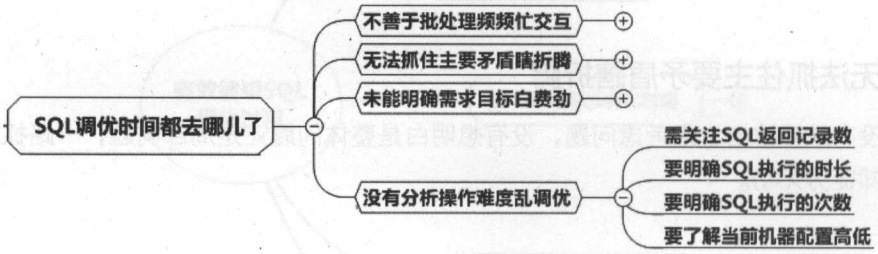
2.1.3 未能明确需求目标白费劲

有的 SQL 其实执行得虽然慢，但是客户并不是很在意，这样的 SQL 除非是耗尽资源影响到全系统，否则不见得就需要立即优化。不过这是我提出的一个重要意识，在小王的案例里，这个问题倒是不明显，因为这个 SQL 客户已经明显感觉慢，提出要优化了。



2.1.4 没有分析操作难度乱调优

其实有一个意识非常重要，SQL 好调优吗，调优空间大吗？小王遇到这个 SQL 的时候，其实应该首先知道这个 SQL 返回记录有多少，如果很少，就说明调优空间很大。反之就要考虑特殊手段了。在小王的案例中，小王根本没有这个意识。



2.2　如何缩短 SQL 调优时间

2.2.1　先获取有助调优的数据库整体信息

如何缩短 SQL 调优时间，我觉得要先把你进行 SQL 优化的思路理顺，当你要优化 SQL 时，你的一般流程是什么？

首先要知道整个数据库的运行情况吧，我们上一章中介绍过数据库 AWR 报告等调优工具，不过介绍得并不全面，因为 AWR 报告等是在数据库出问题时的利器。可是如果数据库当前没有出问题呢？其实不见得，很多时候系统没问题是因为你没触发这个问题，其实是有问题的。比如某表的索引失效了，某 SQL 访问该列时一定只能走全表扫描；比如某表的属性被设置了并行度，这意味着所有扫描该表的 SQL 都会并行执行，这可能会产生严重的资源争用从而让系统瘫痪；比如你的全局临时表被收集了统计信息，访问该表的 SQL 就可能会出现错误的执行计划，等等。不过你的 AWR 报告却可能发现不了这些问题，比如该时段和这些对象相关联的 SQL 根本就没有执行。没发现问题并不代表没有问题！

因此我们需要获取所有可能有问题的对象，同时也需要一键获取所有的相关时段的 AWR 等数据库整体性能报告，获取数据库的整体信息。假如这些信息能一键快速获取，那解决问题的效率肯定会高很多。

如何一键高效获取呢？嗯，纯干货！详情请扫本章最后的二维码。

特别提醒

这里再次强调获取的相关时间段问题，我们自动获取 AWR 等几大性能报告是需要选择时间段的，如何选择呢，能高效全自动判断吗？其实动动脑子就可以想到，我们可以在数据库里自动判断哪些时段的资源消耗最大，然后直接就提取该时段的性能报告，甚至可以取前几名时段，然后我们去自动提取，这不行吗？

2.2.2　快速获取 SQL 运行台前信息

接下来，在获取到数据库整体信息后，调优的方向就非常明确了，对具体的 SQL 进行调优。执行计划是 SQL 调优的重要武器，通过分析 SQL 计划，我们可以判断 SQL 的访问路径是否高效，从而进行调整优化。关于执行计划的获取手段有 6 种之多，这是为啥呢？各有啥区别呢？答案依然展示在后面的章节中。

还需要将执行计划和运行时的统计信息结合在一起分析，这样才会更准确。比如 SQL 产生了多少逻辑读，多少物理读，是否有排序，是否有递归调用，等等。具体细节依然见后续章节。

这里再来一个干货；一键获取 SQL 的执行计划（6 种中最实用的一种）。详情请扫本章最后的二维码。

2.2.3 快速拿到 SQL 关联幕后信息

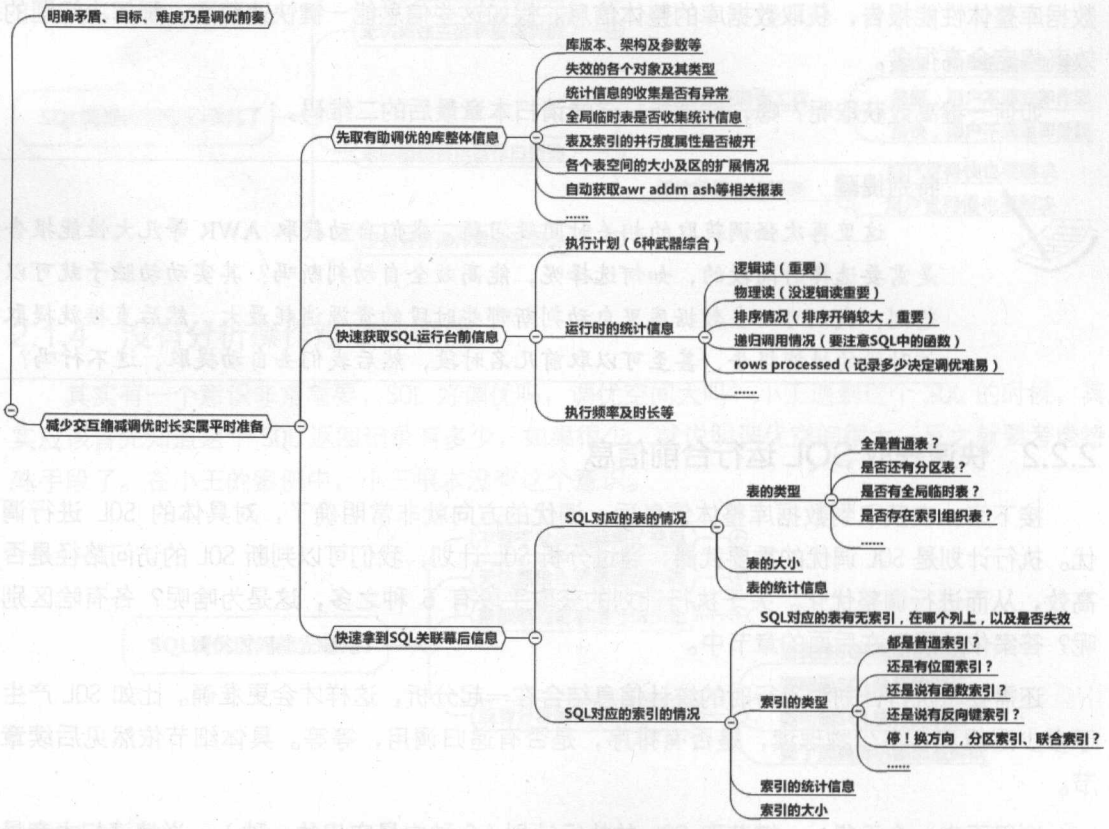
当获取到 SQL 的执行计划后，诸多的确认都和该 SQL 对应的表和索引有关。比如当我们怀疑驱动表的顺序有错时，我们就会去看看这些表的实际大小和对应的统计信息是否准确；我们也关心表的类型是什么，比如是否是分区表，在哪个列有分区，分区的类型是什么，等等。

除了关注表的信息，我们也很关心索引的信息。比如看到执行计划中非常适合走索引的查询走了全表扫描，我们就会去看看是否该列无索引，如果发现有，就看看此列索引是否失效了。一般我们也会关心索引的类型是什么，是 Btree 索引还是位图索引还是函数索引；是单列索引还是组合索引，如果是组合索引，哪列在前；如果索引建在分区表上，我们还关心是全局索引还是局部索引，等等。

总之，我们很希望能一目了然地掌握该 SQL 涉及的所有表和列的相关信息，最好一键就展现在我们面前。这样，解决问题就非常高效了！

大家可能已经猜到了，我们又要来一个一键获取了，这次获取的是 SQL 对应表和索引的相关信息，没错，你猜对了。干货脚本通过扫本章最后的二维码获取。

总体情况，详见下图：



说了这么多，你是不是有些跃跃欲试了，可能特别想知道到底该如何一键获取 SQL 整体信息和 SQL 的相关信息吧。很显然，这样效率一定能提升很多！

好吧，下面就跟随我一起看看吧。

2.3　从案例看快速 SQL 调优

2.3.1　获取数据库整体的运行情况

步骤 1（构造环境，对当前数据库进行各种操作）

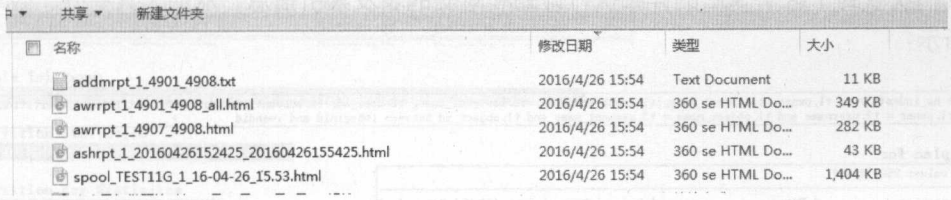
```
sqlplus "/ as sysdba" @d:\mkdb.sql
```

步骤 2（运行脚本，对当前数据库进行整体提取）

```
sqlplus "/ as sysdba" @spooldb.sql
```

步骤 3（输出对应日志文件，以供后续进行分析）

输出如下 5 个文件，其中 addm 是最近一小时文件，ash 是最近半小时文件，而 awr 文件是最近一小时和最近 7 天的两个文件，spool 打头的文件输出数据库所有的相关信息。有了这 5 个文件，基本上数据库的情况可以了解得比较清晰。具体文件如下图所示：



具体这 5 个文件我们就不一一打开给大家看了，否则篇幅显得过大，相关内容我会在线上给大家做详细的讲解。 其中 spool 打头的这个文件基本涵盖了你想获取的所有相关数据库信息。如：数据库版本、数据库参数、主机参数、异常的表和索引信息（表分区有无异常、并行度问题、索引是否失效、是否过大、是否为分区索引、索引类型、哪个列上有索引、索引过多、组合索引的组合列过多、全局临时表的异常信息收集情况，等等）、日志切换情况、序列情况、异常触发器、异常外键设置等。

如果要在书中展现，预计要有 20 多面，这里只告诉读者大致的内容，在线上会展现脚本，并进行视频演示，给读者看实际操作。

2.3.2　获取 SQL 的各种详细信息

步骤 1（构造环境，执行部分效率低下的 SQL）

```
sqlplus "/ as sysdba" @d:\mksql.sql
```

步骤 2（对当前的性能低下的 SQL 进行收集）

```
       405  bytes received via SQL*Net from client
         1  SQL*Net roundtrips to/from client
         6  sorts (memory)
         0  sorts (disk)
         0  rows processed

PL/SQL 过程已成功完成。

已用时间:  00: 00: 00.04

SQL_ID        ELAPSED      CPUTIME          EXECS elapsed/exec %DB time     MODULE                    SQLTXT
------------- ----------- ---------- ---------- ------------ ----------- ------------------------- ----------------------------------------------------------------
--------
5fy6m249h2kr5       16.852       6.864          1       16.852      18.179 sqlplus.exe               select /*+ no_index(t3) */
                                                                                                             t1.owner,


8u4w3j5y75sxb        8.660       5.694          3        2.887       9.342 sqlplus.exe               select t1.owner,
                                                                                                             t1.object_name,
                                                                                                             t1.


1uk5m5qbzj1vt        5.379       4.555          1        5.379       5.802 sqlplus.exe               BEGIN dbms_workload_repository.create_snapshot( EN
bunssq950nnhf        1.586       1.576          1        1.586       1.710                           insert into wrh$_sga_target_advice     (snap_id, dbi
ga89aq3333ks8        1.373       1.342          1        1.373       1.481 sqlplus.exe               insert into hj.spoolsql_dba_objects select * from
dayq182sk41ks        1.343       1.357          1        1.343       1.448                           insert into wrh$_memory_target_advice     (snap_id,
bm2pwrpcr8ru6        1.341       1.342          1        1.341       1.447                           select sga_size s,          sga_size_factor * 100
6ajkhukk78nsr        0.538       0.406          1        0.538       0.580                           begin prvt_hdm.auto_execute(  :dbid, :inst_num , :e
7ugmvmy8uvb9s        0.180       0.000          1        0.180       0.194                           insert into wrh$_tempstatxs    (snap_id, dbid, inst

已选择9行。

已用时间:  00: 00: 00.28

USERNAME        EVENT                          TIME(SECOND) SQL_ID        SQL_TXT
--------------- ------------------------------ ------------ ------------- ----------------------------------------
SYS             Cpu + Wait For Cpu                       33 1ggmjzf1dkk1m BEGIN dbms_stats.gather_table_stats(ownname => 'HJ
SYS             Cpu + Wait For Cpu                       19 6k4u5jyaj4sgk INSERT /*+ append */ INTO HJ.SPOOLSQL_DBA_SEGMENTS
SYS             db file scattered read                   14 6k4u5jyaj4sgk INSERT /*+ append */ INTO HJ.SPOOLSQL_DBA_SEGMENTS
SYS             direct path read                          9 5fy6m249h2kr5 select /*+ no_index(t3) */     t1.owner,
```

步骤 3（对当前的性能低下的 SQL 进行收集）

我们获取到了这个 SQL 的详细执行计划和对应的表及索引信息，从而大大提升了效率，如下图所示：

sql_text

```
select /*+ no_index(t3) */ t1.owner, t1.object_name, t1.subobject_name, t1.last_ddl_time, t2.user_id, t2.account_status, t3.segment_name, t3.partition_name, t3.by
t3 where t1.owner = t2.username and t1.object_name = t3.segment_name and t1.object_id between :vbeginid and :vendid
```

explain plan for

```
Plan hash value: 2501740819

-----------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |     1 |   190 | 39899   (1)| 00:07:59 |
|*  1 |  FILTER               |                     |       |       |            |          |
|*  2 |   HASH JOIN           |                     |     1 |   190 | 39899   (1)| 00:07:59 |
|*  3 |    HASH JOIN          |                     |     1 |   148 |     5  (20)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL | SPOOLSQL_DBA_OBJECTS|     1 |   122 |     2   (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL | SPOOLSQL_DBA_USERS  |    42 |  1092 |     2   (0)| 00:00:01 |
|   6 |    TABLE ACCESS FULL  | SPOOLSQL_DBA_SEGMENTS|  12M|   489M| 39838   (1)| 00:07:59 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

1 - filter(TO_NUMBER(:VBEGINID)<=TO_NUMBER(:VENDID))
2 - access("T1"."OBJECT_NAME"="T3"."SEGMENT_NAME")
3 - access("T1"."OWNER"="T2"."USERNAME")
4 - filter("T1"."OBJECT_ID">=TO_NUMBER(:VBEGINID) AND
"T1"."OBJECT_ID"<=TO_NUMBER(:VENDID))
```

Execution Plan From V$SQLPLAN

SQL:5fy6m249h2kr5, PLAN_HASH_VALUE:2501740819

ID	DEP	OPERITION	NAME	ROWS	BYTES	COST(%CPU)	IO_COST	TIME
0	0	SELECT STATEMENT				39899		00:00:00
*1	1	FILTER						00:00:00
*2	2	HASH JOIN		1	190	39899(1)	39587	00:07:59
*3	3	HASH JOIN		1	148	5(20)	4	00:00:01
*4	4	TABLE ACCESS FULL	[TABLE (TEMP)]HJ.SPOOLSQL_DBA_OBJECTS	1	122	2(0)	2	00:00:01
5	4	TABLE ACCESS FULL	[TABLE]HJ.SPOOLSQL_DBA_USERS	42	1092	2(0)	2	00:00:01
6	3	TABLE ACCESS FULL	[TABLE]HJ.SPOOLSQL_DBA_SEGMENTS	12210432	512838144	39838(1)	39583	00:07:59

```
Predicate Information (identified by operation id):
---------------------------------------------------
*1 => [filter]:VBEGINID<=:VENDID
*2 => [access]"T1"."OBJECT_NAME"="T3"."SEGMENT_NAME"
*3 => [access]"T1"."OWNER"="T2"."USERNAME"
*4 => [filter]("T1"."OBJECT_ID">=:VBEGINID AND "T1"."OBJECT_ID"<=:VENDID)
```

SQL STATISTICS- V$SQLPLAN

ID	HASH	DB_TIME	MEM	VERS	EXES	DISKS	GETS	ROWS	CPU	ELAPSED	CLUSTER_WAIT	CONCURRENCE_WAIT
5fy6m249h2kr5	2501740819	1010394	21532	1	1	227849	269764	608000	6864041	16851677	0	0

Execution Plan From AWR

SQL:5fy6m249h2kr5, PLAN_HASH_VALUE:2501740819

ID	DEP	OPERITION	NAME	ROWS	BYTES	COST(%CPU)	IO_COST	TIME
0	0	SELECT STATEMENT				39899		00:00:00
1	1	FILTER						00:00:00
2	2	HASH JOIN		1	190	39899(1)	39587	00:07:59
3	3	HASH JOIN		1	148	5(20)	4	00:00:01
4	4	TABLE ACCESS FULL	[TABLE (TEMP)]HJ.SPOOLSQL_DBA_OBJECTS	1	122	2(0)	2	00:00:01
5	4	TABLE ACCESS FULL	[TABLE]HJ.SPOOLSQL_DBA_USERS	42	1092	2(0)	2	00:00:01
6	3	TABLE ACCESS FULL	[TABLE]HJ.SPOOLSQL_DBA_SEGMENTS	12210432	512838144	39838(1)	39583	00:07:59

Table Segment Size

owner	segment_name	MB
HJ	SPOOLSQL_DBA_SEGMENTS	1792.00
HJ	SPOOLSQL_DBA_USERS	0.06

Table Statistics

OWNER	table_name	num_rows	blocks	degree	last_analyzed	temporary	partitioned	pct_free	tablespace_name
HJ	SPOOLSQL_DBA_OBJECTS	0	0	1	26-4月 -16	Y	NO	10	
HJ	SPOOLSQL_DBA_SEGMENTS	12210432	228557	1	26-4月 -16	N	NO	10	USERS
HJ	SPOOLSQL_DBA_USERS	42	4	1	26-4月 -16	N	NO	10	USERS

Table Column Statistics

OWNER	table_name	column_name	data_type	nullable	last_analyzed	avg_col_len
HJ	SPOOLSQL_DBA_OBJECTS	CREATED	DATE	Y	04/26/16 15:48:33	0
HJ	SPOOLSQL_DBA_OBJECTS	DATA_OBJECT_ID	NUMBER	Y	04/26/16 15:48:33	0

Table Triggers

table_owner	table_name	base_object_type	tiggger_owner	trigger_name	trigger_type	triggering_event

Partition Statistics

OWNER	table_name	partitioning_type	partition_count

Partition Key Statistics

owner	name	object_type	column_name

Partition Range Statistics

owner	table_name	partition_name	high_value	tablespace_name

Index Segments

OWNER	table_name	segment_name	MB
HJ	SPOOLSQL_DBA_SEGMENTS	IDX_SPOOLSQL_DBA_SEGMENTS_NAME	448.00

Index Statistics

OWNER	table_name	name	rows	type	status	factor	blevel	distinct	leaf	unique	degree	analyzed
HJ	SPOOLSQL_DBA_SEGMENTS	IDX_SPOOLSQL_DBA_SEGMENTS_NAME	12687546	NORMAL	UNUSABLE	3598578	3	12539	57994	NONUNIQUE	1	04/26/16 15:49:15

Index columns

OWNER	table_name	table_name	column_name	column_position	DESCEND
HJ	SPOOLSQL_DBA_SEGMENTS	IDX_SPOOLSQL_DBA_SEGMENTS_NAME	SEGMENT_NAME	1	ASC

Partition Indexes Summary

OWNER	table_name	table_name	partitioning_type	partition_count

Partition Indexes Details

OWNER	index_name	partition_name	status	blevel	leaf_blocks	tablespace_name

大家不要小看了这个一键获取 SQL 相关详细信息的威力。当你认为一条 SQL 有问题时，只要获取到这些信息，根本就不需要进行任何与现场人员的交互动作，因为你要的东西都已经被收集在一起了。让我们一起看看下面一个技术人员解决问题的一个思维片段。

这语句该如何优化，让我先瞧瞧这语句的执行计划是啥?

哦，原来执行计划就在这里，太详细了!

奇怪，这里的执行计划不对啊，应该是要用索引啊，怎么会没有，难道是统计信息不准确，还是说没建索引，还是说索引失效了？

咦，相关统计信息的收集是正常的，奇怪。我再看看，这列有索引吗？咦，是有，奇怪了。哦，索引失效了，原来如此。看来我重建一下索引问题应该可以解决……

看完这段话有何感触，如果没有高效的一键收集手段，你的所有信息都要在数据库中查询而获取，如果需要让他人给你信息，那就更麻烦了，小王 1 小时后才得出如何优化的事情，正是因为无法直接获取全部有效信息，不断交互导致的。

2.4 本章总结延伸与习题

2.4.1 总结延伸

场景再现

我们来假想一下学过这两章内容和拥有工具脚本的小王，是什么样的场景吧。

场景 1：

一键获取数据库整体信息，发现整体主机资源不足，并定位到耗尽资源的外部应用程序，然后协调相关人员，将外部应用移走或者优化，问题迅速准确地得到解决。

场景 2：

一键获取数据库整体信息，根据在第 1 章中学到的相关知识，发现是某些 SQL 需要优化，然后根据一键所得的 SQL 详细信息，判定 SQL 问题在哪里，问题迅速准确地得到解决。

场景 3：

一键获取数据库整体信息，也获取到 SQL 相关详细信息，不过不知该如何下手优化，求助他人。由于提交的信息详细，无须交互迅速得到对方的建议帮助，问题迅速准确地得到解决。

脑洞大开

学到这里，大家已经明白笔者的用意了，后续请读者下载一下脚本，体验一下高效的过程，认真完成下列作业。其实完善是没有尽头的，这只是一个大概，更细节的实现会一直在公众号上更新，比如一键获取数据库整体信息中我就没有提到数据库告警日志、监听日志、主机和数据库是否需要打哪些补丁。比如获取的 AWR 和 ASH 等信息里，能否直接对这些报告的 SQL 进行超链接，进入到详细 SQL 信息中……有没有脑洞大开的感觉？

这两章如果展开讲，可以单独成书。这两章的知识落地到软件中，就是一个非常好的数据库高效分析诊断工具。是不是再次脑洞大开？

嗯，笔者会无偿地将自己的经验分享给大家，能帮到大家也是笔者最大的快乐！

2.4.2　习题训练

习题 1：说说缩短调优时间的大致思路。

习题 2：请下载笔者提供的脚本，完成数据库问题构造与整体信息的一键获取，并截图记录。

习题 3：请下载笔者提供的脚本，完成慢 SQL 的构造与 SQL 执行计划和关联信息的一键获取，并截图记录。

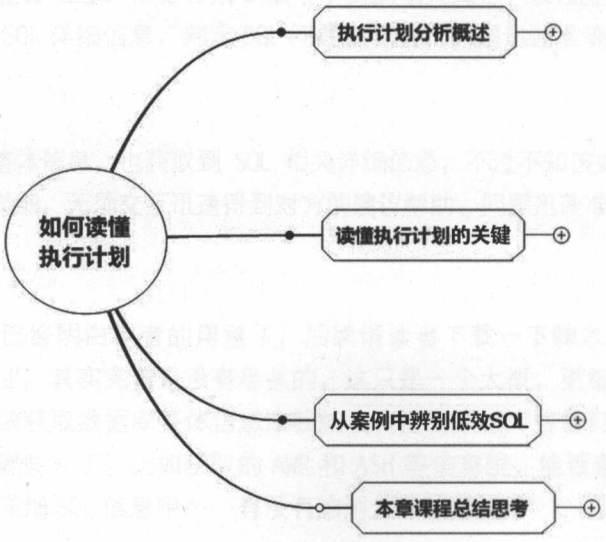
习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

第 3 章　循规蹈矩——如何读懂 SQL 执行计划

很少有 DBA 能完全读懂

前两章给大家介绍了发现问题后的整体解决思路，接下来进入 SQL 优化的局部性思路，这一章我们将学习 SQL 优化的重要知识：执行计划。

执行计划是什么，如何读懂执行计划，读懂执行计划对我们又有什么用？面对这些疑问，我们从执行计划分析概述、读懂执行计划的关键、从案例中辨别低效 SQL、总结思考这四部分入手开始本章的学习分享，如下图所示：



3.1　执行计划分析概述

3.1.1　SQL 执行计划是什么

1. 什么是 SQL 执行计划

执行计划是什么对我们来说其实还是比较好理解的，由于 SQL 语言是一种傻瓜式语言，一个条件就是一个需求，比如 select * from t1 , t2 where t1.id= t2.id and id=6 这样的语句，开发人员实际上只关心能否通过访问 t1、t2 两个表得到两个表相关联的数据，他们并没有指定该 SQL 如何执行，也就是说他们不关心该 SQL 是先访问 t1 表再访问 t2 表呢，还是先访问 t2 表再访问 t1 表。

对 SQL 来说，这两种访问方式就是两个不同的执行计划，而且必须做出选择，一次只能有一种访问路径。那到底选择哪一种呢？答案很简单，哪种执行开销更低，就意味着性能更好，速度更快，我们就选哪一种。这种过程就叫作 Oracle 的解析过程，一般都在 1s 内即可完成。然后数据库把这个更好的执行计划保存下来放到 SGA 的 Shared Pool 里，后续如果再执行同样的 SQL 只需要直接在 Shared Pool 里去获取就可以了，不需要再去分析了。

哦，这就是 SQL 的执行计划。

2. SQL 执行计划选定依据

等等，刚才好像说得有点偏简单了，数据库根据执行计划开销低不低来选择。可是问题来了，如何知道哪种最低呢？

有人说，让 Oracle 分别根据不同的执行计划执行一下，比较性能如何，不就成了，这听起来似乎有点道理，是先访问 t1 还是先访问 t2 也就是执行 2 次就知道谁代价更低了。

不过大家有没有注意到 and id=6 的条件，假如 id 列上有索引，Oracle 就会面临两个选择，一个走 id 列的索引访问方式，另一个不走索引，走全表扫描方式，是不是感觉又复杂了。假设将语句从 from t1 , t2 改成 from t1 , t2, t3 或者是 from t1 , t2, t3, t4，甚至是更多的表呢？请你现在用学过的数学排列组合的方法来判断 Oracle 会有多少个执行计划，是不是多得有点头晕了，难道你真忍心让该 SQL 去尝试执行每个执行计划吗？

当然，要是出现某种执行计划极其低效的情况，该 SQL 可能一直运行着，根本就停不下来，你也别想看到结果。

因此 Oracle 的执行计划的选择是有套路的。这里有一个重要的关键字：统计信息。有了这个统计信息，Oracle 就可以高效快速地完成 SQL 的解析过程（判断出代价更低的执行计划），还记得前面说过的话吗？一般解析都在 1s 内完成。

3.1.2 统计信息用来做什么

1. 统计信息又是什么东西

统计信息是什么东西？别着急，我们回到之前的例子 from t1 , t2 语句。从原理上来说，先访问小表，其成为驱动表，性能更高，因此如果我们知道 t1 和 t2 表谁更小，问题就解决了，不需要根据这两个不同的写法分别执行了。

有人说，此时 count(*)分别统计两个表就 OK 了，好简单啊。

真是如此吗？假如表的记录很大，查询不是很慢吗？你如何保障快速完成 SQL 解析从而选择代价低的执行计划呢？

所以这个大表和小表，是数据库直接告诉这条 SQL 的。统计信息就是这么来的。

数据库会对库里的每张表都做一个统计，记录每张表的记录数，比如可以通过如下语句来分析 t1 表和 t2 表的表及索引的统计信息：

```
---表的相关统计信息
SQL>  select  t.TABLE_NAME,t.NUM_ROWS,t.BLOCKS,t.LAST_ANALYZED  from  user_tables  t
where table_name in ('T1','T2');
TABLE_NAME                     NUM_ROWS     BLOCKS LAST_ANALYZED
------------------------------------------------------------------
T1                                826          14     05-3 月 -16
T2                             222256        3556    29-2 月 -16
---索引的相关统计信息
SQL>  select  table_name,index_name,t.blevel,t.num_rows,t.leaf_blocks,t.last_analyzed
from user_indexes t  where table_name in ('TEST','TEST1');

TABLE_NAME     INDEX_NAME                BLEVEL   NUM_ROWS LEAF_BLOCKS LAST_ANALYZED
-------------- ------------------------- ---------- ---------- ----------- --------------
TEST           IDX_TEST                       1        826           2 05-3 月 -16
TEST1          IDX_OWNER                      2     222256         644 29-2 月 -16
TEST1          IDX_OBJECT_NAME                2     222256        1191 29-2 月 -16
TEST1          IDX_DATA_OBJ_ID                1      97470         264 29-2 月 -16
TEST1          IDX_CREATED                    2     222256         787 29-2 月 -16
TEST1          IDX_LAST_DDL_TIME              2     222256         772 29-2 月 -16
TEST1          IDX_STATUS                     2     222256         634 29-2 月 -16
```

脚本 3-1　查看表和索引的统计信息

可以看出，t1 表比 t2 表小很多，这时 Oracle 就会选择让该 SQL 的执行计划先访问 t1 表，再访问 t2 表。不过问题来了，这 Oracle 如何知道这个数据呢？难道也是 count(*)一把吗？

嗯，这里又有学问了，其实 Oracle 做这事是很有套路的：

step 1 Oracle 会在一个固定的时间将库里的表和索引的相关统计信息进行收集（默认选择周一到周五晚上 10 点和周六日早上 6 点），用户可以自己调整收集时间，主要是为了避开高峰期。这么一来，等于用空间换时间，执行计划通过了解数据字典中的这些

num_rows 和 blocks 即可知道表的大小。

step 2 大家可能注意到 LAST_ANALYZED 字段的取值，似乎时间有点早，不像每天都收集的样子，原来 Oracle 可以专门对表的记录变化量进行管理，当某表一天记录变化量没有超过指定的阈值时，Oracle 就不会对该表进行统计信息收集，所以很多时候不少表被第一次收集统计信息后，由于一直很少更新，故很少再有针对该表收集信息的动作。

step 3 收集表和索引信息的动作非常灵活，比如可以只针对分区表的某个分区进行收集，可以在闲时使用并行机制来收集表和索引的统计信息，这可以极大提升性能，等等。

如此一来，Oracle 收集统计信息的本领真的强大了很多，也怪不得 Oracle 可以轻松地完成执行计划的解析工作了，原来 Oracle 有一套强大的处理机制来保障高效获取执行计划。

3.1.3　数据库统计信息的收集

关于收集统计信息的方法比较多，可以是针对整个数据库的收集，可以是针对整个 schema 的收集，也可以是针对具体某个表和索引的收集。从操作实用性来看，主要的日常操作都是手动收集某表或者某索引的统计信息，方法如下：

```
--收集表统计信息
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'TEST',estimate_percent =>
10,method_opt=> 'for all indexed columns') ;
--收集索引统计信息
exec dbms_stats.gather_index_stats(ownname => 'LJB',indname => 'IDX_OBJECT_ID',estimate_
percent => '10',degree => '4') ;
--收集表和索引统计信息
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'TEST',estimate_percent =>
10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;
```

脚本 3-2　收集表和索引的统计信息

如果是针对分区表，则可以指定只收集某分区，具体如下：

```
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'RANGE_PART_TAB',partname
=>'p_201312', estimate_percent => 10,method_opt=> 'for all indexed columns',cascade=>
TRUE) ;
```

脚本 3-3　针对某分区进行统计信息收集

可以看出，收集统计信息的方法其实非常灵活，具体细节可以多研究 dbms_stats.gather_table_stats 的其他参数。从实际案例来看，本节展现的细节已经够用了。

3.1.4　数据库的动态采样

善于动脑的同学可能会发现一个问题，比如 Oracle 是每天晚上 10 点收集统计信息，那此时如果是早上 9 点新建了一张表，Oracle 该如何知道这个表的记录大小呢？

确实，这是一个问题，Oracle 如何做呢，请看下面我们怎么做。

构造环境，新建一张 T_SAMPLE 表，发现统计信息果然没有被收集，对应的 NUM_ROWS BLOCKS 和 LAST_ANALYZED 都是空的。

```
set autotrace off
set linesize 1000
drop table t_sample purge;
create table t_sample as select * from dba_objects;
create index idx_t_sample_objid on t_sample(object_id);
 select num_rows, blocks, last_analyzed
  from user_tables
 where table_name = 'T_SAMPLE';

 NUM_ROWS    BLOCKS    LAST_ANALYZED
----------------------------------
```

脚本 3-4　新建表统计信息收集情况

那咋办？我们用 set autotrace on 的方式来跟踪 SQL 的执行计划，如下：

```
set autotrace traceonly
set linesize 1000

select  * from t_sample where object_id=20;
执行计划
----------------------------------------------------------
Plan hash value: 1453182238
-------------------------------------------------------------------------------------------
| Id |Operation                    | Name              |Rows  |Bytes |Cost (%CPU)|Time     |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |                   |    1 |  207 |    2   (0)|00:00:01 |
|   1| TABLE ACCESS BY INDEX ROWID| T_SAMPLE          |    1 |  207 |    2   (0)|00:00:01 |
|*  2|  INDEX RANGE SCAN           | IDX_T_SAMPLE_OBJID |    1 |      |    1   (0)|00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("OBJECT_ID"=20)
Note
-----
   - dynamic sampling used for this statement (level=2)
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
       1393  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
```

```
          0  sorts (disk)
          1  rows processed
```

脚本 3-5　未收集统计信息会进行动态采样

请注意看这个 dynamic sampling used for this statement (level=2)，这是啥呢？这就是动态采样，当一张表是新建表时，Oracle 只好动态地收集这个表的相关信息。然后等到晚上 10 点，再将其收集到数据字典中。

我们可以继续做，比如手工收集统计信息：

```
SQL>  exec  dbms_stats.gather_table_stats(ownname  =>  'LJB',tabname  =>  'T_SAMPLE',
estimate_percent => 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;

PL/SQL 过程已成功完成。
SQL> set autotrace off
SQL>  select num_rows, blocks, last_analyzed
  2      from user_tables
  3     where table_name = 'T_SAMPLE';
  NUM_ROWS     BLOCKS LAST_ANALYZED
---------- ---------- --------------
    112140       1753 08-4 月 -16
```

此时我们再跟踪 SQL 的执行计划，情况就变化了：

```
set autotrace traceonly
select  * from t_sample where object_id=20;
执行计划
----------------------------------------------------------------------------------------------
Plan hash value: 1453182238
----------------------------------------------------------------------------------------------
| Id |Operation                    | Name              |Rows  | Bytes |Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |                   |    1 |    97 |    2   (0)| 00:00:01|
|   1| TABLE ACCESS BY INDEX ROWID| T_SAMPLE          |    1 |    97 |    2   (0)| 00:00:01|
|*  2|  INDEX RANGE SCAN           | IDX_T_SAMPLE_OBJID|    1 |       |    1   (0)| 00:00:01|
----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("OBJECT_ID"=20)
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
       1393  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

```
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 3-6　收集统计信息后动态采样消失

有什么新发现？没错，dynamic sampling 关键字不见了！

3.1.5　获取执行计划的方法（6 种武器）

前面大家看到 set autotrace on 关键字，这是获取执行计划的方法。实际上，在 Oracle 里获取执行计划的方法有 6 种。这听起来是不是有些吓人？实际上这些方法各有侧重点，下面我们一一细看这些方法。

1. 六脉神剑

（1）explain plan for 获取

```
set linesize 1000
set pagesize 2000
explain plan for
SELECT  *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n in(18,19);
select * from table(dbms_xplan.display());

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 3532430033
------------------------------------------------------------------------------------------
| Id  |Operation                      | Name     |Rows  | Bytes |Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT               |          |    2 |  8138 |    6   (0)| 00:00:01 |
|   1 | NESTED LOOPS                  |          |      |       |           |          |
|   2 |  NESTED LOOPS                 |          |    2 |  8138 |    6   (0)| 00:00:01 |
|   3 |   INLIST ITERATOR             |          |      |       |           |          |
|   4 |    TABLE ACCESS BY INDEX ROWID| T1       |    2 |  4056 |    2   (0)| 00:00:01 |
|*  5 |     INDEX RANGE SCAN          | T1_N     |    1 |       |    1   (0)| 00:00:01 |
|*  6 |   INDEX RANGE SCAN            | T2_T1_ID |    1 |       |    1   (0)| 00:00:01 |
|   7 |  TABLE ACCESS BY INDEX ROWID  | T2       |    1 |  2041 |    2   (0)| 00:00:01 |
------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access("T1"."N"=18 OR "T1"."N"=19)
   6 - access("T1"."ID"="T2"."T1_ID")
Note
-----
   - dynamic sampling used for this statement (level=2)
```

```
已选择 24 行。
/*
优点:   1.无须真正执行，快捷方便
缺陷:   1.没有输出运行时的相关统计信息（产生多少逻辑读，多少次递归调用，多少次物理读的情况）；
        2.无法判断处理了多少行；
        3.无法判断表被访问了多少次。
确实啊，这毕竟都没有真正执行又如何得知真实运行产生的统计信息。
*/
```

脚本 3-7　explain plan for 获取执行计划

（2）set autotrace on

```
set autotrace on
SELECT  *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n in(18,19);

执行计划
----------------------------------------------------------
Plan hash value: 3532430033
-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |           |     2 |  8138 |     6   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |           |       |       |            |          |
|   2 |   NESTED LOOPS                |           |     2 |  8138 |     6   (0)| 00:00:01 |
|   3 |    INLIST ITERATOR            |           |       |       |            |          |
|   4 |     TABLE ACCESS BY INDEX ROWID| T1        |     2 |  4056 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN         | T1_N      |     1 |       |     1   (0)| 00:00:01 |
|*  6 |    INDEX RANGE SCAN           | T2_T1_ID  |     1 |       |     1   (0)| 00:00:01 |
|   7 |   TABLE ACCESS BY INDEX ROWID | T2        |     1 |  2041 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access("T1"."N"=18 OR "T1"."N"=19)
   6 - access("T1"."ID"="T2"."T1_ID")
Note
-----
   - dynamic sampling used for this statement (level=2)
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         12  consistent gets
          0  physical reads
          0  redo size
       1032  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

```
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed

/*
--优点：1.可以输出运行时的相关统计信息（产生多少逻辑读，多少次递归调用，多少次物理读的情况）；
       2.虽然必须要等语句执行完毕后才可以输出执行计划，但是可以有 traceonly 开关来控制返回结果不打屏输出。

--缺陷：1.必须要等到语句真正执行完毕后，才可以出结果；
       2.无法看到表被访问了多少次。

*/
```

脚本 3-8　set autotrace on 获取执行计划

（3）statistics_level=all

```
/*
  步骤 1：alter session set statistics_level=all ;
  步骤 2：在此处执行你的 SQL
  步骤 3：select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

 另注：

  1. 如果你用 /*+ gather_plan_statistics */的方法，可以省略步骤 1，直接执行步骤 2,3。
  2. 关键字解读（其中 0Mem、1Mem 和 User-Mem 在后续的课程中会陆续见到）：
    Starts 为该 SQL 执行的次数。
    E-Rows 为执行计划预计的行数。
    A-Rows 为实际返回的行数。A-Rows 和 E-Rows 做比较，就可以确定哪一步执行计划出了问题。
    A-Time 为每一步实际执行的时间（HH：MM：SS.FF），根据这一行可以知道该 SQL 耗时在哪个地方。
    Buffers 为每一步实际执行的逻辑读或一致性读。
    Reads 为物理读。
    0Mem:当前操作完成所有内存工作区(Work Aera)操作所总共使用私有内存(PGA)中工作区的大小，
         这个数据是由优化器统计数据以及前一次执行的性能数据估算得出的。
    1Mem:当工作区大小无法满足操作所需的大小时，需要将部分数据写入临时磁盘空间中(如果仅需要写入一次就可以
         完成操作，就称一次通过，One-Pass;否则为多次通过，Multi-Pass)。该列数据为语句最后一次执行中，
         单次写磁盘所需要的内存大小，这个是由优化器统计数据以及前一次执行的性能数据估算得出的。
    User-Mem:语句最后一次执行中，当前操作所使用的内存工作区大小，括号里面为(发生磁盘交换的次数,1 次即为
             One-Pass，大于 1 次则为 Multi-Pass,如果没有使用磁盘，则显示 OPTIMAL)
    0Mem、1Mem 为执行所需的内存评估值，0Mem 为最优执行模式所需内存的评估值，1Mem 为 one-pass 模式所需内
存的评估值。
    0/1/M 为最优/one-pass/multipass 执行的次数。Used-Mem 为消耗的内存
*/
set autotrace off
alter session set statistics_level=all ;
SELECT  *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n in(18,19);
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
SQL_ID  1a914ws3ggfsn, child number 0
-------------------------------------
SELECT  * FROM t1, t2 WHERE t1.id = t2.t1_id AND t1.n in(18,19)
Plan hash value: 3532430033
------------------------------------------------------------------------------------------------------
| Id | Operation                     | Name    |Starts |E-Rows | A-Rows |   A-Time   |Buffers |
------------------------------------------------------------------------------------------------------
|   0| SELECT STATEMENT              |         |     1 |       |      2 |00:00:00.01 |     12 |
|   1|  NESTED LOOPS                 |         |     1 |       |      2 |00:00:00.01 |     12 |
|   2|   NESTED LOOPS                |         |     1 |     2 |      2 |00:00:00.01 |     10 |
|   3|    INLIST ITERATOR            |         |     1 |       |      2 |00:00:00.01 |      5 |
|   4|     TABLE ACCESS BY INDEX ROWID| T1      |     2 |     2 |      2 |00:00:00.01 |      5 |
|*  5|      INDEX RANGE SCAN         | T1_N    |     2 |     1 |      2 |00:00:00.01 |      3 |
|*  6|    INDEX RANGE SCAN           | T2_T1_ID|     2 |     1 |      2 |00:00:00.01 |      5 |
|   7|   TABLE ACCESS BY INDEX ROWID | T2      |     2 |     1 |      2 |00:00:00.01 |      2 |
------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access(("T1"."N"=18 OR "T1"."N"=19))
   6 - access("T1"."ID"="T2"."T1_ID")
Note
-----
   - dynamic sampling used for this statement (level=2)
已选择 29 行。
/*
--优点：1.可以清晰地从 STARTS 得出表被访问多少次；
        2.可以清晰地从 E-ROWS 和 A-ROWS 中得到预测的行数和真实的行数，从而可以准确判断 Oracle 评估是否准确；
        3.虽然没有专门的输出运行时的相关统计信息，但是执行计划中的 BUFFERS 就是真实的逻辑读的数值。

--缺陷：1.必须要等到语句真正执行完毕后，才可以出结果；
        2.无法控制记录打屏输出，不像 autotrace 有 traceonly 可以控制不将结果打屏输出；
        3.看不出递归调用的次数，看不出物理读的数值（不过逻辑读才是重点）。
*/
```

脚本 3-9　statistics_level=all 获取执行计划

（4）dbms_xplan.display_cursor 获取

```
/*
步骤 1：  select  * from table(dbms_xplan.display_cursor('&sq_id'));  （该方法是从共享池里得到）

注：
  1. 还有一个方法，select  * from table(dbms_xplan.display_awr('&sq_id'));（这是从 awr 性能视图里获取）
  2. 如果有多个执行计划，则可以用类似方法查出：
    select * from table(dbms_xplan.display_cursor('cyzznbykb509s',0));
    select * from table(dbms_xplan.display_cursor('cyzznbykb509s',1));
*/
```

```
select * from table(dbms_xplan.display_cursor('1a914ws3ggfsn'));
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
SQL_ID  1a914ws3ggfsn, child number 0
-------------------------------------
SELECT  * FROM t1, t2 WHERE t1.id = t2.t1_id AND t1.n in(18,19)
Plan hash value: 3532430033
------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |          |       |       |     6 (100)|          |
|   1 |  NESTED LOOPS                 |          |       |       |            |          |
|   2 |   NESTED LOOPS                |          |     2 |  8138 |     6   (0)| 00:00:01 |
|   3 |    INLIST ITERATOR            |          |       |       |            |          |
|   4 |     TABLE ACCESS BY INDEX ROWID| T1       |     2 |  4056 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN         | T1_N     |     1 |       |     1   (0)| 00:00:01 |
|*  6 |    INDEX RANGE SCAN           | T2_T1_ID |     1 |       |     1   (0)| 00:00:01 |
|   7 |   TABLE ACCESS BY INDEX ROWID | T2       |     1 |  2041 |     2   (0)| 00:00:01 |
------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access(("T1"."N"=18 OR "T1"."N"=19))
   6 - access("T1"."ID"="T2"."T1_ID")
Note
-----
   - dynamic sampling used for this statement (level=2)

/*
--优点：1.知道 sql_id 立即可得到执行计划，它和 explain plan for 一样无须执行；
       2.可以得到真实的执行计划。（停，等等，啥真实的，刚才这几个套路中，还有假的执行计划吗？）


--缺陷：1.没有输出运行时的相关统计信息（产生多少逻辑读，多少次递归调用，多少次物理读的情况）；
       2.无法判断处理了多少行；
       3.无法判断表被访问了多少次。

*/
```

脚本 3-10　dbms_xplan.display_cursor 获取执行计划

（5）事件 10046 trace 跟踪

```
/*
  步骤 1：alter session set events '10046 trace name context  forever,level 12'; （开启跟踪）
  步骤 2：执行你的语句
  步骤 3：alter session set events '10046 trace name context off';    （关闭跟踪）
  步骤 4：找到跟踪后产生的文件
  步骤 5：tkprof  trc 文件  目标文件  sys=no sort=prsela,exeela,fchela  （格式化命令）
*/
```

```
set autotace off
alter session set statistics_level=typical;
alter session set events '10046 trace name context  forever,level 12';

SELECT  *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n in(18,19);

alter session set events '10046 trace name context off';
select d.value
|| '/'
|| LOWER (RTRIM(i.INSTANCE, CHR(0)))
|| '_ora_'
|| p.spid
|| '.trc' trace_file_name
from (select p.spid
      from v$mystat m,v$session s, v$process p
      where  m.statistic#=1 and s.sid=m.sid and p.addr=s.paddr) p,
      (select t.INSTANCE
       FROM v$thread t,v$parameter v
       WHERE v.name='thread'
       AND(v.VALUE=0 OR t.thread#=to_number(v.value))) i,
       (select value
       from v$parameter
       where name='user_dump_dest') d;

exit

tkprof d:\oracle\diag\rdbms\test11g\test11g\trace/test11g_ora_2492.trc      d:\10046.txt   sys=no
sort=prsela,exeela,fchela

SELECT  *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n in(18,19)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0         12          0           2
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          0         12          0           2

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 94

Rows     Row Source Operation
-------  ---------------------------------------------------
      2  NESTED LOOPS  (cr=12 pr=0 pw=0 time=0 us)
```

```
      2   NESTED LOOPS  (cr=10 pr=0 pw=0 time=48 us cost=6 size=8138 card=2)
      2    INLIST ITERATOR  (cr=5 pr=0 pw=0 time=16 us)
      2     TABLE ACCESS BY INDEX ROWID T1 (cr=5 pr=0 pw=0 time=0 us cost=2 size=4056
card=2)
      2           INDEX  RANGE  SCAN  T1_N  (cr=3  pr=0  pw=0  time=0  us  cost=1  size=0
card=1)(object id 108621)
      2       INDEX  RANGE  SCAN  T2_T1_ID  (cr=5  pr=0  pw=0  time=0  us  cost=1  size=0
card=1)(object id 108622)
      2    TABLE ACCESS BY INDEX ROWID T2 (cr=2 pr=0 pw=0 time=0 us cost=2 size=2041
card=1)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                        2        0.00          0.00
  SQL*Net message from client                      2        1.31          1.31

/*
--优点：1.可以看出 SQL 语句对应的等待事件；
        2.如果 SQL 语句中有函数调用，函数中又有 SQL，将会被列出，无处遁形；
        3.可以方便地看出处理的行数，产生的物理逻辑读；
        4.可以方便地看出解析时间和执行时间；
        5.可以跟踪整个程序包。

--缺陷：1.步骤烦琐，比较麻烦；
        2.无法判断表被访问了多少次；
        3.执行计划中的谓词部分不能清晰地展现出来。
*/
```

脚本 3-11　10046 trace 获取执行计划

（6）awrsqrpt.sql

```
/*
  步骤 1：@?/rdbms/admin/awrsqrpt.sql
  步骤 2：选择你要的断点（begin snap 和 end snap）
  步骤 3：输入你的 sql_id
*/
```

脚本 3-12　awrsqrpt.sql 调用获取执行计划

2. 差异何在

- 如果某 SQL 执行很长时间才出结果或返回不了结果，这时就只能用方法 1。
- 跟踪某条 SQL 最简单的方法是方法 1，其次就是方法 2。
- 如果想观察某条 SQL 多个执行计划的情况，只能用方法 4 和方法 6。
- 如果 SQL 中含函数，函数中又套 SQL 等，即存在多层调用，想准确分析只能用方法 5。
- 要想确保看到真实的执行计划，不能用方法 1 和方法 2。
- 要想获取表被访问的次数，只能使用方法 3。

3. 总结说明

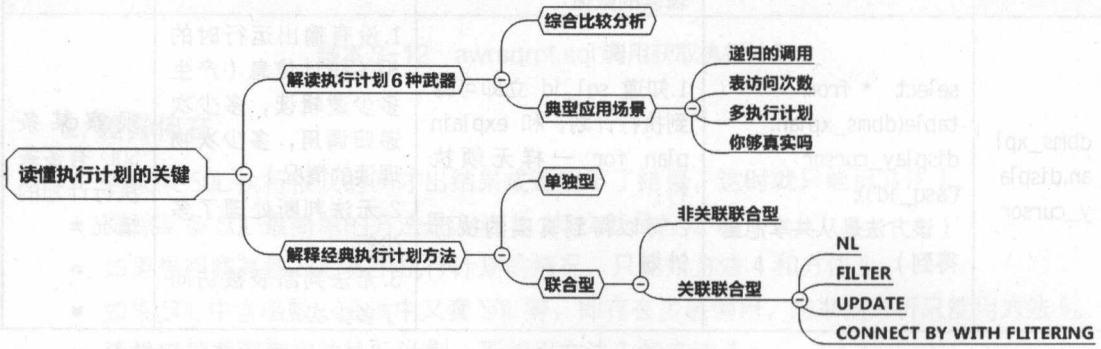
获取执行计划的方法				
方法	获取步骤	优点	缺点	应用场景
explain plan for	步骤 1：explain plan for/跟上你要执行的 SQL/ 步骤 2：select * from table(dbms_xplan.display());	无须真正执行，快捷方便	1.没有输出运行时的相关统计信息（产生多少逻辑读，多少次递归调用，多少次物理读的情况）； 2.无法判断处理了多少行； 3.无法判断表被访问了多少次	如果某 SQL 执行很长时间才出结果或返回不了结果
set autotrace on	步骤 1：set autotrace on 步骤 2：在此处执行你的 SQL	1.可以输出运行时的相关统计信息（产生多少逻辑读，多少次递归调用，多少次物理读的情况）； 2.虽然必须要等语句执行完毕后才可以输出执行计划，但是可以有 traceonly 开关来控制返回结果不打屏输出	1.必须要等到语句真正执行完毕后，才可以出结果； 2.无法看到表被访问了多少次	想粗略知道 recursive calls 递归调用次数，用这个方法，详细用 10046trace 方法
statistics_level=all	步骤 1：alter session set statistics_level=all； 步骤 2：在此处执行你的 SQL 步骤 3：select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));	1.可以清晰地从 STARTS 得出表被访问多少次； 2.可以清晰地从 E-ROWS 和 A-ROWS 中得到预测的行数和真实的行数，从而可以准确判断 Oracle 评估是否准确。 3.虽然没有专门的输出运行时的相关统计信息，但是执行计划中的 BUFFERS 就是真实的逻辑读的数值	1.必须要等到语句真正执行完毕后，才可以出结果； 2.无法控制输出记录展现与否，而 autotrace 有 traceonly 可以控制不将输出记录打屏； 3.看不出递归调用的次数，看不出物理读的数值	要想获取表被访问的次数，只能使用方法 3
dbms_xplan.display_cursor	select * from table(dbms_xplan.display_cursor('&sq_id')); （该方法是从共享池里得到）	1.知道 sql_id 立即可得到执行计划，和 explain plan for 一样无须执行； 2.可以得到真实的执行计划	1.没有输出运行时的相关统计信息（产生多少逻辑读，多少次递归调用，多少次物理读的情况）； 2.无法判断处理了多少行； 3.无法判断表被访问了多少次	观察某条 SQL 有多条执行计划的情况

续表

获取执行计划的方法				
方法	获取步骤	优点	缺点	应用场景
事件 10046 trace 跟踪	步骤 1：alter session set events '10046 trace name context forever,level 12'；（开启跟踪） 步骤 2：执行你的语句 步骤 3：alter session set events '10046 trace name context off'；（关闭跟踪） 步骤 4：找到跟踪后产生的文件 步骤 5：tkprof trc 文件目标文件	1.可以看出 SQL 语句对应的等待事件 2.如果 SQL 语句中有函数调用，SQL 中有 SQL，都将会被列出，无处遁形； 3.可以方便地看出处理的行数，产生的物理逻辑读； 4.可以方便地看出解析时间和执行时间； 5.可以跟踪整个程序包	1.步骤烦琐，比较麻烦； 2.无法判断表被访问了多少次； 3.执行计划中的谓词部分不能清晰地展现出来	如果 SQL 中含函数，函数中又套 SQL 等，即存在多层调用，想准确分析只能用该方法
awrsqrpt.sql	步骤 1：@?/rdbms/admin/awrsqrpt.sql 步骤 2：选择你要的断点（begin snap 和 end snap） 步骤 3：输入你的 sql_id	可以方便地看到多个执行计划	获取的过程比较麻烦	想观察某条 SQL 的多个执行计划用该方法

3.2 读懂执行计划的关键

前面讲了关于执行计划和统计信息的一些基础知识，其实真正要读懂执行计划，并不是一件容易的事，首先要善于利用获取执行计划的工具。下面将介绍如何使用这些经典工具，如下图所示：



其次要了解执行计划中 Oracle 是如何一步一步执行的，我们要做到像 Oracle 一样去思

考。这似乎很难，不过当你从最简单的单独型、联合型开始学习之后，你就会发现，其实这些都很容易。

3.2.1　解释经典执行计划方法

关于执行计划，最重要的一点是要读懂执行的顺序，只有这样，才可以像 Oracle 一样思考问题。而执行的顺序到底是什么呢？是从远到近，还是从上到下呢？这里我们先定义两种类型：1. 单独型；2.联合型。首先我们来看看单独型。

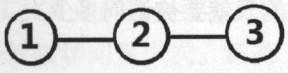
1. 单独型

请看下面 SQL 执行计划 Id=3 处，通过索引定位 JOB='CLERK'，然后观察 Id=2 处，通过 rowid 回到表中得到 sal 等其他列，然后根据 sal<3000 的条件再过滤部分数据。最后完成了 deptno 动作，请看 Id=1 处。

```
SELECT deptno, count(*)
FROM emp WHERE job = 'CLERK' AND sal < 3000
GROUP BY deptno
Plan hash value: 3067371962
---------------------------
| Id  | Operation                    | Name       | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |      1 |        |      2 |
|   1 |  HASH GROUP BY               |            |      1 |      1 |      2 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| EMP        |      1 |      3 |      3 |
|*  3 |    INDEX RANGE SCAN          | EMP_JOB_I  |      1 |      4 |      4 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("SAL"<3000)
   3 - access("JOB"='CLERK')
Note
-----
   - cardinality feedback used for this statement
已选择 26 行。
```

脚本 3-13　单独型执行计划的例子

我们把这种执行计划称之为单独型，有一种父子的关系，如下图所示。执行顺序为 Id=3，Id=2，Id=1，由远到近地执行。注意看，执行计划中 Id=1，Id=2，Id=3 有一定的偏移哦，这就是单独型的特征。



2. 联合型

联合型还分为非关联的联合型和关联的联合型，请看下列执行计划，Id=2，Id=3，Id=4 三处的语句互相独立，没有谁是谁的孩子，这时执行的顺序就是 Id=2，Id=3，Id=4 依次进行。

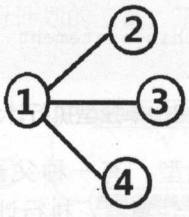
（1）联合型的非关联型

```
SELECT ename FROM emp
UNION ALL
SELECT dname FROM dept
UNION ALL
SELECT '%' FROM dual;
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-----------------
SQL_ID  781xq971h0y2p, child number 0
-------------------------------------
SELECT ename FROM emp UNION ALL SELECT dname FROM dept UNION ALL SELECT
'%' FROM dual
Plan hash value: 4181933179
---------------------------
| Id  | Operation           | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |        |     19 |00:00:00.01 |      16 |
|   1 |  UNION-ALL          |      |      1 |        |     19 |00:00:00.01 |      16 |
|   2 |   TABLE ACCESS FULL| EMP  |      1 |     14 |     14 |00:00:00.01 |       8 |
|   3 |   TABLE ACCESS FULL| DEPT |      1 |      4 |      4 |00:00:00.01 |       8 |
|   4 |   FAST DUAL         |      |      1 |      1 |      1 |00:00:00.01 |       0 |
-----------------------------------------------------------------------------------------
已选择 17 行。
```

脚本 3-14　联合型的非关联型

下图是我们根据 Id 描绘的图，这就是典型的联合型。注意看，执行计划中，Id=2, Id=3, Id=4 是对齐无偏移的，这就是联合型的特征。

这里顺序很显然是 Id=1，Id=2，Id=3，Id=4。请大家注意，Id=2，Id=3，Id=4 三处，互相之间毫无关系，这就是非关联的联合型。请注意看 Id=2 处的 A-rows 为 14，可是 Id=3 处的 Starts 依然为 1，表示只访问一次，和这个 14 的结果毫无关系。



大家应该可以猜到，联合型的关联型是啥样，那就是互相之间有关系，比如 Id=2 的语句涉及的记录返回多少条，Id=3 涉及的表就要被访问多少次，那就是互相之间有关联了。

（2）　联合型的关联型

1）　联合型的关联型（NL）

接下来的例子比较经典，请看 Id=2 和 Id=3 处，这里显然是联合型，不过我们再观察，Id=2 处的 A-Rows 为 10，Id=3 处的 Starts=10，说明 EMP 访问的结果集返回多少条，DEPT 表就

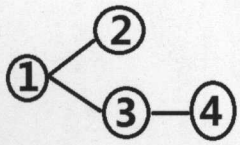
被访问多少次，这是有关联的，这就是联合型的关联型。请回头再看看联合型的非关联型，应该可以明白。

这里其实是联合型和单独型混合的执行计划，请看 Id=3 和 Id=4 处，这显然就是单独型，顺序是先 Id=4，再 Id=3。

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-----------------
SQL_ID  3ux4g340c933p, child number 0
-------------------------------------
SELECT /*+ ordered use_nl(dept) index(dept) */ * FROM emp, dept WHERE emp.deptno =
dept.deptno AND emp.comm IS NULL AND dept.dname != 'SALES'
--为了方便理解，先用 10g 的执行计划，先理解执行计划的类型很重要。
Plan hash value: 3487251775
------------------------------------------------------------------
| Id |Operation                    | Name    | Starts | E-Rows| A-Rows|   A-Time    | Buffers|
-------------------------------------------------------------------------------------------
|   1| NESTED LOOPS                |         |      1 |     10|      8|00:00:00.01 |      20|
|*  2|  TABLE ACCESS FULL          | EMP     |      1 |     10|     10|00:00:00.01 |       8|
|*  3|  TABLE ACCESS BY INDEX ROWID| DEPT    |     10 |      1|      8|00:00:00.01 |      12|
|*  4|   INDEX UNIQUE SCAN         | DEPT_PK|     10 |      1|     10|00:00:00.01 |       2|
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("EMP"."COMM" IS NULL)
   3 - filter("DEPT"."DNAME"<>'SALES')
   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
23 rows selected.
```

脚本 3-15　联合型的关联型

具体的示意图如下，顺序一目了然：Id=2，Id=4，Id=3，Id=1。其中 Id=2 返回的条数将会决定 Id=3 和 Id=4 执行的次数。



现在，无论多复杂的示意图，我们应该都不会害怕了，都可以轻易地画出执行顺序了。接下来我们对知识进行一个拓展，其实关联型的联合型不见得就一定如同 NESTED LOOPS 模式一样，驱动表返回多少条，被驱动表就被访问多少次。还有 FILTER 的模式也是关联型的联合型，具体就有差异，接下来请看下面的例子。

2）　联合型的关联型（FILTER）

我们认真观察会发现，Id=2 处的 A-rows 为 14，但是 Id=3 处的 Starts 却为 3，这是何

故呢?

```
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-----------------
SQL_ID  143p2rxtjxj0m, child number 0
-------------------------------------
SELECT * FROM emp WHERE NOT EXISTS (SELECT /*+ no_unnest */ 0
     FROM dept                WHERE dept.dname = 'SALES' AND
dept.deptno = emp.deptno) AND NOT EXISTS (SELECT /*+ no_unnest */ 0
         FROM bonus              WHERE bonus.ename = emp.ename)

Plan hash value: 2272441335
---------------------------
| Id |Operation                    | Name    |Starts |E-Rows | A-Rows |   A-Time   | Buffers|
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |         |     1 |       |      8 |00:00:00.01 |      14|
|*  1| FILTER                      |         |     1 |       |      8 |00:00:00.01 |      14|
|   2|  TABLE ACCESS FULL          | EMP     |     1 |    14 |     14 |00:00:00.01 |       8|
|*  3|  TABLE ACCESS BY INDEX ROWID| DEPT    |     3 |     1 |      1 |00:00:00.01 |       6|
|*  4|   INDEX UNIQUE SCAN         | DEPT_PK |     3 |     1 |      3 |00:00:00.01 |       3|
|*  5|  TABLE ACCESS FULL          | BONUS   |     8 |     1 |      0 |00:00:00.01 |       0|
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(( IS NULL AND  IS NULL))
   3 - filter("DEPT"."DNAME"='SALES')
   4 - access("DEPT"."DEPTNO"=:B1)
   5 - filter("BONUS"."ENAME"=:B1)
已选择 28 行。
```

脚本 3-16　联合型的关联型（FILTER）

原因分析：为什么执行计划中 Id=3 的地方 STARTS 为 3 次，因为虽然有 8 条记录，但是不重复的只有 3 个（ACCOUNTING、RESEARCH、SALES）。

```
SELECT dname, count(*)
FROM emp, dept
WHERE emp.deptno = dept.deptno
GROUP BY dname;

DNAME          COUNT(*)
----------------------
ACCOUNTING             3
RESEARCH               5
SALES                  6
```

接下来我们看看为什么执行计划中 ID=5 的地方 STARTS 为 8 次，这是因为返回 8 条记录。

```
SELECT ename
FROM emp
WHERE NOT EXISTS (SELECT /*+ no_unnest */ 0
```

```
                  FROM dept
                 WHERE dept.dname = 'SALES' AND dept.deptno = emp.deptno);

ENAME
------
SMITH
JONES
CLARK
SCOTT
KING
ADAMS
FORD
MILLER

已选择 8 行。
```

原来如此，FILTER 其实对比 NESTED LOOPS 是一种优化，驱动表返回多少条不重复记录，被驱动表被访问多少次，请注意“不重复”三个字。接下来还有 UPDATE 的执行计划，由于其和 FILTER 类似，这里就不做说明了。

3）　联合型的关联型（UPDATE）

```
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
------------------
SQL_ID  aj9bzs3ptc6wn, child number 0
-------------------------------------
UPDATE emp e1 SET sal = (SELECT avg(sal) FROM emp e2 WHERE e2.deptno =
e1.deptno),  comm = (SELECT avg(comm) FROM emp e3)

Plan hash value: 1690508028
---------------------------
| Id | Operation           | Name |Starts |E-Rows |A-Rows |   A-Time   |Buffers|
-------------------------------------------------------------------------------
|   0| UPDATE STATEMENT    |      |     1 |       |     0 |00:00:00.01 |     65|
|   1|  UPDATE             | EMP  |     1 |       |     0 |00:00:00.01 |     65|
|   2|   TABLE ACCESS FULL | EMP  |     1 |    14 |    14 |00:00:00.01 |      7|
|   3|   SORT AGGREGATE    |      |     3 |     1 |     3 |00:00:00.01 |     21|
|*  4|    TABLE ACCESS FULL| EMP  |     3 |     5 |    14 |00:00:00.01 |     21|
|   5|   SORT AGGREGATE    |      |     1 |     1 |     1 |00:00:00.01 |      7|
|   6|    TABLE ACCESS FULL| EMP  |     1 |    14 |    14 |00:00:00.01 |      7|
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter("E2"."DEPTNO"=:B1)
已选择 24 行。
```

脚本 3-17　联合型的关联型（UPDATE）

Update 的情况和 Filter 类似，这里就不再阐述了。不过接下来要描述的树形查询差异就很明显了。

4） 联合型的关联型（CONNECT BY WITH FLITERING）

```
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-----------------
SQL_ID  62wv394wc9zqa, child number 0
-------------------------------------
SELECT /*+ connect_by_filtering  */ level, rpad('-',level-1,'-')||ename
AS ename, prior ename AS manager FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
Plan hash value: 1519159851
---------------------------
| Id |Operation                    | Name    |Starts |E-Rows |A-Rows |   A-Time  |Buffers |
-----------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |         |     1 |       |    14 |00:00:00.01 |     15 |
|*  1| CONNECT BY WITH FILTERING   |         |     1 |       |    14 |00:00:00.01 |     15 |
|*  2|  TABLE ACCESS FULL          | EMP     |     1 |     1 |     1 |00:00:00.01 |      7 |
|   3|  NESTED LOOPS               |         |     4 |     2 |    13 |00:00:00.01 |      8 |
|   4|   CONNECT BY PUMP           |         |     4 |       |    14 |00:00:00.01 |      0 |
|   5|   TABLE ACCESS BY INDEX ROWID| EMP     |    14 |     2 |    13 |00:00:00.01 |      8 |
|*  6|    INDEX RANGE SCAN         | EMP_MGR_I|    14 |     2 |    13 |00:00:00.01 |      5 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("MGR"=PRIOR NULL)
   2 - filter("MGR" IS NULL)
   6 - access("connect$_by$_pump$_002"."PRIOR empno "="MGR")
       filter("MGR" IS NOT NULL)
已选择 28 行。
```

脚本 3-18 联合型的关联型（CONNECT BY WITH FLITERING）

原理分析：为什么执行计划中 ID=4 的地方 STARTS 为 4 次，因为完成 4 次执行。

- 第 1 次得到 KING。
- 第 2 次得到 JONES、BLAKE、CLARK。
- 第 3 次得到 SCOTT、FORD、ALLEN、WARD、MARTIN、TURNER、JAMES、MILLER。
- 第 4 次得到 ADAMS、SMITH。

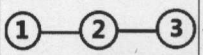
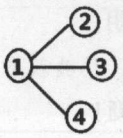
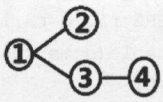
为什么执行计划中 ID=6 的部分执行 14 次，因为返回 14 条记录。

```
SELECT  /*+  connect_by_filtering  */  level,  rpad('-',level-1,'-')||ename  AS  ename,
prior ename AS manager
    FROM emp
    START WITH mgr IS NULL
    CONNECT BY PRIOR empno = mgr;

     LEVEL ENAME      MANAGER
-----------------------------
         1 KING
```

```
         2 -JONES      KING
         3 --SCOTT     JONES
         4 ---ADAMS    SCOTT
         3 --FORD      JONES
         4 ---SMITH    FORD
         2 -BLAKE      KING
         3 --ALLEN     BLAKE
         3 --WARD      BLAKE
         3 --MARTIN    BLAKE
         3 --TURNER    BLAKE
         3 --JAMES     BLAKE
         2 -CLARK      KING
         3 --MILLER    CLARK

已选择 14 行。
```

3.2.2 总结说明

读懂执行计划的关键

类型		典型语句	典型执行计划	语句对应特征图	说明
单独型		SELECT deptno, count(*) FROM emp WHERE job = 'CLERK' AND sal < 3000 GROUP BY deptno	| Id | Operation | Name | | 0 | SELECT STATEMENT | | | 1 | HASH GROUP BY | | |* 2 | TABLE ACCESS BY INDEX ROWID| EMP | |* 3 | INDEX RANGE SCAN | EMP_JOB_I |		访问顺序3，2，1。从远到近
联合型	联合型的非关联型	SELECT ename FROM emp UNION ALL SELECT dname FROM dept UNION ALL SELECT '%' FROM dual;	| Id | Operation | Name | Starts | | 0 | SELECT STATEMENT | | 1 | | 1 | UNION-ALL | | 1 | | 2 | TABLE ACCESS FULL| EMP | 1 | | 3 | TABLE ACCESS FULL| DEPT | 1 | | 4 | FAST DUAL | | 1 |		访问顺序2，3，4，1。从上到下，之间无关联性。比如2返回多少条记录对3的访问次数毫无影响
	联合型的关联型	SELECT /*+ ordered use_nl(dept) index(dept) */ * FROM emp, dept WHERE emp.deptno = dept.deptno AND emp.comm IS NULL AND dept.dname != 'SALES'	| Id |Operation | Name | Starts | | 1| NESTED LOOPS | | 1 | |* 2| TABLE ACCESS FULL | EMP | 1 | |* 3| TABLE ACCESS BY INDEX ROWID| DEPT | 10 | |* 4| INDEX UNIQUE SCAN | DEPT_PK| 10 |		访问顺序2，4，3，1。从上到下，从远到近。并且之间有关联性。比如2返回10条记录，4，3就被访问10次

3.3 从案例辨别低效 SQL

如何快速判断 SQL 执行计划是否高效，其实这是一个知识和经验的完美结合过程。我们可以敏锐地从输出执行计划的关键字中看出执行计划好坏的蛛丝马迹，下面一起来看看都有哪些维度。

3.3.1 从执行计划读出效率

1. 返回行与逻辑读比率

```
select * from t where object_id=6;
SELECT * FROM table(dbms_xplan.display_cursor(NULL,NULL,'allstats last'));
PLAN_TABLE_OUTPUT
-----------------
SQL_ID  8cxbzma1az713, child number 0
-------------------------------------
select * from t where object_id=6

Plan hash value: 1601196873
---------------------------
| Id | Operation         | Name | Starts| E-Rows| A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------
|   0| SELECT STATEMENT  |      |      1|       |      1 |00:00:00.07 |    1048 |    774 |
|*  1|  TABLE ACCESS FULL| T    |      1|     12|      1 |00:00:00.07 |    1048 |    774 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("OBJECT_ID"=6)
Note
-----
   - dynamic sampling used for this statement (level=2)
已选择 22 行。
```

脚本 3-19　返回行与逻辑读比率

说明：

总共获取 1 条记录(A-ROWS)，产生 1048 次逻辑读（Buffers），这个肯定有问题！

2. 评估值准确的重要性

```
SELECT /*+ gather_plan_statistics */ count(t2.col2)
FROM t1 ,t2 WHERE t1.id=t2.id and t1.col1 = 666;
SELECT * FROM table(dbms_xplan.display_cursor(NULL,NULL,'allstats last'));
PLAN_TABLE_OUTPUT
-----------------
SQL_ID  g048suxnxkxyr, child number 0
-------------------------------------
SELECT /*+ gather_plan_statistics */ count(t2.col2) FROM t1 ,t2 WHERE
t1.id=t2.id and t1.col1 = 666

Plan hash value: 3711554156
---------------------------
|Id | Operation                     | Name   |Starts| E-Rows| A-Rows |   A-Time   |Buffers|
-----------------------------------------------------------------------------------------------
```

```
|  0| SELECT STATEMENT                |          |     1|       |      1|00:00:00.30 | 94651 |
|  1|  SORT AGGREGATE                 |          |     1|      1|      1|00:00:00.30 | 94651 |
|  2|   NESTED LOOPS                  |          |     1|       |  75808|00:00:00.31 | 94651 |
|  3|    NESTED LOOPS                 |          |     1|     32|  75808|00:00:00.19 | 18843 |
|  4|     TABLE ACCESS BY INDEX ROWID| T1       |     1|     32|  80016|00:00:00.08 |  1771 |
|* 5|      INDEX RANGE SCAN           | T1_COL1|     1|     32|  80016|00:00:00.03 |   169 |
|* 6|     INDEX UNIQUE SCAN           | T2_PK  | 80016|      1|  75808|00:00:00.08 | 17072 |
|  7|    TABLE ACCESS BY INDEX ROWID | T2       | 75808|      1|  75808|00:00:00.08 | 75808 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("T1"."COL1"=666)
   6 - access("T1"."ID"="T2"."ID")

已选择 26 行。
```

脚本 3-20　评估值准确的重要性

3. 类型转换需认真关注

```
select * from t_col_type where id=6;
执行计划
--------
Plan hash value: 3191204463
---------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     1 |    36 |     9   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T_COL_TYPE |     1 |    36 |     9   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(TO_NUMBER("ID")=6)
```

脚本 3-21　类型转换需认真关注

4. 请小心递归调用部分

```
select sex_id,
first_name||' '||last_name full_name,
get_sex_name(sex_id) gender
from people;
执行计划
-------------------------------------------------------------------------------
Plan hash value: 2528372185
-------------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        | 80635 |   16M|   137   (1)| 00:00:02 |
|   1 |  TABLE ACCESS FULL| PEOPLE | 80635 |   16M|   137   (1)| 00:00:02 |
-------------------------------------------------------------------------------
```

```
Note
-----
   - dynamic sampling used for this statement (level=2)

统计信息
----------------------------------------------------------------------------
      73121  recursive calls
          0  db block gets
     517142  consistent gets
          0  physical reads
          0  redo size
    3382143  bytes sent via SQL*Net to client
      54029  bytes received via SQL*Net from client
       4876  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      73121  rows processed
```

脚本 3-22　请小心递归调用部分

5. 注意表的访问次数

```
SELECT /*+ gather_plan_statistics */ count(t2.col2) FROM t1 ,t2 WHERE
t1.id=t2.id and t1.col1 = 666

Plan hash value: 3711554156
-------------------------------------------------------------------------------------------------
|Id| Operation                      | Name    | Starts| E-Rows | A-Rows |   A-Time    |Buffers|
-------------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT               |         |      1|        |      1 |00:00:00.30 | 94651 |
| 1|  SORT AGGREGATE                |         |      1|      1 |      1 |00:00:00.30 | 94651 |
| 2|   NESTED LOOPS                 |         |      1|        |  75808 |00:00:00.31 | 94651 |
| 3|    NESTED LOOPS                |         |      1|     32 |  75808 |00:00:00.19 | 18843 |
| 4|     TABLE ACCESS BY INDEX ROWID| T1      |      1|     32 |  80016 |00:00:00.08 |  1771 |
|*5|      INDEX RANGE SCAN          | T1_COL1 |      1|     32 |  80016 |00:00:00.03 |   169 |
|*6|     INDEX UNIQUE SCAN          | T2_PK   |  80016|      1 |  75808 |00:00:00.08 | 17072 |
| 7|    TABLE ACCESS BY INDEX ROWID | T2      |  75808|      1 |  75808 |00:00:00.08 | 75808 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
-------------------------------------------------------------------------------------------------
   5 - access("T1"."COL1"=666)
   6 - access("T1"."ID"="T2"."ID")
已选择 26 行。
```

脚本 3-23　注意表的访问次数

6. 注意表真实访问行数

```
select *   from (select t1.*, rownum as rn from t1, t2 where
t1.object_id = t2.id1) a  where a.rn >= 1    and a.rn <= 10
Plan hash value: 3062220019
```

```
-------------------------------------------------------------------
| Id  | Operation            | Name | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |     10 |
|*  1 |  VIEW                |      |      1 |   1008 |     10 |
|   2 |   COUNT              |      |      1 |        |    943 |
|*  3 |    HASH JOIN         |      |      1 |   1008 |    943 |
|   4 |     TABLE ACCESS FULL| T2   |      1 |   1000 |   1000 |
|   5 |     TABLE ACCESS FULL| T1   |      1 |  70183 |  73156 |


select *   from (select t1.*, rownum as rn from t1, t2 where
t1.object_id = t2.id1 and rownum<=10) a  where a.rn >= 1
Plan hash value: 1802812661
-------------------------------------------------------------------
| Id  | Operation            | Name | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |     10 |
|*  1 |  VIEW                |      |      1 |     10 |     10 |
|*  2 |   COUNT STOPKEY      |      |      1 |        |     10 |
|*  3 |    HASH JOIN         |      |      1 |   1008 |     10 |
|   4 |     TABLE ACCESS FULL| T2   |      1 |   1000 |   1000 |
|   5 |     TABLE ACCESS FULL| T1   |      1 |  70183 |     10 |
```

脚本 3-24　注意表真实访问行数

7. 谨慎观察排序与否

```
select * from t where object_id>2 order by object_id;
执行计划
----------------------------------------------------------------------------------------
Plan hash value: 961378228
----------------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 81694 |    16M|       |  3973   (1)| 00:00:48 |
|   1 |  SORT ORDER BY     |      | 81694 |    16M|    19M|  3973   (1)| 00:00:48 |
|*  2 |   TABLE ACCESS FULL| T    | 81694 |    16M|       |   293   (1)| 00:00:04 |
----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
----------------------------------------------------------------------------------------
   2 - filter("OBJECT_ID">2)
Note
-----
   - dynamic sampling used for this statement (level=2)
统计信息
----------------------------------------------------------------------------------------
          1  sorts (memory)
          0  sorts (disk)
      73155  rows processed
```

脚本 3-25　谨慎观察排序与否

3.3.2 执行计划效率总结

如何通过执行计划识别低效的SQL		
注意点	对应执行计划部分	简要分析
返回行与逻辑读比率	A-Rows | Buffers 1 | 1048 **1 | 1048**	真实返回1条记录，花费1048个逻辑读，性能有大问题
评估值准确的重要性	| E-Rows| A-Rows | | 1| | 1| 1| | | 75808| **| 32| 75808|**	评估32条，真实75808条，非常不准确，执行计划的准确性很让人怀疑
类型转换需认真关注	|* 1 | TABLE ACCESS FULL| T_COI Predicate Information (identifiec 1 - filter(TO_NUMBER("ID")=6).	类型转化，一般都用不到索引，所以是filter而不是accss
请小心递归调用部分	统计信息 73121 recursive calls 0 db block gets 517142 consistent gets	产生73121次递归调用，这一般是SQL带函数引发的
注意表的访问次数	| Starts| E-Rows | A-Rows | | 1| 32 | 80016 | **| 80016| 1 | 75808 |**	表被访问了80016次，偏多了，一般不考虑NL连接，要考虑Hash连接等
注意表真实访问行数	E-Rows | A-Rows 1000 | 1000 70183 | **10**	除了预测错外，要在执行计划中有COUNT STOPKEY关键字。还可能是rownum分页查询的执行计划，表示在第10行就停止前进了
谨慎观察排序与否	统计信息 1 sorts (memory) 0 sorts (disk) 73155 rows processed	查看该SQL是否排序，此处说明在内存中排序，并未到磁盘中排序

3.4 本章习题、总结与延伸

习题 1：说说 Oracle 为什么会有多种执行计划?

习题 2：请根据下图所示执行计划，请用单独型和联合型来画出执行计划图，并标上执行的先后顺序。

```
--------------------
|   0 | UPDATE STATEMENT                |       |
|   1 |  UPDATE                         | T     |
|   2 |   NESTED LOOPS                  |       |
|*  3 |    TABLE ACCESS FULL            | T     |
|*  4 |    INDEX UNIQUE SCAN            | T_PK  |
|   5 |   SORT AGGREGATE                |       |
|*  6 |    TABLE ACCESS BY INDEX ROWID| T     |
|   7 |     INDEX FULL SCAN             | I     |
|   8 |   TABLE ACCESS BY INDEX ROWID | T     |
|*  9 |    INDEX UNIQUE SCAN            | T_PK  |
--------------------------------------------------
```

习题 3：请先思考如何通过体检的方式来发现数据库统计信息是否存在问题，然后再设法寻找如下几个脚本：

- 如何看自动收集统计信息功能是否开？
- 哪些表的统计信息未收集或过时了？
- 哪些列统计信息未收集或过时了？
- 哪些索引统计信息未收集或过时了？
- 排查全局临时表被收集统计信息。

习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

第 4 章　运筹帷幄——左右 SQL 执行计划妙招

左右执行计划就是左右人生

经过前面的学习，我们不仅能快速获取整个数据库的整体信息，还能迅速得到 SQL 的所有详细信息。如果能将笔者研发的自动获取脚本应用在工作中，在解决问题上则无疑更是如虎添翼。

接下来笔者将教会大家如何真正读懂执行计划，这个其实并不容易。循序渐进学完这三个章节后，优化的大方向基本上都清晰了，剩下的就是具体的优化实施。可能会：修改数据库及主机相关性能参数，根据业务规则修改 SQL 代码，重新收集统计信息获取更准确的执行计划，等等。

停！你想过在不改写 SQL 和不重新收集统计信息的情况下，改变 SQL 的执行计划吗？

这，可能吗？

当然可以！

不过，Oracle 不是很智能。如果统计信息正确，应该可以得到正确的执行计划；如果统计信息不准确，用收集的方式更合理吧？

你说得太好了！不过假如生产中真的出现了某条 SQL 由于统计信息不准确执行非常慢的情况，你立即收集统计信息这个动作会影响生产吧？如果要等到系统较闲可以收集的时候，这个 SQL 不是要影响性能很久吗？

哦，说的是啊。

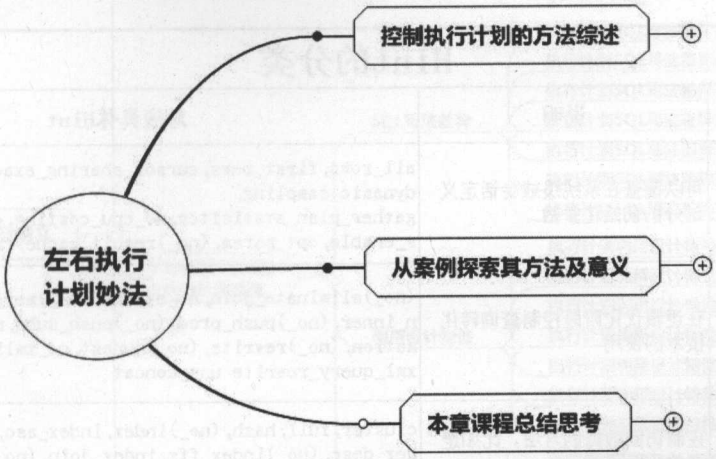
另外要是万一出现 BUG，某 SQL 无论如何收集统计信息都得不到正确的执行计划，而你明知正确的执行计划该是怎么样子，那你该怎么办呢？

自己动手改变执行计划。

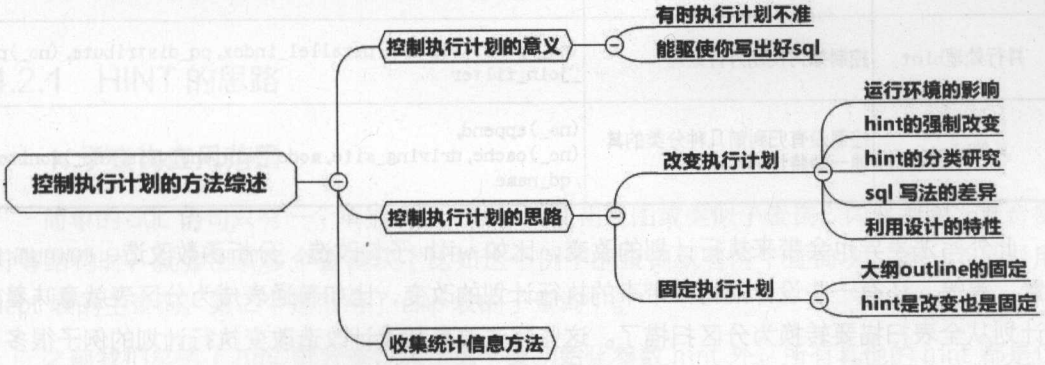
正确！

看来，左右 SQL 执行计划还真有用，那我要好好跟你学学。

OK，让我们开始吧，先看看总体学习思路，如下图所示：



4.1　控制执行计划的方法综述



4.1.1　控制执行计划的意义

前面已经讲过了，这里简单总结为两点：1.可以临时在高峰期解决问题，避免因收集统计信息带来的开销；2.有 BUG 导致执行计划一直不对，只好用人工控制来处理。

4.1.2 控制执行计划的思路

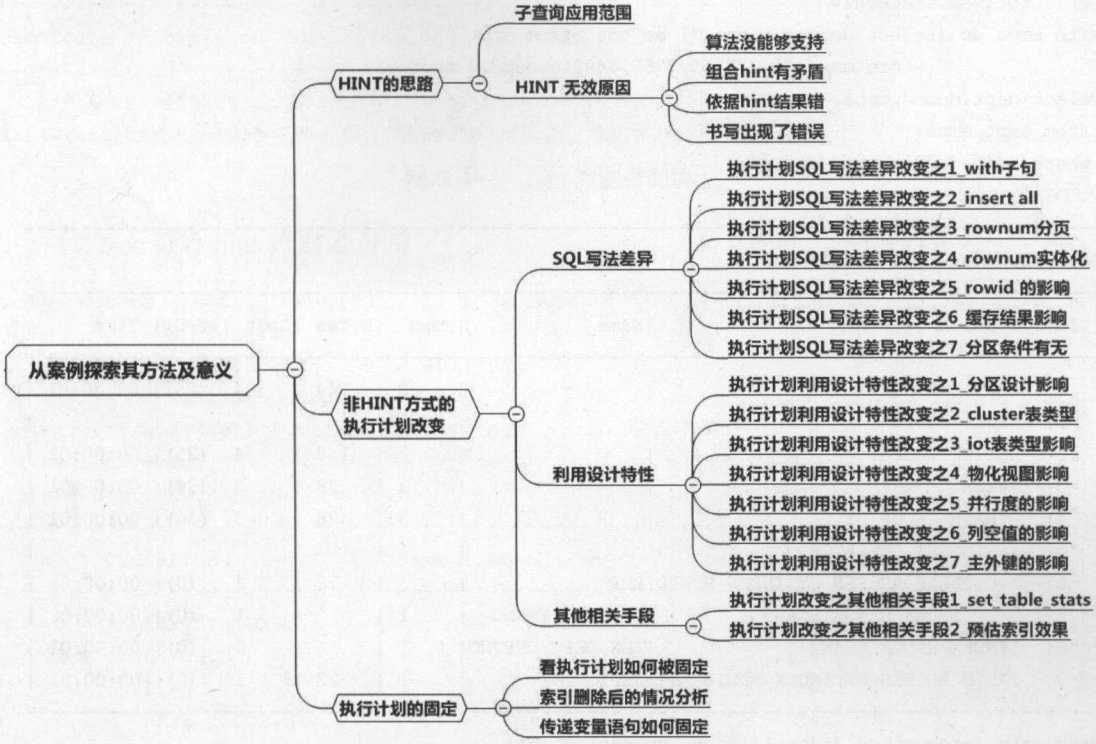
1. 关于 Hint

一般来说，环境的影响会改变 SQL 的执行计划，除此之外就是 Hint 会强制让 Oracle 根据你的要求走对应的执行计划。Hint 的种类有多种，如下表所示：

Hint的分类		
分类	**说明**	**对应具体Hint**
初始化参数hint	可以覆盖在系统级或会话定义部分的初始化参数	all_rows, first_rows, cursor_sharing_exact, dynamic_sampling, gather_plan_statisitcs, no_cpu_costing, optimizer_features_enable, opt_param, (no_)result_cache, rule;
查询转化hint	在逻辑优化阶段控制查询转化技术的使用	(no_)eliminate_join, no_expand, (no_)merge, (no_)outer_join_inner, (no_)push_pred, (no_)push_subq, no_query_transformation, (no_)rewrite, (no_)nunest, no_xmlindex_rewrite, no_xml_query_rewrite use_concat
访问路径hint	控制访问数据的方法，比如是否使用索引等	cluster, full, hash, (no_)index, index_asc, index_combine, index_desc, (no_)index_ffs, index_join, (no_)index_ss, index_ss_asc, index_ss_desc
连接提示hint	不仅控制连接的方法，还控制连接表的顺序	leading, (no_)nljbatching, ordered, (no_)start_transformation, (no_)swap_join_inputs, (no_)use_hash, (no_)use_merge, use_merge_cartesian, (no_)use_nl, use_nl_with_index
并行处理hint	控制如何使用并行处理	(no_)parallel, (no_)parallel_index, pq_distribute, (no_)px_join_filter
其他hint	控制没有归到前几种分类的其他一些特性的使用	(no_)append, (no_)cache, driving_site, model_min_analysis, (no_)monitor, qd_name

此外写法差异也会带来执行计划的改变，比如 with 子句改造、分析函数改造、rownum 的位置，等等。还有一些设计的特性带来的执行计划的改变，比如普通表成为分区表就意味着执行计划从全表扫描要转换为分区扫描了。这些写法改变和设计改造改变执行计划的例子很多，在本章的案例部分将会详细解说。

4.2　从案例探索其方法及意义



4.2.1　HINT 的思路

1. 子查询应用范围

简单的 SQL 语句只有一个单独的查询块。当使用视图或类似子查询、内联视图、集合操作符等结构时，就会出现多个查询块（比如这个例子的查询就有两个查询块，第一个是引用了 dept 表的主查询，第二个是引用了 emp 表的子查询）。

之前我们总结了 hint 的分类，除了第一类初始化参数 hint 外，所有其他的 hint 都是仅针对单个查询块起作用。下面来看如何让各个模块的 HINT 生效的各种方法。

环境准备：

```
drop table emp purge;
create table emp as select * from scott.emp;
create index idx_emp_deptno on emp(deptno);
create index idx_emp_empno on emp(empno);
drop table dept purge;
create table dept as select * from scott.dept;
create index idx_dept_deptno on dept(deptno);
```

请看如下语句的执行计划：

```
set linesize 1000
set pagesize 2000
set autotrace traceonly
with emps as (select deptno,count(*) as cnt  from emp
              where empno in (7369,7782,7499)group by deptno)
select dept.dname,emps.cnt
 from dept,emps
 where dept.deptno=emps.deptno;
执行计划
----------------------------------------------------------------------------------------------
Plan hash value: 174555140
----------------------------------------------------------------------------------------------
| Id|Operation                          |Name           |Rows  |Bytes |Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT                   |               |    3 |  144 |    4  (25)| 00:00:01 |
|  1| NESTED LOOPS                      |               |      |      |           |          |
|  2|  NESTED LOOPS                     |               |    3 |  144 |    4  (25)| 00:00:01 |
|  3|   VIEW                            |               |    3 |   78 |    3  (34)| 00:00:01 |
|  4|    HASH GROUP BY                  |               |    3 |   78 |    3  (34)| 00:00:01 |
|  5|     INLIST ITERATOR               |               |      |      |           |          |
|  6|      TABLE ACCESS BY INDEX ROWID|EMP            |    3 |   78 |    2   (0)| 00:00:01 |
|* 7|       INDEX RANGE SCAN            |IDX_EMP_EMPNO  |    1 |      |    1   (0)| 00:00:01 |
|* 8|   INDEX RANGE SCAN                |IDX_DEPT_DEPTNO|    1 |      |    0   (0)| 00:00:01 |
|  9|  TABLE ACCESS BY INDEX ROWID      |DEPT           |    1 |   22 |    1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   7 - access("EMPNO"=7369 OR "EMPNO"=7499 OR "EMPNO"=7782)
   8 - access("DEPT"."DEPTNO"="EMPS"."DEPTNO")
```

（1）控制在所在的查询块内

两个 hint 的有效区域都被严格控制在它们所在的查询块内，如下：

```
with emps as (select /*+full(emp)*/ deptno,count(*) as cnt
              from emp where empno in (7369,7782,7499)
              group by deptno)
select /*+full(dept)*/ dept.dname,emps.cnt
 from dept,emps
 where dept.deptno=emps.deptno;
执行计划
------------------------------------------------------------------------------
Plan hash value: 2415981340
------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     3 |   144 |     8  (25)| 00:00:01 |
|*  1 |  HASH JOIN           |      |     3 |   144 |     8  (25)| 00:00:01 |
|   2 |   VIEW               |      |     3 |    78 |     4  (25)| 00:00:01 |
|   3 |    HASH GROUP BY     |      |     3 |    78 |     4  (25)| 00:00:01 |
```

```
|*  4 |     TABLE ACCESS FULL| EMP  |     3 |    78 |     3   (0)| 00:00:01 |
|   5 |   TABLE ACCESS FULL  | DEPT |     4 |    88 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("DEPT"."DEPTNO"="EMPS"."DEPTNO")
   4 - filter("EMPNO"=7369 OR "EMPNO"=7499 OR "EMPNO"=7782)
```

脚本 4-1　hint 控制查询块

（2）全局的 hint 的别名引用

```
with emps as (select  deptno,count(*) as cnt
               from emp
               where empno in (7369,7782,7499)
              group by deptno)
select /*+full(dept) full(emps.emp)*/ dept.dname,emps.cnt
 from dept,emps
 where dept.deptno=emps.deptno;
 ------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     3 |   144 |     8  (25)| 00:00:01 |
|*  1 |  HASH JOIN           |      |     3 |   144 |     8  (25)| 00:00:01 |
|   2 |   VIEW               |      |     3 |    78 |     4  (25)| 00:00:01 |
|   3 |    HASH GROUP BY     |      |     3 |    78 |     4  (25)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL| EMP  |     3 |    78 |     3   (0)| 00:00:01 |
|   5 |   TABLE ACCESS FULL  | DEPT |     4 |    88 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("DEPT"."DEPTNO"="EMPS"."DEPTNO")
   4 - filter("EMPNO"=7369 OR "EMPNO"=7499 OR "EMPNO"=7782)
```

脚本 4-2　全局的 hint 的别名引用

（3）用 qb_name 定义方式

有的时候 SQL 不写子查询的别名，比如 WHERE 条件中的子查询显然用不到别名，这时可以用 qb_name 定义方式，其中，qb_name(main)是固定必须写的，比如如下的 full(@main dept)就是来引用主表的。

```
with emps as (select  /*+qb_name(sq)*/ deptno,count(*) as cnt
               from emp
               where empno in (7369,7782,7499)
              group by deptno)
select /*+qb_name(main) full(@main dept) full(@sq emp)*/ dept.dname,emps.cnt
 from dept,emps
 where dept.deptno=emps.deptno;
```

```
执行计划
----------------------------------------------------------
Plan hash value: 2415981340
-------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     3 |   144 |     8  (25)| 00:00:01 |
|*  1 |  HASH JOIN           |      |     3 |   144 |     8  (25)| 00:00:01 |
|   2 |   VIEW               |      |     3 |    78 |     4  (25)| 00:00:01 |
|   3 |    HASH GROUP BY     |      |     3 |    78 |     4  (25)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL| EMP  |     3 |    78 |     3   (0)| 00:00:01 |
|   5 |   TABLE ACCESS FULL  | DEPT |     4 |    88 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("DEPT"."DEPTNO"="EMPS"."DEPTNO")
   4 - filter("EMPNO"=7369 OR "EMPNO"=7499 OR "EMPNO"=7782)
```

脚本 4-3　用 qb_name 定义方式

2. HINT 无效原因

HINT 在使用过程中时常会遇到无法生效的情况，这时我们要冷静分析判断，一般来说，就是算法无法支持、Hint 有矛盾、根据 Hint 的结果执行会错、书写语法错这几个原因。

（1）算法没能够支持

环境准备：

```
DROP TABLE t1 CASCADE CONSTRAINTS PURGE;
DROP TABLE t2 CASCADE CONSTRAINTS PURGE;
CREATE TABLE t1 (
     id NUMBER NOT NULL,
     n NUMBER,
     contents VARCHAR2(4000)
   );
CREATE TABLE t2 (
     id NUMBER NOT NULL,
     t1_id NUMBER NOT NULL,
     n NUMBER,
     contents VARCHAR2(4000)
   );
execute dbms_random.seed(0);
INSERT INTO t1
     SELECT  rownum,  rownum, dbms_random.string('a', 50)
       FROM dual
     CONNECT BY level <= 100
      ORDER BY dbms_random.random;
INSERT INTO t2 SELECT rownum, rownum, rownum, dbms_random.string('b', 50) FROM dual
     CONNECT BY level <= 100000
```

```
    ORDER BY dbms_random.random;
COMMIT;
```

语句 1，使用 use_hash 的 Hint 希望能走 Hash 连接，结果实际是 NL 连接，因为 Hash 连接不支持连接条件是 t1.id > t2.t1_id 这样不等的写法，如下：

```
set linesize 1000
set autotrace traceonly explain
--语句 1
SELECT /*+ leading(t1) use_hash(t2)*/ *
  FROM t1, t2
  WHERE t1.id > t2.t1_id
  AND t1.n = 19;
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    50 |  6150 |   276   (1)| 00:00:04 |
|   1 |  NESTED LOOPS      |      |    50 |  6150 |   276   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T2   |    50 |  3300 |   273   (1)| 00:00:04 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
-------------------------------------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID">"T2"."T1_ID")
```

脚本 4-4　use_hash 的算法不支持不等值连接

语句 2 的连接条件是 Like，同样只能适用于 NL，而不能适用于其他，这里试验 use_merge，一样以失败告终：

```
SELECT /*+ leading(t1) use_merge(t2)*/ *
  FROM t1, t2
  WHERE t1.id like t2.t1_id
  AND t1.n = 19;
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |  5000 |  600K|   276   (1)| 00:00:04 |
|   1 |  NESTED LOOPS      |      |  5000 |  600K|   276   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T2   |  5000 |  322K|   273   (1)| 00:00:04 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
-------------------------------------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter(TO_CHAR("T1"."ID") LIKE TO_CHAR("T2"."T1_ID"))
```

脚本 4-5　use_hash 的算法不支持 LIKE 连接

（2）组合 Hint 有矛盾

我们接上面的例子，继续运行一个 SQL 语句，如下：

```
Set linesize 1000
set pagesize 2000
alter session set statistics_level=all ;
SELECT /*+ leading(t2) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id;
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |    100 |00:00:00.11 |    1019 |
|*  1 |  HASH JOIN         |      |      1 |    100 |    100 |00:00:00.11 |    1019 |
|   2 |   TABLE ACCESS FULL| T2   |      1 |  98778 |    100K|00:00:00.02 |    1005 |
|   3 |   TABLE ACCESS FULL| T1   |      1 |    100 |    100 |00:00:00.01 |      14 |
------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
------------------------------------------------------------------------------------
   1 - access("T1"."ID"="T2"."T1_ID")
Note
-----
   - dynamic sampling used for this statement (level=2)
已选择 25 行。
```

脚本 4-6　组合 Hint 有矛盾

这里我们发现，我们明明使用 use_nl 的 Hint，走的却是 Hash join，这是为啥呢？因为这里 use_nl(t2)表示 t2 被驱动，也就是 t2 表后访问，而 leading(t2)却告诉我们这是表示 t2 表要前驱，先访问。这不是矛盾吗？所以这个 hint 失效了。

（3）依据 Hint 结果执行会错

请看如下语句，使用 index(t,idx_object_id)希望语句强制走索引，结果如何呢？请看：

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id);
set linesize 1000
set pagesize 2000
explain plan for
select /*+index(t,idx_object_id)*/ count(*) from t;
select * from table(dbms_xplan.display());
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
Plan hash value: 2966233522
------------------------------------------------------------------------
```

```
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 74811 |   292   (1)| 00:00:04 |
-----------------------------------------------------------------------
```

脚本 4-7　依据 Hint 结果执行会错

发现依然走的是全表扫描，Hint 失效了。真正原因是，如果走索引，那就要依赖索引来回答条数的问题。这里有巨大风险，因为索引不存储空值，而索引列并没有保证非空，这里的值会不正确。

（4）书写出现了错误

如果 SQL 的表有别名，必须用别名而不能用原表名，否则无法生效，这点要牢记。比如如下：

```
drop table test purge;
create table test as select * from dba_objects;
create index idx_test_objid on test(object_id);
set linesize 1000
set pagesize 2000
explain plan for
select /*+index(test,idx_test_objid)*/ * from test t where object_id>0;
select * from table(dbms_xplan.display());
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
Plan hash value: 1357081020
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 67742 |    13M|   293   (1)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| TEST | 67742 |    13M|   293   (1)| 00:00:04 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
-------------------------------------------------------------------------------
   1 - filter("OBJECT_ID">0)
Note
-----
   - dynamic sampling used for this statement (level=2)
```

脚本 4-8　Hint 书写出现了错误

很显然，这里如果是如下两种写法都不会出现问题：要么用别名 t，要么就干脆这个 SQL 语句本身就没有别名。

```
select /*+index(t,idx_test_objid)*/ * from test t where object_id>0;
select /*+index(test,idx_test_objid)*/ * from test  where object_id>0;
```

当然，还有的是把 index(t,t_idx) 不小心写成了 indexx(t,t_idx)，把/*+full(t)*/ 写成了 /*full(t)*+/ 等诸如此类的低级错误，一定要细心哦，这里就无须举例说明了。

4.2.2 非 HINT 方式的执行计划改变

1. SQL 写法的差异

（1）执行计划 SQL 写法差异改变之 1_with 子句

环境准备：

```
drop table t_with;
CREATE TABLE T_WITH AS SELECT ROWNUM ID, A.* FROM DBA_SOURCE A WHERE ROWNUM < 100001;
SET autotrace traceonly
Set linesize 1000
```

语句 1：

```
SELECT ID, NAME FROM T_WITH
WHERE ID IN
(SELECT MAX(ID) FROM T_WITH
 UNION ALL
 SELECT MIN(ID) FROM T_WITH
 UNION ALL
 SELECT TRUNC(AVG(ID)) FROM T_WITH);
执行计划
----------------------------------------------------------------------------------------
Plan hash value: 647530712
----------------------------------------------------------------------------------------
| Id  | Operation              | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |     3 |   129 |  1382   (1)| 00:00:17 |
|*  1 |  HASH JOIN             |          |     3 |   129 |  1382   (1)| 00:00:17 |
|   2 |   VIEW                 | VW_NSO_1 |     3 |    39 |  1035   (1)| 00:00:13 |
|   3 |    HASH UNIQUE         |          |     3 |    39 |  1035  (67)| 00:00:13 |
|   4 |     UNION-ALL          |          |       |       |            |          |
|   5 |      SORT AGGREGATE    |          |     1 |    13 |            |          |
|   6 |       TABLE ACCESS FULL| T_WITH   | 91060 |  1156K|   345   (1)| 00:00:05 |
|   7 |      SORT AGGREGATE    |          |     1 |    13 |            |          |
|   8 |       TABLE ACCESS FULL| T_WITH   | 91060 |  1156K|   345   (1)| 00:00:05 |
|   9 |      SORT AGGREGATE    |          |     1 |    13 |            |          |
|  10 |       TABLE ACCESS FULL| T_WITH   | 91060 |  1156K|   345   (1)| 00:00:05 |
|  11 |   TABLE ACCESS FULL    | T_WITH   | 91060 |  2667K|   345   (1)| 00:00:05 |
----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
----------------------------------------------------------------------------------------
   1 - access("ID"="MAX(ID)")
Note
-----
```

```
   - dynamic sampling used for this statement (level=2)
统计信息
----------------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
       4969  consistent gets
          0  physical reads
          0  redo size
        558  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          3  rows processed
```

语句 2：

```
WITH AGG AS (SELECT MAX(ID) MAX, MIN(ID) MIN, TRUNC(AVG(ID)) AVG FROM T_WITH)
 SELECT ID, NAME FROM T_WITH
 WHERE ID IN
 ( SELECT MAX FROM AGG  UNION ALL  SELECT MIN FROM AGG  UNION ALL  SELECT AVG FROM AGG);
执行计划
----------------------------------------------------------------------------------------------
Plan hash value: 3705751949
----------------------------------------------------------------------------------------------
| Id |Operation                  | Name                          |Rows  | Bytes |Cost (%CPU)|
----------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT           |                               |    3 |   129 |  697   (1)|
|   1| TEMP TABLE TRANSFORMATION|                               |      |       |           |
|   2|  LOAD AS SELECT           | SYS_TEMP_0FD9D6605_3B91BA4    |      |       |           |
|   3|   SORT AGGREGATE          |                               |    1 |    13 |           |
|   4|    TABLE ACCESS FULL      | T_WITH                        |91060 |  1156K|  345   (1)|
|*  5|  HASH JOIN                |                               |    3 |   129 |  352   (1)|
|   6|   VIEW                    | VW_NSO_1                      |    3 |    39 |    6   (0)|
|   7|    HASH UNIQUE            |                               |    3 |    39 |    6  (67)|
|   8|     UNION-ALL             |                               |      |       |           |
|   9|      VIEW                 |                               |    1 |    13 |    2   (0)|
|  10|       TABLE ACCESS FULL   | SYS_TEMP_0FD9D6605_3B91BA4    |    1 |    13 |    2   (0)|
|  11|      VIEW                 |                               |    1 |    13 |    2   (0)|
|  12|       TABLE ACCESS FULL   | SYS_TEMP_0FD9D6605_3B91BA4    |    1 |    13 |    2   (0)|
|  13|      VIEW                 |                               |    1 |    13 |    2   (0)|
|  14|       TABLE ACCESS FULL   | SYS_TEMP_0FD9D6605_3B91BA4    |    1 |    13 |    2   (0)|
|  15|   TABLE ACCESS FULL       | T_WITH                        |91060 |  2667K|  345   (1)|
----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
----------------------------------------------------------------------------------------------
   5 - access("ID"="MAX")
Note
-----
   - dynamic sampling used for this statement (level=2)
```

```
统计信息
----------------------------------------------------------------------------------------------------
          2  recursive calls
          8  db block gets
       2496  consistent gets
          1  physical reads
        600  redo size
        558  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          3  rows processed
```

脚本 4-9　执行计划 SQL 写法差异改变之 1_with 子句

亲们，看到什么变化了？没错，有点看不懂 SYS_TEMP_0FD9D6605_3B91BA4。这就是 WITH 子句的魅力，一次获取后缓存在内存中多次使用，你看不懂的执行计划是很牛的。

（2）执行计划 SQL 写法差异改变之 2_insert all

环境准备：

```
drop table t1 purge;
create table t1 as select  * from dba_objects where 1=2;
drop table  t2 purge;
create  table t2 as select * from dba_objects where 1=2;
drop table t purge;
create  table t as select * from dba_objects;
```

普通的插入语句：

```
SQL_ID  d4y6zf9bsqk1b, child number 0
-------------------------------------------------------------------------------------------------
insert  into  t1 select * from t
Plan hash value: 1601196873
-------------------------------------------------------------------------------------------------
| Id  | Operation                | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | INSERT STATEMENT         |      |      1 |        |      0 |00:00:00.15 |   10935 |
|   1 |  LOAD TABLE CONVENTIONAL |      |      1 |        |      0 |00:00:00.15 |   10935 |
|   2 |   TABLE ACCESS FULL      | T    |      1 |  74811 |  73107 |00:00:00.02 |    1047 |

insert  into  t2 select * from t

Plan hash value: 1601196873
-------------------------------------------------------------------------------------------------
| Id  | Operation                | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | INSERT STATEMENT         |      |      1 |        |      0 |00:00:00.17 |   10935 |
|   1 |  LOAD TABLE CONVENTIONAL |      |      1 |        |      0 |00:00:00.17 |   10935 |
```

```
|   2 |   TABLE ACCESS FULL      | T    |      1 |  74811 |  73107 |00:00:00.02 |    1047 |
-----------------------------------------------------------------------------------------------
```

假如 T 表的数据不会变化，看如下语句：

```
set linesize 1000
set pagesize 2000
set autotrace off
ALTER SESSION SET statistics_level = all;
rollabck;
insert all
   into  t1
   into  t2
select * from t;

SQL> SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------
SQL_ID  795gsytaz3bkt, child number 0
-----------------------------------------------------------------------------------------------
insert all    into  t1    into  t2 select * from t
Plan hash value: 1563282152
-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------
|   0 | INSERT STATEMENT    |      |      1 |        |      0 |00:00:00.42 |   25858 |    734 |
|   1 |  MULTI-TABLE INSERT |      |      1 |        |      0 |00:00:00.42 |   25858 |    734 |
|   2 |   TABLE ACCESS FULL | T    |      1 |  74811 |  73107 |00:00:00.16 |    1047 |    734 |
-----------------------------------------------------------------------------------------------
Note
-----
   - dynamic sampling used for this statement (level=2)
已选择 18 行。
```

脚本 4-10　执行计划 SQL 写法差异改变之 2_insert all

大家看到一个新词 MULTI-TABLE INSERT，这就是经典的 Insert all 多表插入对应的执行计划。从性能上来看，insert all 不一定会有优势，但是当分开写和合并写不等价的时候，分开写要很麻烦，比如锁表，比如中间表，这样性能就要比 insert all 差多了！

（3）执行计划 SQL 写法差异改变之 3_rownum 分页

环境准备：

```
drop table t;
create table t as select * from dba_objects;
set linesize 1000
set pagesize 2000
set autotrace off
```

语句 1：

```
select * from (select t.*,rownum as rn  from t t) a where a.rn>=1 and a.rn<=10;
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
SQL_ID  37kdqpc1116gu, child number 1
------------------------------------------------------------------------------------------------
select * from (select t.*,rownum as rn  from t t) a where a.rn>=1 and
a.rn<=10
Plan hash value: 1961128543
------------------------------------------------------------------------------------------------
| Id | Operation           | Name| Starts| E-Rows | A-Rows |   A-Time   |Buffers |Reads  |
------------------------------------------------------------------------------------------------
|   0| SELECT STATEMENT    |     |      1|        |     10 |00:00:00.13 |   1048 |   713 |
|*  1|  VIEW               |     |      1|  91662 |     10 |00:00:00.13 |   1048 |   713 |
|   2|   COUNT             |     |      1|        |  73118 |00:00:00.10 |   1048 |   713 |
|   3|    TABLE ACCESS FULL| T   |      1|  91662 |  73118 |00:00:00.09 |   1048 |   713 |
------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
------------------------------------------------------------------------------------------------
   1 - filter(("A"."RN"<=10 AND "A"."RN">=1))
Note
-----
   - dynamic sampling used for this statement (level=2)
已选择 25 行。
```

语句 2：

```
select * from (select t.*,rownum as rn  from t t where rownum<=10) a where a.rn>=1 ;

SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
SQL_ID  gyumtnrhhw672, child number 0
------------------------------------------------------------------------------------------------
select * from (select t.*,rownum as rn  from t t where rownum<=10) a where a.rn>=1
Plan hash value: 3593519476
------------------------------------------------------------------------------------------------
| Id  | Operation           | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |        |     10 |00:00:00.01 |       5 |
|*  1 |  VIEW               |      |      1 |     10 |     10 |00:00:00.01 |       5 |
|*  2 |   COUNT STOPKEY     |      |      1 |        |     10 |00:00:00.01 |       5 |
|   3 |    TABLE ACCESS FULL| T    |      1 |  91662 |     10 |00:00:00.01 |       5 |
------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
------------------------------------------------------------------------------------------------
   1 - filter("A"."RN">=1)
   2 - filter(ROWNUM<=10)
```

```
Note
-----
   - dynamic sampling used for this statement (level=2)
已选择 26 行。
```

脚本 4-11　执行计划 SQL 写法差异改变之 3_rownum 分页

说明：

语句 1 和语句 2 在写法上是等价的。但是语句 2 走的是 COUNT STOPKEY，请分别观察语句 1 和语句 2 的执行计划中的 A-ROWS 部分。

（4）执行计划 SQL 写法差异改变之 4_rownum 实体化

不少开发人员喜欢使用复杂子查询写各类 SQL，其实很多时候，要是不注意某些细节，就会出现很多问题，如性能问题甚至结果值不对。我们举个某运营商系统的例子，具体如下：

```
-----略去
select ta.tch_id, ta.flow_id
          from tache pr,
               (select TCH_ID, c.flow_id
                  from event_q a, staff_event b, tache c
                 where (a.event_type = '2' OR a.event_type = '1')
                   and a.event_id = b.event_id
                   and (a.flag is null or a.flag = '1')
                   and b.staff_id = 1
                   and a.content_id = c.tch_id) ta
         where ta.flow_id = pr.sub_flow_id(+)
           and pr.flow_id is null
-------略去
```

这条 SQL 本身很复杂，我们把其他部分略去，只显示典型的子查询这一小部分，另外主要也是这一小部分慢，如果把这一小部分注释掉，执行可在 1s 内完成，否则需要 30s 才可以完成。

我们单看这一小部分的执行计划，截取如下：

```
|  31 |        NESTED LOOPS                 |                  |     1 |    53 |    63   (2)| 00:00:01 |
|  32 |         MERGE JOIN CARTESIAN        |                  |     1 |    34 |    62   (2)| 00:00:01 |
|* 33 |          FILTER                     |                  |       |       |            |          |
|* 34 |           HASH JOIN RIGHT OUTER     |                  |     1 |    23 |    57   (2)| 00:00:01 |
|* 35 |            TABLE ACCESS FULL        | TACHE            |    10 |    90 |    28   (0)| 00:00:01 |
|  36 |            TABLE ACCESS FULL        | TACHE            |  7582 |  103K|    28   (0)| 00:00:01 |
|  37 |          BUFFER SORT                |                  |    95 |  1045 |    34   (3)| 00:00:01 |
|* 38 |           TABLE ACCESS FULL         | STAFF_EVENT      |    95 |  1045 |     5   (0)| 00:00:01 |
|* 39 |         TABLE ACCESS BY INDEX ROWID | EVENT_Q          |     1 |    19 |     1   (0)| 00:00:01 |
|* 40 |          INDEX UNIQUE SCAN          | IDX_EVENT_Q_EVENTID |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

                35 - filter("PR"."SUB_FLOW_ID"(+) IS NOT NULL)
```

由此我们可以很清楚地判断，该子查询的连接顺序是什么。首先是外面的 TACHE 表先和 ta 结果集的 TACHE 表连接，然后再和 STAFF_EVENT 连接，最后和 EVENT_Q 完成连接！由于 staff_event 这个表和 TACHE 表没有连接条件，就是没有体现 staff_event 的某某字段和

TACHE 表的某某字段关联的地方，所以这里产生了笛卡儿乘积！因此本次查询非常慢。

该如何处理呢？其实很简单，我们发现，整个语句最终只返回 42 条记录。

```
|* 30 |          INDEX UNIQUE SCAN          | PK_TACHE          |     0 |     1 |     0 |00:00:00.
|  31 |        NESTED LOOPS                 |                   |     1 |     1 |    42 |00:00:09.
|  32 |         MERGE JOIN CARTESIAN        |                   |     1 |     1 |  726K|00:00:01.
```

说明 ta 这个结果集本身返回的记录数并不多，虽然 tache 是一张大表，但是这段语句如果没有笛卡儿乘积，应该非常快！

通过简单研究发现，只要 ta 结果集内部先完成表连接，再和外部连接，就不至于产生笛卡儿乘积了。因为在 ta 结果集里，我们可以让 STAFF_EVENT b 和 EVENT_Q a 先完成连接，它们是有连接条件的，条件是 a.event_id = b.event_id。接下来这两个连接再和 ta 结果集中的 tache c 表连接，又有连接条件，条件是：a.content_id=c.tch_id。最后再和外面的 tache pr 表完成连接，连接条件是：ta.flow_id = pr.sub_flow_id(+)。如此看来，不可能有笛卡儿乘积！而如何让 Oracle 的子查询不要先自行出去和外面表连接，再回过来连接内部剩下的部分呢？

很简单，只要修改为如下语句即可，增加 rownum 部分：

```
select ta.tch_id, ta.flow_id
          from tache pr,
               (select TCH_ID, c.flow_id,rownum
                  from event_q a, staff_event b, tache c
                 where (a.event_type = '2' OR a.event_type = '1')
                   and a.event_id = b.event_id
                   and (a.flag is null or a.flag = '1')
                   and b.staff_id = 1
                   and a.content_id = c.tch_id) ta
         where ta.flow_id = pr.sub_flow_id(+)
           and pr.flow_id is null
```

脚本 4-12　执行计划 SQL 写法差异改变之 4_rownum 实体化

为了保证 rownum 的结果集是对的，oracle 不可能先出去和结果集外的表关联一部分再回到结果集内，只可能内部先关联了，因此，我们成功了！最终执行计划如下，笛卡儿乘积消失，总的 SQL 语句执行时间从 30s 变为 1s。

```
5 |* 31 |          HASH JOIN OUTER                |                   |     1 |    59 |   707   (1)| 00:00:09 |
6 |  32 |           NESTED LOOPS                  |                   |    65 |  3120 |   319   (1)| 00:00:04 |
7 |* 33 |            HASH JOIN                    |                   |   190 |  6080 |   129   (3)| 00:00:02 |
8 |* 34 |             TABLE ACCESS FULL           | STAFF_EVENT       |   190 |  2280 |    16   (0)| 00:00:01 |
9 |* 35 |             TABLE ACCESS FULL           | EVENT_Q           |  5220 |  101K|   112   (2)| 00:00:02 |
0 |  36 |            TABLE ACCESS BY INDEX ROWID  | TACHE             |     1 |    16 |     1   (0)| 00:00:01 |
1 |* 37 |             INDEX UNIQUE SCAN           | PK_TACHE          |     1 |       |     0   (0)| 00:00:01 |
2 |* 38 |           TABLE ACCESS FULL             | TACHE             |     1 |    11 |   388   (1)| 00:00:05 |
```

（5）执行计划 SQL 写法差异改变之 5_rowid 的影响

环境准备：

```
drop table t purge;
create table t as select * from dba_objects;
```

```
update t set object_id=rownum;
commit;
create  index idx_object_id on t(object_id);
set autotrace off
select rowid from t where object_id=8;
ROWID
-----
AAAaq2AALAAApOrAAH
```

语句 1：

```
update t set object_name='abc' where object_id=8;
执行计划
----------------------------------------------------------------------------------------
Plan hash value: 3890322047
----------------------------------------------------------------------------------------
| Id  | Operation          | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT   |               |     1 |    79 |     2   (0)| 00:00:01 |
|   1 |  UPDATE            | T             |       |       |            |          |
|*  2 |   INDEX RANGE SCAN| IDX_OBJECT_ID |     1 |    79 |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------
   - dynamic sampling used for this statement (level=2)
统计信息
----------------------------------------------------------------------------------------
         30  recursive calls
          2  db block gets
         75  consistent gets
          0  physical reads
        448  redo size
        680  bytes sent via SQL*Net to client
        616  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

语句 2：

```
set autotrace traceonly
set linesize 1000
--语句 2
update t set object_name='abc'  where object_id=8 and t.rowid='AAAaq2AALAAApOrAAH';
执行计划
----------------------------------------------------------------------------------------
Plan hash value: 815416494
----------------------------------------------------------------------------------------
| Id  | Operation                    | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT             |      |     1 |    91 |     1   (0)| 00:00:01 |
|   1 |  UPDATE                      | T    |       |       |            |          |
```

```
|*  2 |   TABLE ACCESS BY USER ROWID| T    |     1 |    91 |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
--------------------------------------------------------------------------------------
   2 - filter("OBJECT_ID"=8)
统计信息
--------------------------------------------------------------------------------------
          0  recursive calls
          3  db block gets
          1  consistent gets
          0  physical reads
          0  redo size
        682  bytes sent via SQL*Net to client
        650  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 4-13　执行计划 SQL 写法差异改变之 5_ rowid 的影响

说明：

语句 1 和语句 2 在某些业务场景下，显然是等价的。但是语句 2 走的是 TABLE ACCESS BY USER ROWID，而语句 1 走的是 INDEX RANGE SCAN。请注意这个 TABLE ACCESS BY USER ROWID 扫描方式，其直接根据 rowid 来访问，是最快的访问方式！

（6）执行计划 SQL 写法差异改变之 6_缓存结果影响

环境准备：

```
drop table t purge;
create table t as select * from dba_objects;
insert into t  select * from t;
commit;
```

语句 1：

```
set autotrace  on
set linesize 1000
set pagesize 2000
select  count(*) from t;
执行计划
----------------------------------------------------------------------
Plan hash value: 2966233522
----------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   589   (1)| 00:00:08 |
```

```
|   1 |  SORT AGGREGATE     |       |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T     |   315K|   589   (1)| 00:00:08 |
-------------------------------------------------------------------

   - dynamic sampling used for this statement (level=2)
统计信息
-------------------------------------------------------------------

          0  recursive calls
          0  db block gets
       2127  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

语句 2（注，这是连续执行第 2 次的输出结果）：

```
select /*+ result_cache */ count(*) from t;
执行计划
-----------------------------------------------------------------------------------------------
Plan hash value: 2966233522
-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name                       | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                            |     1 |   589   (1)| 00:00:08 |
|   1 |  RESULT CACHE       | 1n8r6qr4v9h12fhyynpun7yn4q |       |            |          |
|   2 |   SORT AGGREGATE    |                            |     1 |            |          |
|   3 |    TABLE ACCESS FULL| T                          |   315K|   589   (1)| 00:00:08 |
-----------------------------------------------------------------------------------------------
Result Cache Information (identified by operation id):
-----------------------------------------------------------------------------------------------
   1 - column-count=1; dependencies=(LJB.T); attributes=(single-row); name="select /*+
     result_cache */ count(*) from t"
Note
-----
   - dynamic sampling used for this statement (level=2)
统计信息
-----------------------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
```

```
         0  sorts (disk)
         1  rows processed
```

脚本 4-14　执行计划 SQL 写法差异改变之 6_缓存结果影响

可以看到，语句 2 出现 RESULT CACHE 关键字，这就表示该写法用了缓存结果集的特性。当然，这是特定场合应用的技术，不可滥用。

（7）执行计划 SQL 写法差异改变之 7_分区条件有无

环境准备：

```
drop table list_part_tab purge;
--注意，此分区为列表分区
create   table   list_part_tab   (id   number,deal_date   date,area_code   number,nbr
     number,building varchar2(4000))
          partition by list (area_code)
          (
          partition p_591 values  (591),
          partition p_592 values  (592),
          partition p_593 values  (593),
          partition p_594 values  (594),
          partition p_595 values  (595),
          partition p_596 values  (596),
          partition p_597 values  (597),
          partition p_598 values  (598),
          partition p_599 values  (599),
          partition p_other values  (DEFAULT)
          )
--以下是插入一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万条，如下：
insert into list_part_tab (id,deal_date,area_code,nbr,building)
      select rownum,
             to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)),'J'),
             ceil(dbms_random.value(590,599)),
             ceil(dbms_random.value(18900000001,18999999999)),
             rpad('*',400,'*')
        from dual
      connect by rownum <= 100000;
commit;

update list_part_tab set building='福州市政府' where area_code=591 and rownum=1;
commit;
```

语句 1：

```
select * from list_part_tab where building='福州市政府';
执行计划
----------------------------------------------------------------------------------------------
Plan hash value: 1711407169
----------------------------------------------------------------------------------------------
```

```
| Id | Operation          | Name          |Rows | Bytes | Cost (%CPU)| Time     |Pstart| Pstop |
-------------------------------------------------------------------------------------------
|   0| SELECT STATEMENT   |               |   86|  172K|  1829   (1)| 00:00:22 |      |       |
|   1|  PARTITION LIST ALL|               |   86|  172K|  1829   (1)| 00:00:22 |    1 |    10 |
|*  2|   TABLE ACCESS FULL| LIST_PART_TAB|   86|  172K|  1829   (1)| 00:00:22 |    1 |    10 |
-------------------------------------------------------------------------------------------
统计信息
-------------------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
       6576  consistent gets
          0  physical reads
          0  redo size
        694  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

语句 2：

```
select * from list_part_tab where building='福州市政府' and area_code=591;
执行计划
-------------------------------------------------------------------------------------------
Plan hash value: 42322831
-------------------------------------------------------------------------------------------
| Id | Operation             | Name          |Rows | Bytes |Cost (%CPU)| Time     |Pstart| Pstop|
-------------------------------------------------------------------------------------------
|   0| SELECT STATEMENT      |               |    1|  2050 |  205   (1)| 00:00:03|      |      |
|   1|  PARTITION LIST SINGLE|               |    1|  2050 |  205   (1)| 00:00:03|  KEY |   KEY|
|*  2|   TABLE ACCESS FULL   | LIST_PART_TAB|    1|  2050 |  205   (1)| 00:00:03|    1 |     1|
-------------------------------------------------------------------------------------------
统计信息
-------------------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
        695  consistent gets
          0  physical reads
          0  redo size
        694  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 4-15　执行计划 SQL 写法差异改变之 7_分区条件有无

说明：

两个语句的执行计划似乎没有差别，但是经过仔细查看我们发现，语句 1 的执行计划中 Pstart 为 1 而 Pstop 为 10，说明从第 1 个分区遍历到第 10 个分区。而第 2 个语句的执行计划中 Pstart 和 Pstop 对应的是 KEY，说明它们落在了指定的分区中。执行计划的不同性能也不言自明，逻辑读前者是 6576，后者是 695。请认真体会这句话：语句 1 和语句 2 表面上看不等价，实际从业务角度来看，是等价的，因为福州市政府肯定只位于福州市。

2. 利用设计特性

（1）执行计划利用设计特性改变之 1_分区设计影响

观察范围分区表的分区消除带来的性能优势。

```
set linesize 1000
set autotrace traceonly
set timing on
--语句 1（针对分区表）
select *
      from range_part_tab
     where deal_date >= TO_DATE('2015-08-04', 'YYYY-MM-DD')
       and deal_date <= TO_DATE('2015-08-07', 'YYYY-MM-DD');
执行计划
----------------------------------------------------------------------------------------------
| Id|Operation                |Name          |Rows |Bytes |Cost (%CPU)| Time     |Pstart| Pstop |
----------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT         |              |  958| 1917K|  171   (1)| 00:00:03|      |       |
|  1| PARTITION RANGE SINGLE|              |  958| 1917K|  171   (1)| 00:00:03|    8 |     8 |
|* 2|  TABLE ACCESS FULL      |RANGE_PART_TAB|  958| 1917K|  171   (1)| 00:00:03|    8 |     8 |
----------------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
        657  consistent gets
          0  physical reads
          0  redo size
      42053  bytes sent via SQL*Net to client
       1240  bytes received via SQL*Net from client
         77  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1136  rows processed
```

脚本 4-16　分区表的分区消除特性

比较相同的语句，普通表无法用到利用 DEAL_DATE 条件进行分区消除的情况：

```
select *
      from norm_tab
     where deal_date >= TO_DATE('2015-08-04', 'YYYY-MM-DD')
       and deal_date <= TO_DATE('2015-08-07', 'YYYY-MM-DD');
执行计划
----------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |   599 |  1199K|  1709   (1)| 00:00:21 |
|*  1 |  TABLE ACCESS FULL| NORM_TAB |   599 |  1199K|  1709   (1)| 00:00:21 |
----------------------------------------------------------------------------
统计信息
----------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
       6373  consistent gets
          0  physical reads
          0  redo size
      42222  bytes sent via SQL*Net to client
       1240  bytes received via SQL*Net from client
         77  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1137  rows processed
```

很显然，分区表的设计影响了执行计划。

（2）执行计划利用设计特性改变之 2_Cluster 类型

普通表中，排序不可避免。

```
--以下是普通方法，排序不可避免。
drop table t;
CREATE TABLE t
(  cust_id        number,
   order_dt       timestamp ,
   order_number   number,
   username       varchar2(30),
   ship_addr      number,
   bill_addr      number,
   invoice_num    number
);

set linesize 1000
set pagesize 2000
set autotrace traceonly

variable x number
--以下是利用有序散列聚簇表的方法，发现排序被避免
--语句 1（语句 1 和接下来的语句 2 一样，但是两个语句的 t 表的类型不同）
select cust_id, order_dt, order_number
```

```
  from t
 where cust_id = :x
 order by order_dt;

执行计划
----------------------------------------------------------------------------
Plan hash value: 961378228
----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    39 |     3  (34)| 00:00:01 |
|   1 |  SORT ORDER BY     |      |     1 |    39 |     3  (34)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| T    |     1 |    39 |     2   (0)| 00:00:01 |
----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------------------------------
   2 - filter("CUST_ID"=TO_NUMBER(:X))
Note
-----
   - dynamic sampling used for this statement (level=2)
统计信息
---------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
        416  bytes sent via SQL*Net to client
        404  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

以下是利用有序散列聚簇表的方法，发现排序被避免：

```
set autotrace off
drop table t;
drop cluster shc;

CREATE CLUSTER shc
(
   cust_id     NUMBER,
   order_dt    timestamp SORT
)
HASHKEYS 10000
HASH IS cust_id
SIZE  8192
/
CREATE TABLE t
(  cust_id       number,
   order_dt      timestamp SORT,
```

```
   order_number  number,
   username      varchar2(30),
   ship_addr     number,
   bill_addr     number,
   invoice_num   number
)
CLUSTER shc ( cust_id, order_dt )
/

---开始执行分析
set autotrace traceonly
variable x number
--以下是利用有序散列聚簇表的方法，发现排序被避免
--语句 2（语句 2 和上面执行过的语句 1 一样，但是两个语句的 t 表类型不同）
select cust_id, order_dt, order_number
  from t
 where cust_id = :x
 order by order_dt;

执行计划
----------------------------------------------------------
Plan hash value: 2719348108
----------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
----------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    39 |     0   (0)|
|*  1 |  TABLE ACCESS HASH| T    |     1 |    39 |            |
----------------------------------------------------------
Predicate Information (identified by operation id):
----------------------------------------------------------
   1 - access("CUST_ID"=TO_NUMBER(:X))
Note
-----
   - dynamic sampling used for this statement (level=2)
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
        416  bytes sent via SQL*Net to client
        404  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

脚本 4-17　有序散列聚簇表消除排序

（3）执行计划利用设计特性改变之 3_Iot 表类型

环境准备：

```
set autotrace off
drop table heap_addresses purge;
drop table iot_addresses purge;
create table heap_addresses
   (empno    number(10),
    addr_type varchar2(10),
    street    varchar2(10),
    city      varchar2(10),
    state     varchar2(2),
    zip       number,
    primary key (empno)
   )
/

create table iot_addresses
   (empno    number(10),
    addr_type varchar2(10),
    street    varchar2(10),
    city      varchar2(10),
    state     varchar2(2),
    zip       number,
   primary key (empno)
   )
   organization index
/
insert into heap_addresses
   select object_id,'WORK','123street','washington','DC',20123
   from all_objects;
insert into iot_addresses
    select object_id,'WORK','123street','washington','DC',20123
    from all_objects;
commit;
```

接下来分别比较索引组织表和普通表的查询性能，首先是普通表，语句 1 如下：

```
set linesize 1000
set autotrace traceonly
--语句 1(针对普通表)
select * from heap_addresses where empno=22;
执行计划
----------------------------------------------------------------------------------------------
| Id  | Operation                   | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                |     1 |    50 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| HEAP_ADDRESSES |     1 |    50 |     1   (0)| 00:00:01 |
```

```
|*  2 |   INDEX UNIQUE SCAN         | SYS_C0013751   |     1 |       |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------
统计信息
---------------------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        659  bytes sent via SQL*Net to client
        405  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

接下来针对索引组织表，语句 2 如下：

```
select * from iot_addresses where empno=22;
执行计划
----------------------------------------------------------------------------------------------
| Id  | Operation         | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                     |     1 |    50 |     1   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN| SYS_IOT_TOP_104441  |     1 |    50 |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
        751  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 4-18　IOT 表消除回表

观察语句 1 和语句 2 的执行计划的差异，语句 1 的执行计划中有 TABLE ACCESS BY INDEX ROWID 部分，而语句 2 却没有。因为在索引组织表中，表就是索引，索引就是表，没有回表的概念。

（4）执行计划利用设计特性改变之 4_物化视图影响

语句 1，普通写法未用到物化视图的特性，如下：

```
drop materialized view MV_COUNT_T;
drop table t purge;
```

```
create table t as  select * from dba_objects;
set autotrace traceonly
set linesize 1000

--语句 1（未建过物化视图）
select COUNT(*) FROM T;
执行计划
----------------------------------------------------------------------
Plan hash value: 2966233522
----------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 80592 |   292   (1)| 00:00:04 |
----------------------------------------------------------------------
统计信息
----------------------------------------------------------------------
          0  recursive calls
          0  db block gets
       1047  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

语句 2，用到物化视图的特性，如下：

```
create  materialized view  mv_count_t
                  build immediate
                  refresh on commit
                  enable query rewrite
                  as
                  select count(*) FROM T;

set autotrace traceonly
set linesize 1000

--开始执行语句 2（语句 2 本身和语句 1 没区别，但是建过物化视图）
select COUNT(*) FROM T;
执行计划
----------------------------------------------------------------------------------------------
Plan hash value: 3655891017
----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |     1 |    13 |     3   (0)| 00:00:01 |
```

```
|   1 |  MAT_VIEW REWRITE ACCESS FULL| MV_COUNT_T |     1 |    13 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------
统计信息
---------------------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        429  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 4-19　物化视图改变了访问的表

说明：

观察语句 1 和语句 2 的执行计划的差异，重点是执行计划的 NAME 部分的输出，一个是 T，一个是 MV_COUNT_T。还可以观察 Operation 部分，一个有 SORT AGGREGATE，另一个没有。

最后，再观察比较输出的统计信息的逻辑读大小。

（5）执行计划利用设计特性改变之 5_并行度影响

语句 1：

```
drop table t;
create table t as select * from dba_objects;
set linesize 1000
set pagesize 2000
set autotrace off
ALTER SESSION SET statistics_level = all;
--语句 1
select count(*) from t;
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------------------
SQL_ID  cyzznbykb509s, child number 0
---------------------------------------------------------------------------------------------
select count(*) from t

Plan hash value: 2966233522
---------------------+-----------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
---------------------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT   |      |      1 |        |      1 |00:00:00.08 |    1047 |    692 |
|   1 |  SORT AGGREGATE    |      |      1 |      1 |      1 |00:00:00.08 |    1047 |    692 |
|   2 |   TABLE ACCESS FULL| T    |      1 |  80575 |  73118 |00:00:00.07 |    1047 |    692 |
--------------------------------------------------------------------------------------------------

Note
-----
   - dynamic sampling used for this statement (level=2)
已选择 18 行。
```

语句 2：

```
select /*+parallel(t,4)*/ count(*) from t;
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------
SQL_ID  bwpwc1h0h768t, child number 0
--------------------------------------------------------------------------------------------------
select /*+parallel(t,4)*/ count(*) from t
Plan hash value: 3126468333
--------------------------------------------------------------------------------------------------
| Id  | Operation              | Name     | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
--------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |      1 |        |      1 |00:00:00.09 |       5 |
|   1 |  SORT AGGREGATE        |          |      1 |      1 |      1 |00:00:00.09 |       5 |
|   2 |   PX COORDINATOR       |          |      1 |        |      4 |00:00:00.09 |       5 |
|   3 |    PX SEND QC (RANDOM) | :TQ10000 |      0 |      1 |      0 |00:00:00.01 |       0 |
|   4 |     SORT AGGREGATE     |          |      0 |      1 |      0 |00:00:00.01 |       0 |
|   5 |      PX BLOCK ITERATOR |          |      0 |  80575 |      0 |00:00:00.01 |       0 |
|*  6 |       TABLE ACCESS FULL| T        |      0 |  80575 |      0 |00:00:00.01 |       0 |
--------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
--------------------------------------------------------------------------------------------------
   6 - access(:Z>=:Z AND :Z<=:Z)
Note
-----
   - dynamic sampling used for this statement (level=2)
已选择 27 行。
```

语句 3（注意这时表的属性被设置成并行度为 4）：

```
alter table t parallel 4;
select count(*) from t;
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------
SQL_ID  cyzznbykb509s, child number 0
--------------------------------------------------------------------------------------------------
select count(*) from t
Plan hash value: 3126468333
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name     | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |      1 |        |      1 |00:00:00.03 |       5 |
|   1 |  SORT AGGREGATE        |          |      1 |      1 |      1 |00:00:00.03 |       5 |
|   2 |   PX COORDINATOR       |          |      1 |        |      4 |00:00:00.03 |       5 |
|   3 |    PX SEND QC (RANDOM) | :TQ10000 |      0 |      1 |      0 |00:00:00.01 |       0 |
|   4 |     SORT AGGREGATE     |          |      0 |      1 |      0 |00:00:00.01 |       0 |
|   5 |      PX BLOCK ITERATOR |          |      0 |  80575 |      0 |00:00:00.01 |       0 |
|*  6 |       TABLE ACCESS FULL| T        |      0 |  80575 |      0 |00:00:00.01 |       0 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------------------------------------------------------
   6 - access(:Z>=:Z AND :Z<=:Z)
Note
----
   - dynamic sampling used for this statement (level=2)
已选择 27 行。
```

脚本 4-20　表的属性设置为并行，执行计划变化了

语句 4（注意这个 no_parallel 的 hint，该 hint 可以消除并行特性）：

```
select /*+no_parallel*/ count(*) from t;
SELECT * FROM table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------------
select /*+no_parallel*/ count(*) from t
Plan hash value: 2966233522
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |      1 |00:00:00.03 |    1047 |
|   1 |  SORT AGGREGATE    |      |      1 |      1 |      1 |00:00:00.03 |    1047 |
|   2 |   TABLE ACCESS FULL| T    |      1 |  68386 |  73109 |00:00:00.02 |    1047 |
-------------------------------------------------------------------------------------------
Note
----
   - dynamic sampling used for this statement (level=2)
```

这里语句 1 和语句 4 没用到并行，语句 2 和语句 3 用到并行。其中语句 3 是表的属性本身被设置为并行，而语句 4 是虽然表的属性为并行，但是用 Hint 强制避免走并行，请好好体会这些语句，并认真观察并行对应的执行计划 PX COORDINATOR。

（6）执行计划利用设计特性改变之 6_列空值影响

语句 1（表的索引列的属性没有被限制为非空）：

```
drop table t purge;
create table t as select * from dba_objects where object_id is not null;
create index idx_object_id on t(object_id);
set linesize 1000
```

```
set pagesize 2000
set autotrace traceonly
--语句 1
select count(*) from t;
执行计划
----------------------------------------------------------------------
Plan hash value: 2966233522
----------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 67030 |   292   (1)| 00:00:04 |
----------------------------------------------------------------------
统计信息
----------------------------------------------------------------------
          0  recursive calls
          0  db block gets
       1047  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

语句 2（表的索引列的属性被限制为非空）：

```
alter table t modify object_id not null;
--语句 2（语句 2 和语句 1 没差别，但是 t 表的 object_id 列的属性被设置为不允许空了）
select count(*) from t;
执行计划
----------------------------------------------------------------------------
Plan hash value: 1131838604
----------------------------------------------------------------------------
| Id  | Operation             | Name          | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               |     1 |    49   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |               |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX_OBJECT_ID | 67030 |    49   (0)| 00:00:01 |
----------------------------------------------------------------------------
统计信息
----------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
        170  consistent gets
          0  physical reads
          0  redo size
```

```
       425  bytes sent via SQL*Net to client
       415  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

脚本 4-21　索引列是否允许为空对执行计划的影响

观察语句 1 和语句 2 的执行计划的差异，语句 1 的 object_id 列允许为空，这导致走全表扫描。当设定了 object_id 列不允许为空后，一模一样的语句 2 开始走 INDEX FAST FULL SCAN 了。

（7）执行计划利用设计特性改变之 7_主外键影响

场景 1，普通语句的写法：

```
drop table t1 cascade constraints purge;
drop table t2 cascade constraints  purge;
create table t1 as select * from dba_objects;
create table t2 as select * from dba_objects where rownum<=10000;
update t1 set object_id=rownum ;
update t2 set object_id=rownum ;
commit;

create or replace view v_t1_join_t2
as select t2.object_id,t2.object_name,t1.object_type,t1.owner from t1,t2
where t1.object_id=t2.object_id;

set linesize 1000
set pagesize 2000
set autotrace traceonly

--语句 1（此时还没有对这两个表建主外键）
select object_id,object_name from v_t1_join_t2;
执行计划
----------------------------------------------------------------------------
Plan hash value: 2959412835
----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 12187 |  1094K|   333   (1)| 00:00:04 |
|*  1 |  HASH JOIN         |      | 12187 |  1094K|   333   (1)| 00:00:04 |
|   2 |   TABLE ACCESS FULL| T2   | 12186 |   940K|    40   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T1   | 64114 |   813K|   292   (1)| 00:00:04 |
----------------------------------------------------------------------------
统计信息
----------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
```

```
      1841  consistent gets
         0  physical reads
         0  redo size
    319671  bytes sent via SQL*Net to client
      7741  bytes received via SQL*Net from client
       668  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
     10000  rows processed
```

场景 2，接下来，为 T1 表增加一个主键，继续看该语句，看看语句执行计划有啥变化：

```
alter table T1
add constraint pk_object_id primary key (OBJECT_ID);
--场景 2（此时已经对这 T1 表建了主键）
select object_id,object_name from v_t1_join_t2;
执行计划
----------------------------------------------------------------------------------------
Plan hash value: 1632777847
----------------------------------------------------------------------------------------
| Id  | Operation          | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |              | 12187 |  1094K|    41   (3)| 00:00:01 |
|   1 |  NESTED LOOPS      |              | 12187 |  1094K|    41   (3)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T2           | 12186 |   940K|    40   (0)| 00:00:01 |
|*  3 |   INDEX UNIQUE SCAN| PK_OBJECT_ID |     1 |    13 |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------------------------------------
         0  recursive calls
         0  db block gets
      1617  consistent gets
         0  physical reads
         0  redo size
    319671  bytes sent via SQL*Net to client
      7741  bytes received via SQL*Net from client
       668  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
     10000  rows processed
```

场景 3，继续为 T1 表加主键后，再为 T2 表增加一个指向 T1 表的外键：

```
alter table T2
add constraint fk_objecdt_id foreign key (OBJECT_ID) references t1 (OBJECT_ID);
--场景 3（此时已经为 T2 表增加一个指向 T1 表的外键）
select object_id,object_name from v_t1_join_t2;
执行计划
----------------------------------------------------------------------------------------
Plan hash value: 1513984157
----------------------------------------------------------------------------
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 12186 |   940K|    40   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T2   | 12186 |   940K|    40   (0)| 00:00:01 |
-------------------------------------------------------------------------------
统计信息
-------------------------------------------------------------------------------
          0  recursive calls
          0  db block gets
        794  consistent gets
          0  physical reads
          0  redo size
     319671  bytes sent via SQL*Net to client
       7741  bytes received via SQL*Net from client
        668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      10000  rows processed
```

脚本 4-22　主外键对执行计划的影响

认真观察执行计划大家会发现，同样的语句在场景 1、2、3 下的执行计划各不相同。其中场景 3 最为神奇的是，表的访问居然只有 T2 表，而没有 T1 表。这是为啥呢？因为主外键确保了只访问 T2 表和访问两表关联的视图的效果是一样的，所以 Oracle 聪明地选择了只访问 T2 表。

3. 其他相关手段

（1）执行计划改变之其他相关手段 1_set_table_stats

如下语句运行的效率比较低下，请仔细看其运行：

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n<=20;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------
        -
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |     20 |00:00:01.16 |   10003 |
|*  1 |  HASH JOIN         |      |      1 |     20 |     20 |00:00:01.16 |   10003 |
|   2 |   TABLE ACCESS FULL| T2   |      1 |     10 |   1000K|00:00:00.21 |    9985 |
|*  3 |   TABLE ACCESS FULL| T1   |      1 |     20 |     20 |00:00:00.01 |      18 |
-------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
-------------------------------------------------------------------------------------
```

```
   1 - access("T1"."ID"="T2"."T1_ID")
   3 - filter("T1"."N"<=20)
Note
-----
   - cardinality feedback used for this statement
已选择 25 行。
```

大家注意观察就会发现，T2 表的记录是 1000K，显然比 T1 的结果集要大得多，这里驱动顺序显然错了。应该是统计信息出现问题了，预测 10 条实际是 1000K 条。这时候我们可以通过如下的方式来重新收集统计信息：

```
exec dbms_stats.gather_table_stats(ownname => user,tabname => 'T1',estimate_percent
      => 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;
exec dbms_stats.gather_table_stats(ownname => user,tabname => 'T2',estimate_percent
      => 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;
```

不过遗憾的是，这个动作恐怕在业务高峰期影响生产，所以需要等到业务低峰的时候去完成。当然我们还可以用 hint 来强制改变驱动顺序，不过需要修改代码，也不合适。这时，有一种思路就出来了，就是用 set_table_stats 的方式来告诉 Oracle 表的大小情况，如下：

```
EXEC  dbms_stats.SET_table_stats(user, 'T1', numrows => 1000  ,numblks => 10);
EXEC  dbms_stats.SET_table_stats(user, 'T2', numrows => 2000000  ,numblks => 100000);
set linesize 1000
set pagesize 2000
alter session set statistics_level=all ;
--有的版本数据库需要执行 alter system flush shared_pool;才能生效，一般情况下不需要。
--alter system flush shared_pool;
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n<=20;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |     20 |00:00:00.53 |    9879 |
|*  1 |  HASH JOIN         |      |      1 |    100K|     20 |00:00:00.53 |    9879 |
|*  2 |   TABLE ACCESS FULL| T1   |      1 |     50 |     20 |00:00:00.01 |      16 |
|   3 |   TABLE ACCESS FULL| T2   |      1 |   2000K|   1000K|00:00:00.18 |    9863 |
------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
------------------------------------------------------------------------------------
   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T1"."N"<=20)
已选择 21 行。
```

脚本 4-23　set_table_stats 改变执行计划

好神奇，我们在几乎没有付出什么代价的情况下，改变了 Oracle 的执行计划，让该 SQL

性能得以提升。

（2）执行计划改变之其他相关手段 2_预估索引效果

这是一个很有意思的特性，请看如下代码，由于没有索引，走的是全表扫描：

```
drop table t purge;
create table t as select * from dba_objects;
set linesize 300
explain plan for select * from t where object_id=1;
select * from table(dbms_xplan.display());
SQL> explain plan for select * from t where object_id=1;

已解释。

SQL> select * from table(dbms_xplan.display());

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1601196873
--------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    19 |  3933 |   307   (1)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| T    |    19 |  3933 |   307   (1)| 00:00:04 |
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
--------------------------------------------------------------------------------
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
   1 - filter("OBJECT_ID"=1)
Note
-----
   - dynamic sampling used for this statement
已选择 17 行。
```

我们会想，要不建一个索引看看？问题来了，建索引后有几个问题，比如索引建完后，Oracle 会选择使用这个索引吗？选择后代价会降低吗？建索引本身是一件很辛苦的事，要是建完后效果不好，怎么办？其实不用担心，有一个思路叫虚拟索引，我们来测试一下它的效果，如下：

```
drop table t purge;
create table t as select * from dba_objects;
--创建虚拟索引，首先要将_use_nosegment_indexes 的隐含参数设置为 true
alter session set "_use_nosegment_indexes"=true;
--虚拟索引的创建语法比较简单，实际上就是普通索引语法后面加一个 nosegment 关键字
create index ix_t_id on t(object_id) nosegment;
set linesize 1000
explain plan for select * from t where object_id=1;
select * from table(dbms_xplan.display());
```

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------
Plan hash value: 206018885
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |    12 |  2484 |     5   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T       |    12 |  2484 |     5   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IX_T_ID |   313 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------
   2 - access("OBJECT_ID"=1)
Note
   - dynamic sampling used for this statement (level=2)
```

脚本 4-24　虚拟索引预估有无必要建索引

发现果然走索引，而且代价下降了不少，恩，很好用。可以建！

4.2.3　执行计划的固定

场景 1，首先我们看一个简单的 SQL，由于表的索引列没有被设置为非空，因此执行计划肯定是走不了 INDEX FAST FULL SCAN 的，而只能走 TABLE ACCESS FULL，如下：

```
--首先建测试表
--环境构造
drop table t purge;
create table t as select * from dba_objects where object_id is not null;
create index idx_object_id on t(object_id);
set linesize 1000
set pagesize 2000
explain plan for select count(*) from t ;
select * from table(dbms_xplan.display());
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------
Plan hash value: 2966233522
--------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 53129 |   292   (1)| 00:00:04 |
--------------------------------------------------------------------
Note
----
   - dynamic sampling used for this statement (level=2)
```

```
已选择 13 行。
```

接下来切入正题，开始最关键的命令，建立大纲，也就是 OUTLINE！

```
create or replace outline myoutline
    for category mycategory
    on
 select count(*) from t  ;
select category,SQL_text,signature from user_outlines where name='MYOUTLINE' ;
```

场景 2，如果我们把 object_id 这个索引列的属性设置为非空，那 select count(*) from t 语句必然走 INDEX FAST FULL SCAN，如下：

```
alter table t modify object_id not null;
explain plan for select count(*) from t ;
select * from table(dbms_xplan.display());
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1131838604
--------------------------------------------------------------------------------
| Id  | Operation             | Name          | Rows  | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               |     1 |    49   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |               |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX_OBJECT_ID | 53129 |    49   (0)| 00:00:01 |
--------------------------------------------------------------------------------
   - dynamic sampling used for this statement (level=2)
已选择 13 行。
```

不过大纲的作用是固定执行计划，因此假如我们沿用原来的大纲，这个语句就不会走索引，如下：

```
alter session set use_stored_outlines =mycategory;
explain plan for select count(*) from t ;
select * from table(dbms_xplan.display());
PLAN_TABLE_OUTPUT
------------------------------------------------------------------
Plan hash value: 2966233522
------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 87235 |   292   (1)| 00:00:04 |
------------------------------------------------------------------
Note
----
   - outline "MYOUTLINE" used for this statement

已选择 13 行。
```

脚本 4-25　执行计划的大纲固定

4.3 本章习题、总结与延伸

习题 1：说说你对 hint 的认识。

习题 2：说说你对 rownum 实体化视图优化方法表连接的认识。

习题 3：简要谈谈你对非 hint 方式影响执行计划的认识。

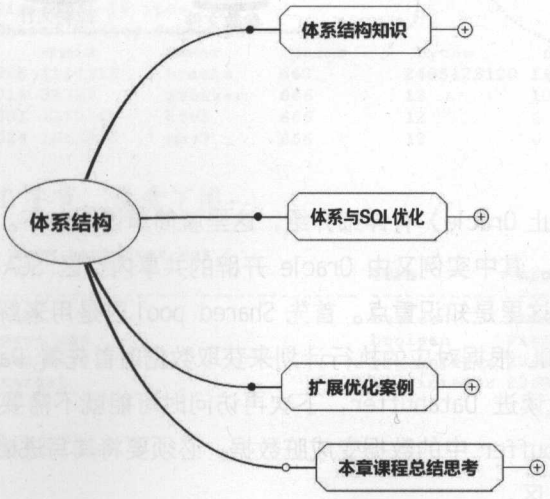
习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

第 5 章　且慢，感受体系结构让 SQL 飞

原来学习体系结构是有用的

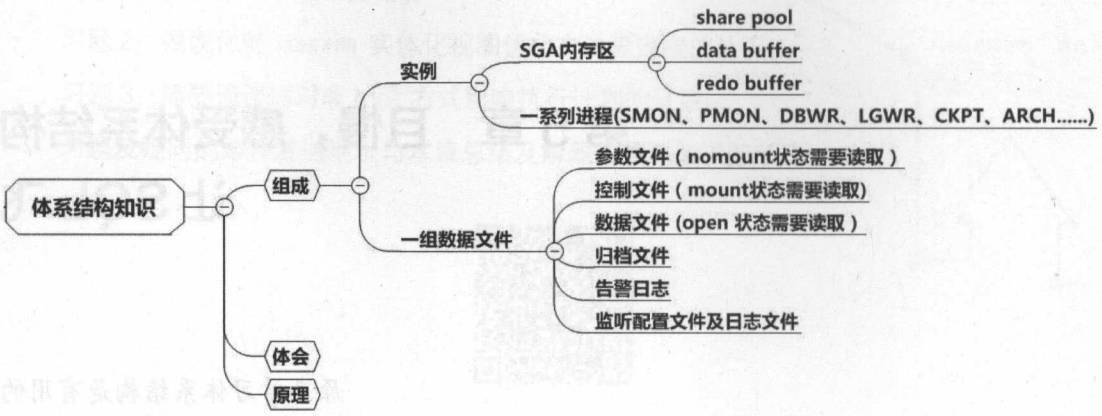
前言故事 5 中，老丁没有改写 SQL，只是加了一个索引，然后 SQL 就变快了，这是看起来最省事的方法了。其实我们在接下来的 7 章里大部分内容都是教大家如何不改写 SQL 完成 SQL 优化。不过这里要注意几点：1. 你必须深刻理解数据库的原理；2.你能把握业务场景；3.接下来的这些章节是教大家如何不改写 SQL 进行优化，但也不是全都不用改写，比如绑定变量、批量提交等还是必须要改造 SQL 的。

由于《收获，不止 Oracle》已经介绍得比较详细了，本章会较为简要地给大家介绍一下体系结构知识，然后描述体系结构和 SQL 优化的关系。最后通过系列扩展的相关优化案例来拓宽我们的视野，从而使我们更深入地了解体系结构的原理。最后是思考回顾。本章总体学习思路如下图所示：



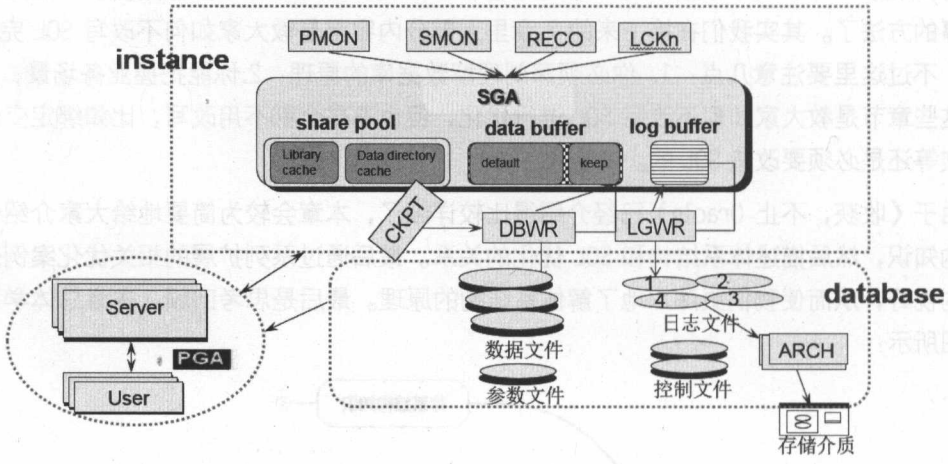
5.1 体系结构知识

如下图所示：



5.1.1 组成

组成如下图所示：



5.1.2 原理

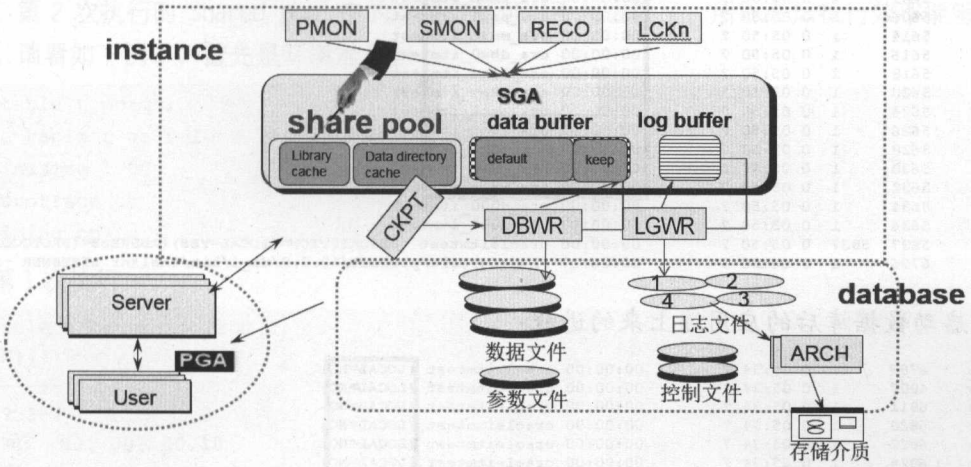
考虑到《收获，不止 Oracle》有详细介绍，这里就简单说明一下。Oracle 体系结构由实例和数据文件两部分组成。其中实例又由 Oracle 开辟的共享内存区 SGA 和一组进程组成。其中 SGA 被划分成 3 部分，这里是知识重点。首先 Shared pool 区是用来解析 SQL 并保存相关执行计划的区域，接下来 SQL 根据对应的执行计划来获取数据时首先看 Databuffer 中有没有所需的数据，没有则从磁盘读进 Databuffer，下次再访问时可能就不需要从磁盘读取了。当更新 SQL 语句出现时，Databuffer 中的数据变成脏数据，必须要将其写进磁盘。而为了保护这些数据，才有了 Log buffer 区。

更多的细节就不再累述了，详情阅读《收获，不止 Oracle》。

5.1.3　体会

1. 体会体系结构中的 SGA

Oracle 的体系结构的 SGA 部分，如下图所示：



未启动数据库前的 SGA 分配情况：

```
[oracle@itmapp3 ~]$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00020014 32769      gserver    666        12         10
0x00028081 131076     kfv3       666        12         6
0x00024024 1015828    nmv3       666        12         6
```

启动数据库后的 SGA 分配情况：

```
[oracle@itmapp3 ~]$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch
0x66a02988 1114112    oracle     660        2485125120 14
0x00020014 32769      gserver    666        12         10
0x00028081 131076     kfv3       666        12         6
0x00024024 1015828    nmv3       666        12         6
```

为啥是 2485125120 字节，请看下图：

```
SQL> show parameter sga
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------
lock_sga                             boolean     FALSE
pre_page_sga                         boolean     FALSE
sga_max_size                         big integer 2368M
sga_target                           big integer 2368M
```

原来 SGA 开辟的就是这么大。

2. 体会体系结构中的进程

未启动数据库前的 oracle 进程情况：

```
[oracle@itmapp3 ~]$ ps -ef |grep ora
oracle    6796     1  0 Oct28 ?        00:06:16 /home/oracle/product/10.2.0/db_1/bin/tnslsnr LISTENER -inherit
```

启动数据库后的 oracle 本地进程：

```
[oracle@itmapp3 ~]$ ps -ef |grep ora
oracle    5601     1  0 05:50 ?        00:00:00 ora_pmon_itmtest
oracle    5606     1  0 05:50 ?        00:00:00 ora_psp0_itmtest
oracle    5614     1  0 05:50 ?        00:00:00 ora_mman_itmtest
oracle    5616     1  0 05:50 ?        00:00:00 ora_dbw0_itmtest
oracle    5618     1  0 05:50 ?        00:00:00 ora_lgwr_itmtest
oracle    5620     1  0 05:50 ?        00:00:00 ora_ckpt_itmtest
oracle    5624     1  0 05:50 ?        00:00:00 ora_smon_itmtest
oracle    5626     1  0 05:50 ?        00:00:00 ora_reco_itmtest
oracle    5628     1  0 05:50 ?        00:00:00 ora_cjq0_itmtest
oracle    5630     1  0 05:50 ?        00:00:00 ora_mmon_itmtest
oracle    5632     1  0 05:50 ?        00:00:00 ora_mmnl_itmtest
oracle    5634     1  0 05:50 ?        00:00:00 ora_d000_itmtest
oracle    5636     1  0 05:50 ?        00:00:00 ora_s000_itmtest
oracle    5637  5037  0 05:50 ?        00:00:00 oracleitmtest (DESCRIPTION=(LOCAL=YES)(ADDRESS=(PROTOCOL=beq)))
oracle    6796     1  0 Oct28 ?        00:06:17 /home/oracle/product/10.2.0/db_1/bin/tnslsnr LISTENER -inherit
```

启动数据库后的应用连上来的进程：

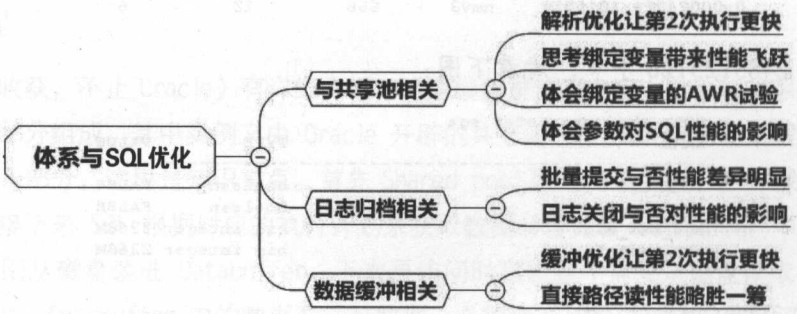
```
oracle    6799     1  0 05:34 ?        00:00:00 oracleitmtest (LOCAL=NO)
oracle    6807     1  0 05:34 ?        00:00:00 oracleitmtest (LOCAL=NO)
oracle    6811     1  0 05:34 ?        00:00:00 oracleitmtest (LOCAL=NO)
oracle    6820     1  0 05:34 ?        00:00:00 oracleitmtest (LOCAL=NO)
oracle    6822     1  0 05:34 ?        00:00:00 oracleitmtest (LOCAL=NO)
oracle    6824     1  0 05:34 ?        00:00:00 oracleitmtest (LOCAL=NO)
oracle    8801     1  0 Nov04 ?        00:00:00 oracleitmtest (LOCAL=NO)
oracle    8803     1  0 Nov04 ?        00:00:00 oracleitmtest (LOCAL=NO)
```

启动数据库后查看 Oracle 应用连接数和总的连接数：

```
[oracle@itmapp3 ~]$ ps -ef |grep LOCAL=NO|wc -l
151
[oracle@itmapp3 ~]$ ps -ef |grep ora|wc -l
172
```

5.2 体系与 SQL 优化

接下来我们从共享池、日志及数据缓冲三个方面来描述体系结构与 SQL 优化的关系，总体思路如下图所示：



5.2.1　与共享池相关

首先我们看看体系结构 SGA 的三大组成之一的共享池 Shared pool，这是 SQL 语句执行中最先访问到的内存部件。

1. 解析优化让第 2 次执行更快

由于 SQL 第 1 次执行时已经完成了语法、语义的判断，完成了解析，保存了优选过的执行计划，第 2 次执行时 Shared pool 中的这些事就可以不用去做了，所以必然执行效率能够提升不少。请看如下例子，首先是环境准备。

```
drop table t purge;
create table t as select  * from dba_objects;
set linesize 1000
set autotrace on
set timing on
```

第 1 次执行：

```
SQL> select count(*) from t;
  COUNT(*)
----------
     72884
已用时间:  00: 00: 00.10
执行计划
--------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   291   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 60918 |   291   (1)| 00:00:04 |
-----------------------------------------------------------------------
统计信息
--------
         28  recursive calls
          0  db block gets
       1103  consistent gets
       1038  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

```
SQL> --第 2 次执行
SQL> --该命令只是为了先不考虑 2 次执行物理读减少带来的效果，只考虑减少解析的优化效果
SQL> alter system flush buffer_cache;
系统已更改。
已用时间:  00: 00: 00.03
```

```
SQL> select count(*) from t;
  COUNT(*)
----------
     72884
已用时间:  00: 00: 00.07
执行计划
--------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   291   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 60918 |   291   (1)| 00:00:04 |
-----------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1043  consistent gets
       1039  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 5-1　解析优化让第 2 次执行更快

结论：

第 1 次执行时间是 00: 00: 00.10，recursive calls 次数为 28；第 2 次执行时间是 00: 00: 00.07，recursive calls 次数为 0。很显然，解析的优化让第 2 次执行更快。

2. 思考绑定变量带来的性能飞跃

现实生活中绝大部分用户都有输入自己手机相关信息来访问某系统的经历，如 SQL 简单构造：select * from t where nbr=18900000001。可以预见到，系统很快就会出现新的 nbr 取值的 SQL，如 select * from t where nbr=18900000002。用户一多，系统中将会出现大量高度相似仅 nbr 不同的 SQL，这些不同的 SQL 实际上执行计划应该都是一样的，但是在 Shared pool 里都要挨个解析，因而做了很多的无用功，由于存储在共享池中，也耗费宝贵资源。

如果用绑定变量，这些 SQL 都变成 select * from t where nbr=:x，这下形成一条 SQL，减少了系统大量的解析时间，也节省了共享池资源，性能得到大幅提升。下面我们看一组例子，如下：

未使用绑定变量：

```
SQL> begin
```

```
  2      for i in 1 .. 100000
  3      loop
  4          execute immediate
  5          'insert into t values ( '||i||')';
  6      end loop;
  7      commit;
  8  end;
  9  /

PL/SQL 过程已成功完成。

已用时间:  00: 00: 43.50
```

脚本 5-2　未使用绑定变量脚本

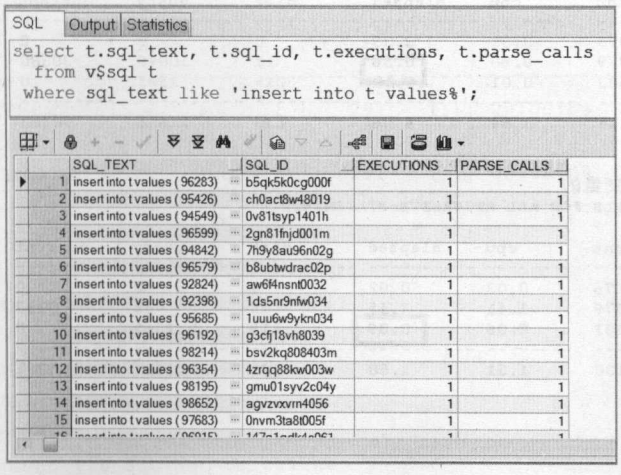
使用绑定变量：

```
SQL> begin
  2      for i in 1 .. 100000
  3      loop
  4          execute immediate
  5          'insert into t values ( :x )' using i;
  6      end loop;
  7          commit;
  8  end;
  9  /

PL/SQL 过程已成功完成。

已用时间:  00: 00: 04.77
```

脚本 5-3　使用绑定变量脚本

可以看出性能差异非常明显，未使用绑定变量是 43s，使用后仅 4s。

3. 体会硬解析次数和执行次数



	SQL_TEXT	SQL_ID	EXECUTIONS	PARSE_CALLS
1	insert into t values (96283)	b5qk5k0cg000f	1	1
2	insert into t values (95426)	ch0act8w48019	1	1
3	insert into t values (94549)	0v81tsyp1401h	1	1
4	insert into t values (96599)	2gn81fnjd001m	1	1
5	insert into t values (94842)	7h9y8au96n02g	1	1
6	insert into t values (96579)	b8ubtwdrac02p	1	1
7	insert into t values (92824)	aw6f4nsnt0032	1	1
8	insert into t values (92398)	1ds5nr9nfw034	1	1
9	insert into t values (95685)	1uuu6w9ykn034	1	1
10	insert into t values (96192)	g3cfj18vh8039	1	1
11	insert into t values (98214)	bsv2kq808403m	1	1
12	insert into t values (96354)	4zrqq88kw003w	1	1
13	insert into t values (98195)	gmu01syv2c04y	1	1
14	insert into t values (98652)	agvzvxvrn4056	1	1
15	insert into t values (97683)	0nvm3ta8t005f	1	1

SQL | Output | Statistics

```
select t.sql_text, t.sql_id, t.executions, t.parse_calls
  from v$sql t
 where sql_text like 'insert into t values (:x)%';
```

	SQL_TEXT	SQL_ID	EXECUTIONS	PARSE_CALLS
1	insert into t values (x)	bbz6pdq577unj	100000	1

脚本 5-4　体会硬解析次数和执行次数

4. 体会绑定变量的 AWR 试验

	Per Second	Per Transaction	Per Exec	Per Call
DB Time(s):	0.9	3.2	0.00	3.17
DB CPU(s):	0.8	3.1	0.00	3.14
Redo size:	436,015.3	1,631,772.8		
Logical reads:	7,943.7	29,729.2		
Block changes:	3,625.5	13,568.4		
Physical reads:	8.0	30.0		
Physical writes:	0.3	1.2		
User calls:	0.3	1.0		
Parses:	1,823.4	6,824.0		
Hard parses:	1,790.5	6,701.0		
W/A MB processed:	0.5	1.9		
Logons:	0.1	0.2		
Executes:	1,931.9	7,229.9		
Rollbacks:	0.0	0.0		
Transactions:	0.3			

Buffer Nowait %:	100.00	Redo NoWait %:	100.00
Buffer Hit %:	99.90	In-memory Sort %:	100.00
Library Hit %:	62.91	Soft Parse %:	1.80
Execute to Parse %:	5.61	Latch Hit %:	100.00
Parse CPU to Parse Elapsd %:	98.37	% Non-Parse CPU:	25.58

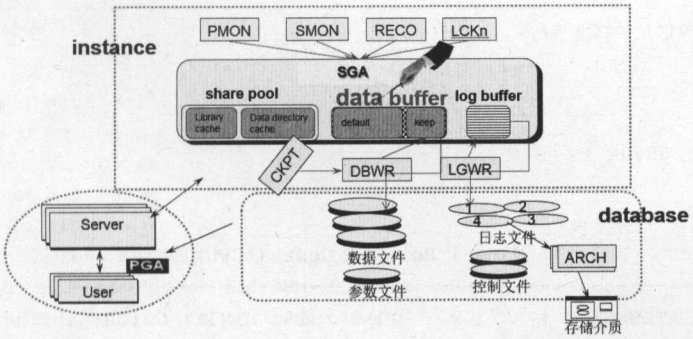
5. 思考绑定变量的 TRACE 试验

```
--未使用绑定变量的

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    10075      3.32       3.39          0          0          0           0
Execute  10379      0.60       0.58         11      10049      30380       10006
Fetch      683      0.01       0.10        219       1565          0        1418
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    21137      3.94       4.06        230      11614      30380       11424

---使用绑定变量的
OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       76      0.03       0.02          0          0          0           0
Execute  10379      1.41       1.45         12         52      10397       10006
Fetch      683      0.06       0.09        219       1565          0        1418
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    11138      1.51       1.58        231       1617      10397       11424
```

6. 注意静态 SQL 自动绑定变量

```
SQL> set linesize 1000
SQL> column sql_text format a50
SQL> select t.sql_text, t.sql_id, t.executions, t.parse_calls
  2    from v$sql t
  3   where lower(sql_text) like 'insert into t values%';

SQL_TEXT                                           SQL_ID        EXECUTIONS PARSE_CALLS
-------------------------------------------------- ------------- ---------- -----------
INSERT INTO T VALUES (:B1 )                        2n9c7yuww4dx4     100000           0
```

5.2.2　数据缓冲相关



1. 缓冲优化让第 2 次执行更快

```
SQL> --第 1 次执行
SQL> select count(*) from t;

  COUNT(*)
----------
     72884

已用时间:  00: 00: 00.12

执行计划
--------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   291   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 60918 |   291   (1)| 00:00:04 |
-----------------------------------------------------------------------
统计信息
--------
         28  recursive calls
          0  db block gets
       1103  consistent gets
       1038  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
```

```
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

```
SQL> --第 2 次执行
SQL> --该命令只是为了先不考虑解析的优化,单纯考虑第 2 次执行物理读减少带来的优化效应
SQL> alter system flush shared_pool;
系统已更改。
已用时间:  00: 00: 00.11
SQL> select count(*) from t;

  COUNT(*)
----------
     72884

已用时间:  00: 00: 00.04
执行计划
----------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   291   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 60918 |   291   (1)| 00:00:04 |
-------------------------------------------------------------------
统计信息
--------
        282  recursive calls
          0  db block gets
       1131  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 5-5　缓冲优化让第二次执行更快

通过试验一看出，第 1 次执行的 physical reads 的值为 1038，第 2 次执行则值为 0，物理读瞬间大幅度减少，性能自然就提高了。不过这里细心的读者可能会发现，recursive calls 都为 282，并没有变化，因为这里我们通过 alter system flush shared_pool，特意将解析优化的缓存给取消了。

2. 解析和缓冲优化一起来

```
SQL> --第 1 次执行
SQL> select count(*) from t;
```

```
  COUNT(*)
----------
     72884
已用时间:  00: 00: 00.11
执行计划
--------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   291   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 69684 |   291   (1)| 00:00:04 |
-----------------------------------------------------------------
统计信息
--------
         28  recursive calls
          0  db block gets
       1110  consistent gets
       1038  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

```
SQL> --第 2 次执行
SQL> --这里不做 shared_pool 和 buffer_cache 的 flush
SQL> select count(*) from t;
  COUNT(*)
----------
     72884
已用时间:  00: 00: 00.02
执行计划
--------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   291   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 69684 |   291   (1)| 00:00:04 |
-----------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1043  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

```
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 5-6　解析和缓冲优化一起来

接下来的这个试验还请同时观察 recursive calls 与 physical reads，第 2 次执行后这两个取值皆为 0，这是解析和缓存同时优化后的情况。

3. 直接路径读性能略胜一筹

环境准备：

```
--环境准备（构造一个有 100 万左右记录的表）
drop table t purge;
create table t   as select * from dba_objects;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
commit;
```

测试普通插入：

```
SQL> set timing off
SQL> set autotrace off
SQL> drop table test;
表已删除。
SQL> create table test   as select * from dba_objects where 1=2;
表已创建。
SQL> set timing on
SQL> insert into test select * from t;
已创建 1777808 行。
已用时间:  00: 00: 14.81
SQL> commit;
提交完成。
已用时间:  00: 00: 00.04
SQL> --注意这个普通方式插入试验输出的物理读（这是首次读哦）
SQL> set autotrace traceonly
SQL> select count(*) from test;
已用时间:  00: 00: 00.13
执行计划
--------
Plan hash value: 1950795681
-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |  4863   (1)| 00:00:59 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| TEST |  1868K|  4863   (1)| 00:00:59 |
-------------------------------------------------------------------------
```

```
Note
----
   - dynamic sampling used for this statement
统计信息
--------
         29  recursive calls
          1  db block gets
      27798  consistent gets
          0  physical reads
        176  redo size
        422  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

测试直接路径插入方式：

```
SQL> set timing off
SQL> drop table test;
表已删除。
SQL> create table test  as select * from dba_objects where 1=2;
表已创建。
SQL> set timing on
SQL> insert  /*+ append */ into test select * from t;
已创建 1777808 行。
已用时间:  00: 00: 04.22
SQL> commit;
提交完成。
已用时间:  00: 00: 00.00
SQL> --注意这个直接路径方式插入试验输出的物理读（这是首次读哦）
SQL> set autotrace traceonly
SQL> select count(*) from test;
已用时间:  00: 00: 05.56
执行计划
--------
Plan hash value: 1950795681
---------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |  4815   (1)| 00:00:58 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| TEST |  1445K|  4815   (1)| 00:00:58 |
-----------------------------------------------------------------------
Note
----
   - dynamic sampling used for this statement
统计信息
--------
         28  recursive calls
```

```
      1  db block gets
  27551  consistent gets
  27469  physical reads
    168  redo size
    422  bytes sent via SQL*Net to client
    416  bytes received via SQL*Net from client
      2  SQL*Net roundtrips to/from client
      0  sorts (memory)
      0  sorts (disk)
      1  rows processed
```

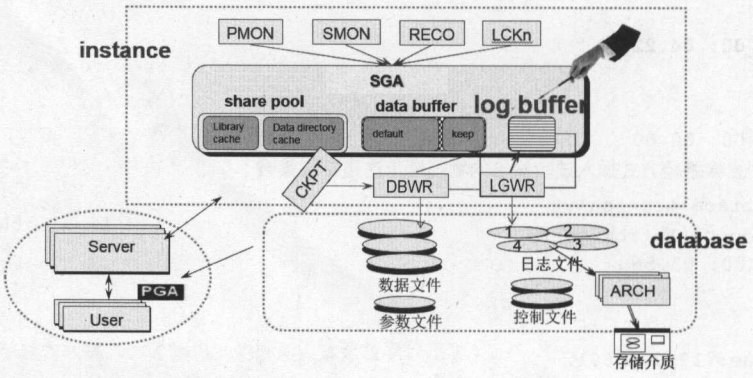
脚本 5-7　解析和缓冲优化一起来

可以看出，普通的插入需要的时间是 00: 00: 14.81，而直接路径插入的时间是 00: 00: 04.22，差异还是非常大的。为啥会有这个差别呢？继续分析上面的试验我们可以看出，普通插入后，继续查询的物理读为 0，而直接路径插入后，物理读为 27469。这是为啥呢？因为直接路径读写可以绕开 SGA，这对于插入而言少做事了，所以性能当然更好。

不过话说回来，优化都是相对的，插入由于不缓存数据，所以插入快了，查询同样也会因为没有缓存数据，而导致接下来的查询就不会快了。

5.2.3　日志归档相关

Oracle 的体系结构的 log buffer 部分，如下图所示：



1. 批量提交与否性能差异明显

```
SQL> drop table t purge;
表已删除。
SQL> create table t(x int);
表已创建。
SQL> set timing on
SQL> begin
  2      for i in 1 .. 100000 loop
  3         insert into t1 values (i);
  4        commit;
  5      end loop;
```

```
  6  end;
  7  /
PL/SQL 过程已成功完成。
已用时间:  00: 00: 11.21
```

```
SQL> drop table t purge;
表已删除。
SQL> create table t(x int);
表已创建。
SQL> begin
  2      for i in 1 .. 100000  loop
  3         insert into t values (i);
  4      end loop;
  5    commit;
  6  end;
  7  /
PL/SQL 过程已成功完成。
已用时间:  00: 00: 04.26
```

脚本 5-8　批量提交与否的性能差异

可以看出，未使用批量提交，耗时 11.21s，使用后耗时 4.26s，有天壤之别。

2. 日志关闭与否对性能的影响

```
--环境准备(构造一个有 700 万左右记录的表)
drop table t purge;
create table t   as select * from dba_objects;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
--多插几次，让数据大一点
insert into t  select * from t;
insert into t  select * from t;
commit;
```

测试直接路径写方式：

```
SQL> drop table test;
表已删除。
SQL> create table test  as select * from dba_objects where 1=2;
表已创建。
SQL> set timing on
SQL> insert  /*+ append */ into test select * from t;
已创建 7111232 行。
已用时间:  00: 00: 23.68
SQL> commit;
提交完成。
已用时间:  00: 00: 00.01
```

脚本 5-9　直接路径写插入

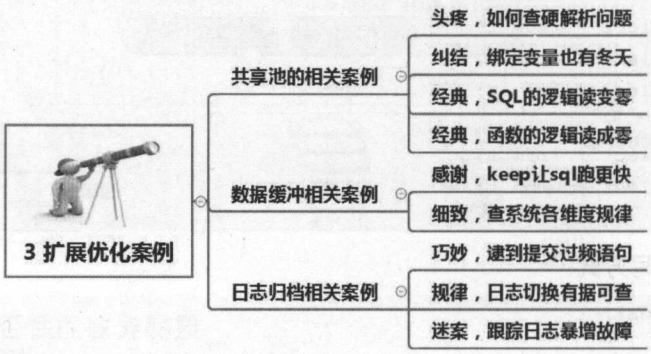
测试 nolgging 关闭日志+直接路径写方式：

```
SQL> drop table test;
表已删除。
已用时间:  00: 00: 00.19
SQL> create table test  as select * from dba_objects where 1=2;
表已创建。
已用时间:  00: 00: 00.10
SQL> alter table test nologging;
表已更改。
已用时间:  00: 00: 00.05
SQL> set timing on
SQL> insert  /*+ append */ into test select * from t;
已创建 7111232 行。
已用时间:  00: 00: 12.08
SQL> commit;
提交完成。
已用时间:  00: 00: 00.01
```

脚本 5-10　直接路径写加关闭日志的插入

从这里可以清晰看出，不关闭日志的插入耗时 23.68s，而关闭日志的插入耗时 12.08s，差别巨大。当然我们不是说鼓励大家都把日志关闭了，只是告诉大家，在某些特定的场景下，数据不是非常重要的时候，允许关闭日志的时候，性能可以有较大幅度的提升。

5.3　扩展优化案例



5.3.1　与共享池相关

1. 头疼，如何查硬解析问题

某系统遭遇大量硬解析的性能瓶颈，相关人员无从下手来找到具体的未使用绑定变量的 SQL，这时候该如何处理呢？我们模拟一个这样的场景，来看看如何定位出未使用绑定变量的 SQL。首先构造一个未使用绑定变量并频繁执行的 SQL 如下：

```
drop table t purge;
create table t(x int);
select * from v$mystat where rownum=1;
begin
    for i in 1 .. 100000
    loop
        execute immediate
        'insert into t values ( '||i||')';
    end loop;
    commit;
end;
/
```

脚本 5-11　构造未使用绑定变量的 SQL

捕获出需要使用绑定变量的 SQL 的思路如下，其原理是未使用绑定变量的 SQL 比较类似，通过@替换相似部分，然后提取相同的分组，从而找出未使用绑定变量的 SQL。

```
drop table t_bind_sql purge;
create table t_bind_sql as select sql_text,module from v$sqlarea;
alter table t_bind_sql add sql_text_wo_constants varchar2(1000);
create or replace function
remove_constants( p_query in varchar2 ) return varchar2
as
    l_query long;
    l_char  varchar2(10);
    l_in_quotes boolean default FALSE;
begin
    for i in 1 .. length( p_query )
    loop
        l_char := substr(p_query,i,1);
        if ( l_char = '''' and l_in_quotes )
        then
            l_in_quotes := FALSE;
        elsif ( l_char = '''' and NOT l_in_quotes )
        then
            l_in_quotes := TRUE;
            l_query := l_query || '''#';
        end if;
        if ( NOT l_in_quotes ) then
            l_query := l_query || l_char;
        end if;
    end loop;
    l_query := translate( l_query, '0123456789', '@@@@@@@@@@' );
    for i in 0 .. 8 loop
        l_query := replace( l_query, lpad('@',10-i,'@'), '@' );
        l_query := replace( l_query, lpad(' ',10-i,' '), ' ' );
    end loop;
    return upper(l_query);
```

```
end;
/
update t_bind_sql set sql_text_wo_constants = remove_constants(sql_text);
commit;
```

接下来用如下方式就可以快速定位了：

```
--执行完上述动作后，以下 SQL 语句可以完成未绑定变量语句的统计
set linesize 266
col  sql_text_wo_constants format a30
col  module format  a30
col  CNT format  999999
select sql_text_wo_constants, module,count(*) CNT
  from t_bind_sql
 group by sql_text_wo_constants,module
having count(*) > 100
 order by 3 desc;
```

脚本 5-12　获取未使用绑定变量 SQL 的方法

接下来的情况如下所示，果然定位非常清晰，就是它了！

```
SQL> ---执行完上述动作后，以下SQL语句可以完成未绑定变量语句的统计
SQL> set linesize 266
SQL> col  sql_text_wo_constants format a30
SQL> col  module format  a30
SQL> col  CNT format  999999
SQL> select sql_text_wo_constants, module,count(*) CNT
  2    from t_bind_sql
  3   group by sql_text_wo_constants,module
  4  having count(*) > 100
  5   order by 3 desc;

SQL_TEXT_WO_CONSTANTS          MODULE                           CNT
------------------------------ ------------------------------ -------
INSERT INTO T VALUES ( @)      SQL*Plus                        7366
```

2. 怪哉，SQL 的逻辑读成零

大家知道，SQL 的逻辑读一般不可能为 0，不过有一种情况，真的可以让 SQL 的逻辑读为 0，那就是缓存结果集。设定方法是在 SQL 语句中加上如/*+ result_cache */的 hint 或者在 session 属性上固定这个特性。当设定完毕后，第一次执行的时候，SQL 执行的结果集会被扔到 SGA 的 Shared pool 中，下次执行的时候直接把它当作结果来用就 OK 了，因为根本不需要访问任何指定对象，所以逻辑读为 0，试验如下：

```
drop table t purge;
create table t as select * from dba_objects;
insert into t  select * from t;
commit;
SQL> set autotrace on
select /*+ result_cache */ count(*) from t;
SQL> ---接下来再次执行(居然发现逻辑读为 0 )：
SQL> select /*+ result_cache */ count(*) from t;
  COUNT(*)
```

```
----------
    145762
已用时间:  00: 00: 00.01
执行计划
--------
| Id  | Operation           | Name                          | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                               |     1 |   589   (1)| 00:00:08 |
|   1 |  RESULT CACHE       | d827qx1jmwjc86yqynrp1kvpny |       |            |          |
|   2 |   SORT AGGREGATE    |                               |     1 |            |          |
|   3 |    TABLE ACCESS FULL| T                             |   277K|   589   (1)| 00:00:08 |
-----------------------------------------------------------------------------------------
-
统计信息
--------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 5-13　缓存结果集让逻辑读为 0

使用缓存结果集一定要注意应用的场景，因为 SQL 对应的对象（比如表）等一变化，相关 SQL 的缓存结果集就自动失效了。所以这个技术一般用在表记录很少变化的情况。

3. 惊奇，函数的逻辑读成零

与 SQL 的缓存结果集类似，缓存函数的结果集也可以使逻辑读为 0。某些情况下，函数的结果集由于基表等数据没有发生变化，故而其也会保持不变，这个时候为了避免重复计算函数，就可以应用缓存函数结果集的技术。同样这个结果集也存在 Shared pool 中。具体构造的案例脚本如下：

```
drop table t;
CREATE TABLE T AS SELECT * FROM DBA_OBJECTS;

CREATE OR REPLACE FUNCTION F_NO_RESULT_CACHE RETURN NUMBER AS
V_RETURN NUMBER;
BEGIN
SELECT COUNT(*) INTO V_RETURN FROM T;
RETURN V_RETURN;
END;
/
```

```
SQL> --看调用 F_NO_RESULT_CACHE 执行第 2 次后的结果
SQL> SELECT F_NO_RESULT_CACHE FROM DUAL;
F_NO_RESULT_CACHE
-----------------
      72883
统计信息
--------
          1  recursive calls
          0  db block gets
       1043  consistent gets
          0  physical reads
          0  redo size
        434  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

SQL> --看调用 F_RESULT_CACHE 执行第 2 次后的结果
SQL> SELECT F_RESULT_CACHE FROM DUAL;
F_RESULT_CACHE
--------------
    72883
统计信息
--------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
        431  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 5-14　缓存函数结果集让函数逻辑读为 0

5.3.2　数据缓冲相关

1. 感谢，keep 让 SQL 跑得更快

大家知道，Data buffer 的数据是会被挤出去的，所以有的时候，为了避免某些重要的数据被挤出去，我们会采取一些特殊的手段来固定这部分数据，这也是一个常见的做法。如何来固定这部分数据呢？具体的思路和方法如下。

未固定时的情况：

```
SQL> alter system set db_keep_cache_size=100M;
系统已更改。
SQL> drop table t;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> create index idx_object_id on t(object_id);
索引已创建。
SQL> select BUFFER_POOL from user_tables where  TABLE_NAME='T';
BUFFER_
-------
DEFAULT
SQL> select BUFFER_POOL from user_indexes where INDEX_NAME='IDX_OBJECT_ID';
BUFFER_
-------
DEFAULT
```

脚本 5-15　未固定缓存的情况

固定的方式和查询的方式如下：

```
SQL> alter index idx_object_id storage(buffer_pool keep);
索引已更改。
SQL> --以下将索引全部读进内存
SQL> select /*+index(t,idx_object_id)*/ count(*) from t where object_id is not null;

  COUNT(*)
----------
    111113
SQL> --以下将数据全部读进内存
SQL> alter table t storage(buffer_pool keep);
表已更改。
SQL> select /*+full(t)*/ count(*) from  t;
  COUNT(*)
----------
    111113
SQL> --执行 KEEP 操作后，通过如下方法查询出 BUFFER_POOL 列值为 KEEP,表示已经 KEEP 成功了
SQL> select BUFFER_POOL from user_tables where  TABLE_NAME='T';
BUFFER_
-------
KEEP
SQL> select BUFFER_POOL from user_indexes where INDEX_NAME='IDX_OBJECT_ID';
BUFFER_
-------
KEEP
```

脚本 5-16　固定缓存的情况

2. 细致，查系统各维度规律

数据库运行得健康与否是有很多指标的，我们可以通过每段时间数据库运行时间、日志、

逻辑读、解析、事务数等指标的变化，来分析数据库在某个时段是否有异常。这是一个非常有用的监控数据库运行状态、定位数据库问题故障点的方法。脚本如下：

```
select s.snap_date,
       decode(s.redosize, null, '--shutdown or end--', s.currtime) "TIME",
       to_char(round(s.seconds/60,2)) "elapse(min)",
       round(t.db_time / 1000000 / 60, 2) "DB time(min)",
       s.redosize redo,
       round(s.redosize / s.seconds, 2) "redo/s",
       s.logicalreads logical,
       round(s.logicalreads / s.seconds, 2) "logical/s",
       physicalreads physical,
       round(s.physicalreads / s.seconds, 2) "phy/s",
       s.executes execs,
       round(s.executes / s.seconds, 2) "execs/s",
       s.parse,
       round(s.parse / s.seconds, 2) "parse/s",
       s.hardparse,
       round(s.hardparse / s.seconds, 2) "hardparse/s",
       s.transactions trans,
       round(s.transactions / s.seconds, 2) "trans/s"
  from (select curr_redo - last_redo redosize,
               curr_logicalreads - last_logicalreads logicalreads,
               curr_physicalreads - last_physicalreads physicalreads,
               curr_executes - last_executes executes,
               curr_parse - last_parse parse,
               curr_hardparse - last_hardparse hardparse,
               curr_transactions - last_transactions transactions,
               round(((currtime + 0) - (lasttime + 0)) * 3600 * 24, 0) seconds,
               to_char(currtime, 'yy/mm/dd') snap_date,
               to_char(currtime, 'hh24:mi') currtime,
               currsnap_id endsnap_id,
               to_char(startup_time, 'yyyy-mm-dd hh24:mi:ss') startup_time
          from (select a.redo last_redo,
                       a.logicalreads last_logicalreads,
                       a.physicalreads last_physicalreads,
                       a.executes last_executes,
                       a.parse last_parse,
                       a.hardparse last_hardparse,
                       a.transactions last_transactions,
                       lead(a.redo, 1, null) over(partition by b.startup_time order
by b.end_interval_time) curr_redo,
                       lead(a.logicalreads, 1, null) over(partition by b.startup_time
order by b.end_interval_time) curr_logicalreads,
                       lead(a.physicalreads, 1, null) over(partition by b.startup_
time order by b.end_interval_time) curr_physicalreads,
                       lead(a.executes, 1, null) over(partition by b.startup_time
order by b.end_interval_time) curr_executes,
                       lead(a.parse, 1, null) over(partition by b.startup_time order
```

```
by b.end_interval_time) curr_parse,
                    lead(a.hardparse, 1, null) over(partition by b.startup_time
order by b.end_interval_time) curr_hardparse,
                    lead(a.transactions, 1, null) over(partition by b.startup_time
order by b.end_interval_time) curr_transactions,
                    b.end_interval_time lasttime,
                    lead(b.end_interval_time, 1, null) over(partition by b.startup_
time order by b.end_interval_time) currtime,
                    lead(b.snap_id, 1, null) over(partition by b.startup_time
order by b.end_interval_time) currsnap_id,
                    b.startup_time
               from (select snap_id,
                            dbid,
                            instance_number,
                            sum(decode(stat_name, 'redo size', value, 0)) redo,
                            sum(decode(stat_name,
                                       'session logical reads',
                                       value,
                                       0)) logicalreads,
                            sum(decode(stat_name,
                                       'physical reads',
                                       value,
                                       0)) physicalreads,
                            sum(decode(stat_name, 'execute count', value, 0)) executes,
                            sum(decode(stat_name,
                                       'parse count (total)',
                                       value,
                                       0)) parse,
                            sum(decode(stat_name,
                                       'parse count (hard)',
                                       value,
                                       0)) hardparse,
                            sum(decode(stat_name,
                                       'user rollbacks',
                                       value,
                                       'user commits',
                                       value,
                                       0)) transactions
                       from dba_hist_sysstat
                      where stat_name in
                            ('redo size',
                             'session logical reads',
                             'physical reads',
                             'execute count',
                             'user rollbacks',
                             'user commits',
                             'parse count (hard)',
                             'parse count (total)')
                      group by snap_id, dbid, instance_number) a,
```

```
                    dba_hist_snapshot b
                where a.snap_id = b.snap_id
                  and a.dbid = b.dbid
                  and a.instance_number = b.instance_number
                order by end_interval_time)) s,
       (select lead(a.value, 1, null) over(partition by b.startup_time order by
b.end_interval_time) - a.value db_time,
               lead(b.snap_id, 1, null) over(partition by b.startup_time order by
b.end_interval_time) endsnap_id
          from dba_hist_sys_time_model a, dba_hist_snapshot b
         where a.snap_id = b.snap_id
           and a.dbid = b.dbid
           and a.instance_number = b.instance_number
           and a.stat_name = 'DB time') t
 where s.endsnap_id = t.endsnap_id
 order by  s.snap_date ,time desc;
```

	SNAP_DATE	TIME	elapse(min)	DB time(min)	REDO	redo/s	LOGICAL	ogical/s	PHYSICAL	phy/s	EXECS	execs/s	PARSE	parse/s	HARDP
1	13/11/07	18:00	64.22	0.51	105873536	27478.21	1375950	357.11	225	0.06	160181	41.57	9018	2.34	
2	13/11/07	16:56	115.77	0.04	1622588	233.6	43732	6.3	11	0	7177	1.03	4045	0.58	
3	13/11/07	15:00	60.18	0.06	5690688	1575.93	106134	29.39	5518	1.53	13346	3.7	7542	2.09	
4	13/11/07	14:00	60.17	0.65	197937568	54830.35	2338751	647.85	7149	1.98	406318	112.55	10217	2.83	
5	13/11/07	13:00	60.18	0.21	126920828	35148.39	666700	184.63	6791	1.88	14344	3.97	9617	2.66	
6	13/11/07	12:00	59.17	0.9	357269612	100639.33	1655494	466.34	6976	1.97	15255	4.3	9805	2.76	
7	13/11/07	11:00	60.17	0.57	60423804	16737.9	310342	85.97	4853	1.34	553894	153.43	9828	2.72	
8	13/11/07	08:01	60.18	0.06	2089688	578.7	59616	16.51	136	0.04	10912	3.02	6564	1.82	
9	13/11/07	07:00	60.17	0.05	2543732	704.63	68438	18.96	469	0.13	12292	3.4	7171	1.99	
10	13/11/07	06:00	34.1	0.62	3908728	1910.42	2313056	1130.53	2890	1.41	41855	20.46	8246	4.03	
11	13/11/08	22:00	60.18	1.25	4584236	1269.52	500443	138.59	9367	2.59	50851	14.08	10969	3.04	
12	13/11/08	21:00	60.17	0.06	2188264	606.17	57236	15.85	10	0	10882	3.01	6581	1.82	
13	13/11/08	20:00	59.18	0.06	2340524	659.12	59905	16.87	11	0	10844	3.05	6570	1.85	
14	13/11/08	19:01	60.9	0.09	2363340	646.78	55629	15.22	26	0.01	9964	2.73	5928	1.62	
15	13/11/08	18:00	59.18	0.06	2395280	674.54	64119	18.06	40	0.01	11448	3.22	6930	1.95	

脚本 5-17　查询数据库分时段的健康状况

很显然，这个输出结果对我们确定数据库的峰值时间点能起到一定的指导作用。

5.3.3　日志归档相关

1. 巧妙，逮到提交过频语句

很多生产系统由于没有注意批量提交的事情，出现了性能问题，产生了相关的日志等待。表现在告警日志上，就是日志切换过于频繁。此时我们也可以通过观察 AWR 报表中事务个数和每个事务的尺寸来看出问题所在。不过很多时候，我们也可以通过如下方法及时地跟踪到具体未批量提交的 SQL 是哪些，如下所示：

```
drop table t purge;
create table t(x int);

select * from v$mystat where rownum=1;
begin
    for i in 1 .. 100000 loop
       insert into t values (i);
```

```
      commit;
    end loop;
end;
/
```

step 1 获取提交次数超过一个阈值的 SID。

```
SQL>  select t1.sid, t1.value, t2.name
  2     from v$sesstat t1, v$statname t2
  3    where t2.name like '%user commits%'
  4      and t1.STATISTIC# = t2.STATISTIC#
  5      and value >= 10000
  6    order by value desc;
   SID      VALUE NAME
----------------------
   132     100003 user commits
```

step 2 获取到对应的 Sql_id。

```
SQL>   select t.SID,
  2           t.PROGRAM,
  3              t.EVENT,
  4           t.LOGON_TIME,
  5           t.WAIT_TIME,
  6           t.SECONDS_IN_WAIT,
  7           t.SQL_ID,
  8           t.PREV_SQL_ID
  9     from v$session t
 10    where sid in(132);
SID PROGRAM     EVENT                          LOGON_TIME    WAIT_TIME SECONDS_IN_WAIT SQL_ID    PREV_SQL_ID
-------------------------------------------------------------------------------------------------------------
132 sqlplus.exe  SQL*Net message from client  13-11 月-13       0             77          ccpn5c32bmfmf
```

step 3 通过 Sql_id 得到对应的 SQL。

```
SQL>   select t.sql_id,
  2           t.sql_text,
  3           t.EXECUTIONS,
  4           t.FIRST_LOAD_TIME,
  5           t.LAST_LOAD_TIME
  6    from v$sqlarea t
  7   where sql_id in ('ccpn5c32bmfmf');
SQL_ID             SQL_TEXT                EXECUTIONS     FIRST_LOAD_TIME        LAST_LOAD_TIME
-------------------------------------------------------------------------------------------------
ccpn5c32bmfmfbegin    for i in 1 .. 100000 loop    1    2013-11-13/16:13:56       13-11 月-13
                        insert into t values (i);
        commit;
       end loop;
         end;
```

脚本 5-18　如何捕获提交过于频繁的语句

2. 规律，日志切换有据可查

```
SELECT SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH:MI:SS'),1,5) Day,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'00',1,0)) H00,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'01',1,0)) H01,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'02',1,0)) H02,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'03',1,0)) H03,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'04',1,0)) H04,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'05',1,0)) H05,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'06',1,0)) H06,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'07',1,0)) H07,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'08',1,0)) H08,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'09',1,0)) H09,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'10',1,0)) H10,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'11',1,0)) H11,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'12',1,0)) H12,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'13',1,0)) H13,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'14',1,0)) H14,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'15',1,0)) H15,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'16',1,0)) H16,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'17',1,0)) H17,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'18',1,0)) H18,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'19',1,0)) H19,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'20',1,0)) H20,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'21',1,0)) H21,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'22',1,0)) H22 ,
       SUM(DECODE(SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH24:MI:SS'),10,2),'23',1,0)) H23,
       COUNT(*) TOTAL
FROM v$log_history  a
   where first_time>=to_char(sysdate-11)
GROUP BY SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH:MI:SS'),1,5)
ORDER BY SUBSTR(TO_CHAR(first_time, 'MM/DD/RR HH:MI:SS'),1,5) DESC;
```

输出结果如下，因此可以很清楚地看出 11 月 12 日问题比较明显。

	DAY	H00	H01	H02	H03	H04	H05	H06	H07	H08	H09	H10	H11	H12	H13	H14	H15	H16	H17	H18	H19	H20	H21	H22	H23	TOTAL
1	11/13	0	0	0	0	0	0	1	0	1	0	0	0	0	3	9	2	25	18	1	0	5	0	0	0	65
2	11/12	0	0	0	0	1	4	1	0	7	8	0	1	17	36	0	0	1	5	1	0	1	0	1	0	84
3	11/11	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	5
4	11/10	0	0	0	0	0	0	1	0	0	0	8	0	0	0	0	0	0	1	1	0	0	0	0	0	11
5	11/09	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	1	0	1	0	1	0	0	0	6
6	11/08	1	0	0	0	0	1	0	0	0	2	3	0	0	1	1	0	0	0	0	0	0	0	2	0	11
7	11/07	0	0	0	0	0	1	0	0	5	0	2	8	3	5	0	0	0	3	1	0	1	0	0	1	30
8	11/06	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	4	0	1	0	2	0	11
9	11/05	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	4
10	11/04	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	4

脚本 5-19　查询日志切换规律的语句

3. 迷案，跟踪日志暴增故障

XXX 工程点最近经常出现数据库无法连接的问题，经查是归档空间满导致，清理归档后恢复正常。由 awr 报表中 load profile 可以算出，每天日志量大约为 160GB，而 XXX 工程点的归

档空间大约为 1TB 左右，这样不到一周就满了。

按照以下步骤可以查出这些 redo 日志是谁产生的，本例中的故障是由一个自身监控的存储过程导致。脚本如下：

```
--1. redo 大量产生必然是由于大量产生"块改变"。从 awr 视图中找出"块改变"最多的 segments。
select * from (
SELECT to_char(begin_interval_time, 'YYYY_MM_DD HH24:MI') snap_time,
       dhsso.object_name,
       SUM(db_block_changes_delta)
  FROM dba_hist_seg_stat     dhss,
       dba_hist_seg_stat_obj dhsso,
       dba_hist_snapshot     dhs
 WHERE dhs.snap_id = dhss. snap_id
   AND dhs.instance_number = dhss. instance_number
   AND dhss.obj# = dhsso. obj#
   AND dhss.dataobj# = dhsso.dataobj#
   AND begin_interval_time> sysdate - 60/1440
 GROUP BY to_char(begin_interval_time, 'YYYY_MM_DD HH24:MI'),
          dhsso.object_name
 order by 3 desc)
 where rownum<=5;

--2. 从 awr 视图中找出步骤 1 中排序靠前的对象涉及的 SQL。
SELECT to_char(begin_interval_time, 'YYYY_MM_DD HH24:MI'),
       dbms_lob.substr(sql_text, 4000, 1),
       dhss.instance_number,
       dhss.sql_id,
       executions_delta,
       rows_processed_delta
  FROM dba_hist_sqlstat dhss, dba_hist_snapshot dhs, dba_hist_sqltext dhst
 WHERE UPPER(dhst.sql_text) LIKE '%这里写对象名大写%'
   AND dhss.snap_id = dhs.snap_id
   AND dhss.instance_Number = dhs.instance_number
   AND dhss.sql_id = dhst.sql_id;

--3. 从 ASH 相关视图中找出执行这些 SQL 的 session、module 和 machine。
select * from dba_hist_active_sess_history WHERE sql_id = '';
select * from v$active_session_history where sql_Id = '';

--4. dba_source 看看是否有存储过程包含这个 SQL。
```

```
--以下操作产生大量的 redo,可以用上述的方法跟踪它们。
drop table   test_redo  purge;
create table test_redo as select * from dba_objects;
insert into  test_redo select * from test_redo;
```

```
insert into  test_redo select * from test_redo;
insert into  test_redo select * from test_redo;
insert into  test_redo select * from test_redo;
insert into  test_redo select * from test_redo;
exec dbms_workload_repository.create_snapshot();
```

解决过程

```
--执行了大量的针对 test_redo 表的 INSERT 操作后，我们开始按如下方法进行跟踪，看能否发现更新的是哪张表，是哪些语句。
SQL> select * from (
  2   SELECT to_char(begin_interval_time, 'YYYY_MM_DD HH24:MI') snap_time,dhsso.object_name,SUM(db_block_changes_delta)
  3    FROM dba_hist_seg_stat dhss,dba_hist_seg_stat_obj dhsso,dba_hist_snapshot  dhs
  4   WHERE dhs.snap_id = dhss. snap_id
  5      AND dhs.instance_number = dhss. instance_number AND dhss.obj# = dhsso. obj# AND dhss.dataobj# = dhsso.dataobj#
  6     AND begin_interval_time> sysdate - 60/1440
  7    GROUP BY to_char(begin_interval_time, 'YYYY_MM_DD HH24:MI'), dhsso.object_name order by 3 desc)
  8  where rownum<=3;
SNAP_TIME        OBJECT_NAME                    SUM(DB_BLOCK_CHANGES_DELTA)
-------------------------------------------------------------------------------
2015_11_13 20:00 TEST_REDO                                          178272
2015_11_13 20:00 MGMT_CURRENT_METRICS_PK                               224
2015_11_13 20:00 MGMT_SYSTEM_PERF_LOG_IDX_01                           160


SQL>  SELECT  to_char(begin_interval_time,'YYYY_MM_DD  HH24:MI'),dbms_lob.substr(sql_text,4000,1),dhss.sql_id,executions_delta,rows_processed_delta
  2    FROM dba_hist_sqlstat dhss, dba_hist_snapshot dhs, dba_hist_sqltext dhst
  3   WHERE UPPER(dhst.sql_text) LIKE '%TEST_REDO%' AND dhss.snap_id = dhs.snap_id
  4    AND dhss.instance_Number = dhs.instance_number AND dhss.sql_id = dhst.sql_id;
TO_CHAR(BEGIN_IN    DBMS_LOB.SUBSTR(SQL_TEXT,4000,1)   INSTANCE_NUMBER SQL_ID EXECUTIONS_DELTA   ROWS_PROCESSED_DELTA
--------------------
2015_11_13  17:00    create  table  test_redo  as  select  *  from  dba_objects  1
dsf2uj3pzzadg       1        72884
2015_11_13  20:00    insert  into  test_redo  select  *  from  test_redo          1
5w8pb7t27c85n       5        2259404
```

脚本 5-20　查出日志暴增的 SQL 语句

5.4　本章习题、总结与延伸

习题 1：说说绑定变量的好处和坏处，请举场景说明。

习题 2：说说直接路径访问方式为什么更快。

习题 3：请用自己的话描述什么时候能用缓存结果集，什么时候能用 keep 内存。

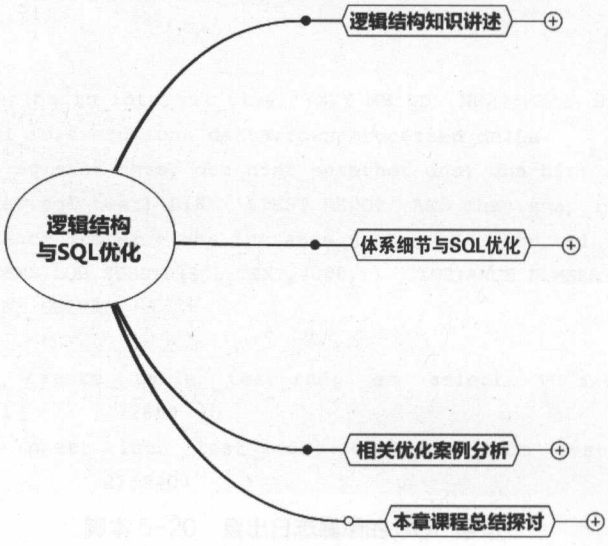
习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

第6章　且慢，体验逻辑结构让 SQL 飞

你可能没想过从逻辑结构原理中动手优化

逻辑结构与 SQL 优化之间的关系是大部分人容易忽略的，本章我们先从简单的逻辑结构知识开始介绍，接下来对所有可能和 SQL 优化有关的逻辑结构的细节做进一步的描述。

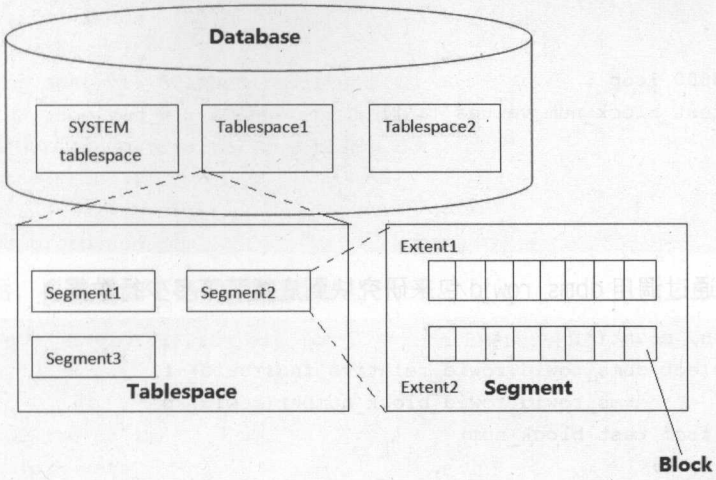
随后就是案例剖析环节，在该环节让读者真真切切感知到逻辑结构在影响着工作中的各种场景。最后是思考回顾。本章总体学习思路如下图所示：



6.1　逻辑结构

Oracle 的逻辑结构是一种层次结构。主要由表空间、段、区和数据块等概念组成。逻辑结

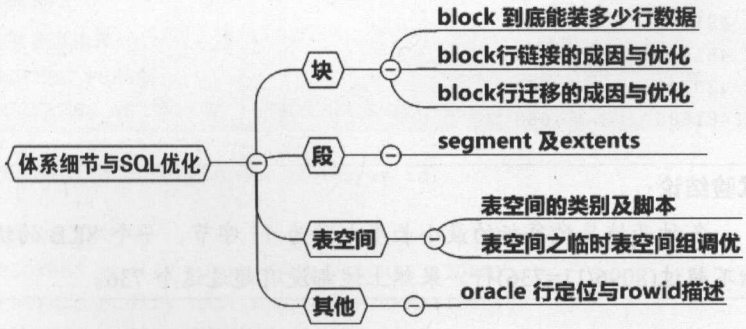
构是面向用户的，用户使用 Oracle 开发应用程序使用的就是逻辑结构。数据库存储层次结构及其构成关系，结构对象也从数据块到表空间形成了不同层次的粒度关系。如下图所示：



数据库（DATABASE）由若干表空间（TABLESPACE）组成， 表空间（TABLESPACE）由若干段（SEGMENT）组成，段（SEGMENT）由若干区（EXTENT）组成，区（EXTENT）又由 Oracle 的最小单元块（BLOCK）组成。

6.2 体系细节与 SQL 优化

逻辑体系结构和 SQL 优化是息息相关的，下面我们从块、段、表空间、rowid 等维度来进行阐述，具体如下图所示：



6.2.1 Block

1. Block 最多能装多少行

一个 8KB 的块其最大可用空间有 8096 字节，如果一个字节装一行的数据，能否插入 8000 行呢？我们做些试验看看。首先是构造环境，如下：

```
drop table test_block_num purge;

create table test_block_num (id varchar2(1));

begin
 for i  in 1..8000 loop
  insert into test_block_num values('a');
 end loop;
 commit;
end;
/
```

接下来我们通过调用 dbms_rowid 包来研究块到底能装下多少行数据。

```
SQL> select f, b, count(*)
  2    from (select dbms_rowid.rowid_relative_fno(rowid) f,
  3                 dbms_rowid.rowid_block_number(rowid) b
  4            from test_block_num)
  5   group by f, b;

         F          B   COUNT(*)
---------- ---------- ----------
         4     433740        660
         4     481685        660
         4     433742        660
         4     433743        660
         4     481681        660
         4     481683         80
         4     481684        660
         4     481688        660
         4     433744        660
         4     481682        660
         4     481687        660
         4     433741        660
         4     481686        660
```

试验结论：

各种开销导致每行的最小长度大致为 11 字节，一个 8KB 的块理论上最多存储不超过(8096/11=736)行。果然上述都没有超过这个 736。

脚本 6-1　研究块究竟能装多少行数据

2. Block 行迁移的成因与优化

行迁移优化前，看看如下语句逻辑读情况：

```
SQL> select /*+index(EMPLOYEES,idx_emp_id)*/ * from EMPLOYEES  where employee_id>0;
统计信息
--------
```

```
          0  recursive calls
          0  db block gets
        219  consistent gets
          0  physical reads
          0  redo size
     437664  bytes sent via SQL*Net to client
        492  bytes received via SQL*Net from client
          9  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        107  rows processed
```

行迁移优化后，再看看如下语句逻辑读情况：

```
SQL> select /*+index(EMPLOYEES,idx_emp_id)*/ * from EMPLOYEES  where employee_id>0;
统计信息
--------
          0  recursive calls
          0  db block gets
        116  consistent gets
          0  physical reads
          0  redo size
     437034  bytes sent via SQL*Net to client
        492  bytes received via SQL*Net from client
          9  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        107  rows processed
```

逻辑读从原来的219下降到116，性能明显提升。

具体试验步骤如下：

```
--- PCTFREE试验准备之建表
DROP TABLE EMPLOYEES PURGE;
CREATE TABLE EMPLOYEES AS SELECT * FROM HR.EMPLOYEES ;
desc EMPLOYEES;
create index idx_emp_id on employees(employee_id);

--- PCTFREE试验准备之扩大字段
alter table EMPLOYEES modify FIRST_NAME VARCHAR2(1000);
alter table EMPLOYEES modify LAST_NAME  VARCHAR2(1000);
alter table EMPLOYEES modify EMAIL VARCHAR2(1000);
alter table EMPLOYEES modify PHONE_NUMBER  VARCHAR2(1000);

--- PCTFREE试验准备之更新表
UPDATE EMPLOYEES
  SET FIRST_NAME = LPAD('1', 1000, '*'), LAST_NAME = LPAD('1', 1000, '*'), EMAIL =
LPAD('1', 1000, '*'),
  PHONE_NUMBER = LPAD('1', 1000, '*');
COMMIT;
```

```
---行迁移优化前，先看看该语句逻辑读情况(执行计划及代价都一样，没必要展现了，就展现 statistics 即可)。
SET AUTOTRACE traceonly
set linesize 1000
select /*+index(EMPLOYEES,idx_emp_id)*/ * from EMPLOYEES  where employee_id>0;
/
set autotrace off

----- 发现存在行迁移的方法。
--首先建 chaind_rows 相关表，这是必需的步骤。
--sqlplus "/ as sysdba"

sqlplus ljb/ljb
drop table chained_rows purge;
@?/rdbms/admin/utlchain.sql
----以下命令针对 EMPLOYEES 表和 EMPLOYEES_BK 做分析，将产生行迁移的记录插入到 chained_rows 表中
analyze table EMPLOYEES list chained rows into chained_rows;
select count(*)  from chained_rows where table_name='EMPLOYEES';

---以下方法可以去除行迁移

drop table EMPLOYEES_TMP;
create table EMPLOYEES_TMP as select * from EMPLOYEES where rowid in (select
head_rowid from chained_rows);
Delete from EMPLOYEES where rowid in (select head_rowid from chained_rows);
Insert into EMPLOYEES select * from EMPLOYEES_TMP;
delete from chained_rows ;
commit;
analyze table EMPLOYEES list chained rows into chained_rows;
select count(*)  from chained_rows where table_name='EMPLOYEES';
--这时的取值一定为 0，用这种方法做行迁移消除，肯定是没问题的!

---行迁移优化后，先看看该语句逻辑读情况(执行计划及代价都一样，没必要展现了，就展现 statistics 即可)
SET AUTOTRACE traceonly statistics
set linesize 1000
select /*+index(EMPLOYEES,idx_emp_id)*/ * from EMPLOYEES  where employee_id>0;
/
```

脚本 6-2　行迁移优化前后的测试脚本

3. Block 行链接的成因与优化

行链接一般来说无法避免，要通过增大数据块的方式消除（请注意脚本中的 TBS_LJB_16K 是一个 BLOCK 为 16KB 的表空间），试验如下：

```
SQL> analyze table EMPLOYEES list chained rows into chained_rows;
表已分析。
SQL> select count(*)  from chained_rows where table_name='EMPLOYEES';
  COUNT(*)
----------
       107
```

```
SQL> --行链接只有通过加大 BLOCK 块的方式才可以避免，如下：
SQL> DROP TABLE EMPLOYEES_BK PURGE;
表已删除。
SQL> CREATE TABLE EMPLOYEES_BK TABLESPACE TBS_LJB_16K AS SELECT * FROM EMPLOYEES;
表已创建。
SQL> delete from chained_rows ;
已删除 107 行。
SQL> commit;
提交完成。
SQL> analyze table EMPLOYEES_BK list chained rows into chained_rows;
表已分析。
SQL> select count(*)  from chained_rows where table_name='EMPLOYEES_BK';
  COUNT(*)
----------
         0
```

脚本 6-3　行链接消除的试验

6.2.2　Segment 与 extent

建一个 T 表就产生了表段、T 段(SEGMENT)，请观察区(EXTENT)及块（BLOCK）的个数。建一个索引 IDX_OBJ_ID 就产生了索引段，IDX_OBJ_ID 段(SEGMENT)和表的情况类似，试验如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t tablespace tbs_ljb as select * from dba_objects where rownum=1 ;
表已创建。
SQL> col segment_name format a15
SQL> col segment_type format a10
SQL> col tablespace_name format a20
SQL> col blocks  format 9999
SQL> col extents format 9999
SQL> select segment_name,
  2         segment_type,
  3         tablespace_name,
  4         blocks,extents,
  5         bytes/1024/1024
  6  from user_segments  where segment_name = 'T';
SEGMENT_NAME    SEGMENT_TY TABLESPACE_NAME      BLOCKS EXTENTS BYTES/1024/1024
--------------- ---------- -------------------- ------ ------- ---------------
T               TABLE      TBS_LJB                   8       1           .0625
SQL> select count(*) from   user_extents  WHERE segment_name='T';
  COUNT(*)
----------
         1
SQL> ---建一个索引 IDX_OBJ_ID 就产生了索引段，IDX_OBJ_ID 段(SEGMENT)和表的情况类似，如下：
SQL> create index idx_obj_id on t(object_id);
索引已创建。
```

```
SQL> select segment_name,
  2         segment_type,
  3         tablespace_name,
  4         blocks,
  5         extents,
  6         bytes/1024/1024
  7  from user_segments
  8  where segment_name = 'IDX_OBJ_ID';
SEGMENT_NAME    SEGMENT_TY TABLESPACE_NAME      BLOCKS EXTENTS BYTES/1024/1024
--------------- ---------- -------------------- ------ ------- ---------------
IDX_OBJ_ID      INDEX      USERS                     8       1           .0625
SQL> select count(*) from  user_extents  WHERE segment_name='IDX_OBJ_ID';
  COUNT(*)
----------
     1
```

脚本 6-4　体会 segment 与 extent 的试验

随着表记录的增加，表对应的 Extents 及 Blocks 的个数也不断增多。随着 Idx_obj_id 不断增大，索引对应的 Extents 及 Blocks 的个数也不断增多。如下：

```
SQL> insert into t select * from dba_objects ;
已创建 72882 行。
SQL> commit;
提交完成。
SQL> ---随着 T 表数据不断增加，区(EXTENT)及块（BLOCK）的个数也不断增多。如下：
SQL> select segment_name,
  2         segment_type,
  3         tablespace_name,
  4         blocks,
  5         extents,bytes/1024/1024
  6  from user_segments
  7  where segment_name = 'T';
SEGMENT_NAME    SEGMENT_TY TABLESPACE_NAME      BLOCKS EXTENTS BYTES/1024/1024
--------------- ---------- -------------------- ------ ------- ---------------
T               TABLE      TBS_LJB                1152      24               9

SQL> ---随着 IDX_OBJ_ID 不断增大，区(EXTENT)及块（BLOCK）的个数也不断增多。如下：
SQL> select segment_name,
  2         segment_type,
  3         tablespace_name,
  4         blocks,
  5         extents,
  6         bytes/1024/1024
  7  from user_segments
  8  where segment_name = 'IDX_OBJ_ID';
SEGMENT_NAME    SEGMENT_TY TABLESPACE_NAME      BLOCKS EXTENTS BYTES/1024/1024
--------------- ---------- -------------------- ------ ------- ---------------
IDX_OBJ_ID      INDEX      TBS_LJB                 384      18               3
```

脚本 6-5　体会 Extents 及 Blocks 的增多

6.2.3 Tablespace

查看表空间的总体情况：

```
SQL>  SELECT A.TABLESPACE_NAME "表空间名",
  2         A.TOTAL_SPACE "总空间(G)",
  3         NVL(B.FREE_SPACE, 0) "剩余空间(G)",
  4         A.TOTAL_SPACE - NVL(B.FREE_SPACE, 0) "使用空间(G)",
  5             CASE WHEN A.TOTAL_SPACE=0 THEN 0 ELSE trunc(NVL(B.FREE_SPACE, 0) /
A.TOTAL_SPACE * 100, 2) END "剩余百分比%" --避免分母为 0
  6       FROM (SELECT TABLESPACE_NAME, trunc(SUM(BYTES) / 1024 / 1024/1024 ,2)
TOTAL_SPACE
  7             FROM DBA_DATA_FILES
  8            GROUP BY TABLESPACE_NAME) A,
  9              (SELECT TABLESPACE_NAME, trunc(SUM(BYTES / 1024 / 1024/1024 ),2)
FREE_SPACE
 10             FROM DBA_FREE_SPACE
 11            GROUP BY TABLESPACE_NAME) B
 12   WHERE A.TABLESPACE_NAME = B.TABLESPACE_NAME(+)
 13   ORDER BY 5;

表空间名                       总空间(G) 剩余空间(G) 使用空间(G) 剩余百分比%
------------------------------ ---------- ---------- ---------- ----------
SYSTEM                               1.04          0       1.04          0
SYSAUX                                .93        .05        .88       5.37
EXAMPLE                               .09        .02        .07      22.22
USERS                                5.96       1.77       4.19      29.69
TBS_BOSSWG_PDM                         16       7.98       8.02      49.87
UNDOTBS1                             2.67       2.63        .04       98.5
TBS_LJB_B                               8       7.99        .01      99.87
TBS_LJB_C                               8       7.99        .01      99.87
TBS_LJB                                 8       7.99        .01      99.87
TBS_CS                                 16      15.99        .01      99.93
TBS_LJB_A                             .15        .15          0       100

已选择 11 行。
```

脚本 6-6 查看表空间大小

6.2.4 rowid

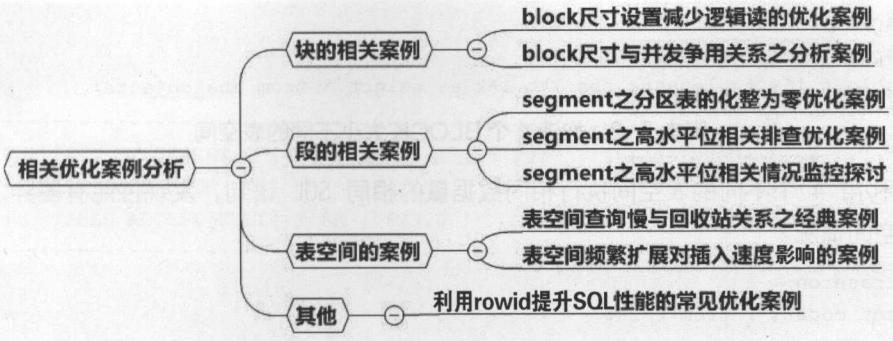
- rowid 一般由 18 位组成，如 000000FFFBBBBBBRRR，0 是对象 ID，F 是文件 ID，B 是块 ID，R 是行 ID。（注：Oracle 8 以上）。
- rowid 为一行的物理地址，当一行插入数据库块后，rowid 就唯一了，除非行物理移动，否则不变。
- rowid 不真正存在表数据块中，但是会存在索引中，方便根据索引中的 rowid 找到表数据。

```
drop table t purge;
create table t as select * from dba_objects;
select rowid from t where rownum=1;
ROWID
------------------
AAAYPJAAQAAATNDAAA
--以下可定位该行具体在哪个对象、文件、块、行（注：rowid 是 64 进制的）
data object number=AAAYPJ
file              =AAQ
block             =AAATND
row               =AAA
select dbms_rowid.rowid_object('AAAYPJAAQAAATNDAAA') data_object_id#,
       dbms_rowid.rowid_relative_fno('AAAYPJAAQAAATNDAAA') rfile#,
       dbms_rowid.rowid_block_number('AAAYPJAAQAAATNDAAA') block#,
       dbms_rowid.rowid_row_number('AAAYPJAAQAAATNDAAA') row#
 from dual;

DATA_OBJECT_ID#     RFILE#     BLOCK#       ROW#
--------------- ---------- ---------- ----------
          99273         16      78659          0
set linesize 266
col  owner format       a10
col  object_name format a20
col  file_name format   a40
col  tablespace_name format  a30

select owner,object_name from dba_objects where object_id=99273;
OWNER      OBJECT_NAME
---------- -----------
LJB             T

select file_name,tablespace_name from dba_data_files where file_id=16;
FILE_NAME                                TABLESPACE_NAME
---------------------------------------- ---------------
D:\ORACLE\ORADATA\TEST11G\TBS_LJB02.DBF      TBS_LJB
```

脚本 6-7　对 rowid 的体会与实践

6.3　相关优化案例分析

前面大家对逻辑结构有了一些直观的认识，接下来，我们来看看在现实应用中，如何巧妙地利用逻辑结构的块、段、表空间的一些知识和业务场景来进行 SQL 优化，让 SQL 跑得更快。如下图所示：



6.3.1　块的相关案例

环境准备（分别构造出 BLOCK 为 2KB、4KB、8KB、16KB 的 4 个表空间）：

```
--启动大小为 2KB 的块新建表空间(Windows 下只能使用 2KB、4KB、8KB 和 16KB)
alter system set db_2k_cache_size=100M;
drop tablespace tbs_ljb_2k including contents and datafiles;
create tablespace TBS_LJB_2k
blocksize 2K
datafile  'D:\ORACLE\ORADATA\TEST11G\TBS_LJB_2K_01.DBF' size 100M
autoextend on
extent management local
segment space management auto;
create table t_2k tablespace tbs_ljb_2k as select * from dba_objects;

--启动大小为 4KB 的块新建表空间
alter system set db_4k_cache_size=100M;
drop tablespace tbs_ljb_4k including contents and datafiles;
create tablespace TBS_LJB_4k
blocksize 4K
datafile  'D:\ORACLE\ORADATA\TEST11G\TBS_LJB_4K_01.DBF' size 100M
autoextend on
extent management local
segment space management auto;
create table t_4k tablespace tbs_ljb_4k as select * from dba_objects;

--启动大小为 8KB 的块新建表空间(默认就是 8KB)

drop table t_8k purge;
create table t_8k as select * from dba_objects;

--启动大小为 16KB 的块新建表空间
alter system set db_16k_cache_size=100M;
drop tablespace tbs_ljb_16k including contents and datafiles;
create tablespace TBS_LJB_16k
blocksize 16K
datafile  'D:\ORACLE\ORADATA\TEST11G\TBS_LJB_16K_01.DBF' size 100M
autoextend on
```

```
extent management local
segment space management auto;
create table t_16k tablespace tbs_ljb_16k as select * from dba_objects;
```

脚本 6-8　构造 4 个 BLOCK 大小不同的表空间

结果利用 4 个不同的表空间执行相同数据量的相同 SQL 语句，发现性能有差异，首先是 2KB 的表空间情况：

```
set autotrace on
SQL> select count(*) from t_2k;
COUNT(*)
--------
 111208
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |     1 |   891   (1)| 00:00:11 |
|   1 |  SORT AGGREGATE    |       |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T_2K  | 83292 |   891   (1)| 00:00:11 |
-----------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       4511  consistent gets
```

脚本 6-9　BLOCK 为 2KB 的 SQL 性能

4KB 的表空间情况：

```
set autotrace on
SQL> select count(*) from t_4k;
COUNT(*)
--------
 111208
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |     1 |   480   (1)| 00:00:06 |
|   1 |  SORT AGGREGATE    |       |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T_4K  | 63139 |   480   (1)| 00:00:06 |
-----------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       2137  consistent gets
```

脚本 6-10　BLOCK 为 4KB 的 SQL 性能

8KB 的表空间情况：

```
set autotrace on
SQL> select count(*) from t_8k;
```

```
COUNT(*)
--------
 111208
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |     1 |   291    (1)| 00:00:04 |
|   1 |  SORT AGGREGATE     |       |     1 |             |          |
|   2 |   TABLE ACCESS FULL| T_8K  | 62320 |   291    (1)| 00:00:04 |
-------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1043  consistent gets
```

脚本 6-11　BLOCK 为 8KB 的 SQL 性能

16KB 的表空间情况：

```
set autotrace on
SQL> select count(*) from t_16k;
COUNT(*)
--------
 111208
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |     1 |   200    (1)| 00:00:03 |
|   1 |  SORT AGGREGATE     |       |     1 |             |          |
|   2 |   TABLE ACCESS FULL| T_16K | 80144 |   200    (1)| 00:00:03 |
-------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        517  consistent gets
```

脚本 6-12　BLOCK 为 16KB 的 SQL 性能

试验结论：

BLOCK 越大，相同数据量的情况下存储的行就越多，BLOCK 需要的越少，而 Oracle 的最小访问单位是 BlOCK，所以访问产生的逻辑读就越小，对应的 consistent gets 就越小，如下图所示：

Block大小与性能			
Block大小	对应的Sql	表记录	consistent gets
2KB	select count(*) from t_2k;	111208	4511
4KB	select count(*) from t_4k;	111208	2137
8KB	select count(*) from t_8k;	111208	1043
16KB	select count(*) from t_16k;	111208	517

不过要切记，实际情况并非 BLOCK 越大越好，因为 BLOCK 越大，不同的人访问不同的数据落在同一个 BLOCK 的概率就大大增加，这就容易产生热点块竞争，因此在 OLTP 系统里 BLOCK 过大是不适合的，但是在 OLAP 系统里，BLOCK 大还是比较有用的！

6.3.2 段的相关案例

1. segment 之分区表的化整为零优化

普通表由单个段组成。可以将分区表理解为人们想把这个单个段切割成多个小段，如果访问能落到具体的某个小段里，则不要去理会别的小段，这样访问路径大幅度减少了，性能自然就增强了。请看下面的分段情况：

```
SQL> SET LINESIZE 666
SQL> set pagesize 5000
SQL> column segment_name format a20
SQL> column partition_name format a20
SQL> column segment_type format a20
SQL> select segment_name,
  2         partition_name,
  3         segment_type,
  4         bytes / 1024 / 1024 "字节数(M)",
  5         tablespace_name
  6    from user_segments
  7   where segment_name IN('RANGE_PART_TAB','NORM_TAB');

SEGMENT_NAME         PARTITION_NAME       SEGMENT_TYPE          字节数(M) TABLESPACE_NAME
-------------------------------------------------------------------------------------
NORM_TAB                                  TABLE                        47 USERS
RANGE_PART_TAB       P1                   TABLE PARTITION          .0625 USERS
RANGE_PART_TAB       P10                  TABLE PARTITION             5 USERS
RANGE_PART_TAB       P11                  TABLE PARTITION             4 USERS
RANGE_PART_TAB       P12                  TABLE PARTITION             5 USERS
RANGE_PART_TAB       P2                   TABLE PARTITION          .1875 USERS
RANGE_PART_TAB       P3                   TABLE PARTITION             4 USERS
RANGE_PART_TAB       P4                   TABLE PARTITION             4 USERS
RANGE_PART_TAB       P5                   TABLE PARTITION             5 USERS
RANGE_PART_TAB       P6                   TABLE PARTITION             4 USERS
RANGE_PART_TAB       P7                   TABLE PARTITION             4 USERS
RANGE_PART_TAB       P8                   TABLE PARTITION             5 USERS
RANGE_PART_TAB       P9                   TABLE PARTITION             4 USERS
RANGE_PART_TAB       P_MAX                TABLE PARTITION             8 USERS

已选择 14 行。
```

脚本 6-13　分区表的分段

接下来具体地查询通过 range_part_tab 表访问的执行计划，发现出现 Pstart 和 Pstop，由此我们就看得一清二楚了，访问只落在了第 9 个段里，也就是第 9 个分区中。

（1）普通表全扫描

```
SQL> select *
  2         from norm_tab
  3        where deal_date >= TO_DATE('2015-09-04', 'YYYY-MM-DD')
  4          and deal_date <= TO_DATE('2015-09-07', 'YYYY-MM-DD');
-------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |   223 |   443K|  1606   (1)| 00:00:20 |
|*  1 |  TABLE ACCESS FULL| NORM_TAB |   223 |   443K|  1606   (1)| 00:00:20 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       5923  consistent gets
```

脚本 6-14 普通表的全表扫描

（2）分区表局部扫描

```
SQL> select *
  2         from range_part_tab
  3        where deal_date >= TO_DATE('2015-09-04', 'YYYY-MM-DD')
  4          and deal_date <= TO_DATE('2015-09-07', 'YYYY-MM-DD');
-------------------------------------------------------------------
| Id |Operation             |Name          |Rows | Bytes|Cost (%CPU)| Time    |Pstart|Pstop |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT      |              |    1|  2037|    2   (0)| 00:00:01|      |      |
|   1| PARTITION RANGE SINGLE|              |    1|  2037|    2   (0)| 00:00:01|    9 |    9 |
|*  2|  TABLE ACCESS FULL   |RANGE_PART_TAB|    1|  2037|    2   (0)| 00:00:01|    9 |    9 |
-------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          3  consistent gets
```

脚本 6-15 分区表的局部分区扫描

数据量相同，SQL 语句相同，就是表结构不一样，逻辑读从 5923 迅速下降到 3，性能大幅度提升。

2. segment 之高水平位相关排查优化案例

```
--构造表
drop table t purge;
create table t as select * from dba_objects;
```

```
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
commit;
```

（1）未删表记录前逻辑读巨大

```
SQL> --测试表的大小及语句的效率
SQL> select bytes/1024/1024 from user_segments where segment_name='T';
BYTES/1024/1024
---------------
     264
SQL> select count(*) from t;
  COUNT(*)
---------------
   2332096
统计信息
--------
          0  recursive calls
          0  db block gets
      33350  consistent gets
          0  physical reads
```

脚本 6-16　未删除表记录前逻辑读巨大

（2）删大量记录后逻辑读不变

接下来删除大量数据，发现 SEGMENT 未见减少，依然是 264，关键是，表记录少了这么多，逻辑读依然不变，还是 33350，如下：

```
SQL> delete from t where rownum<=2000000;
已删除 2000000 行。
SQL> commit;
提交完成。
SQL> select bytes/1024/1024 from user_segments where segment_name='T';
BYTES/1024/1024
---------------
            264
SQL> select count(*) from t;
  COUNT(*)
----------
    332096
统计信息
--------
          0  recursive calls
          0  db block gets
      33350  consistent gets
          0  physical reads
```

脚本 6-17　删大量记录后逻辑读不变

（3）释放高水平位后性能飞跃

这就是高水平位的问题，用 move 重组数据后，高水平位释放，情况即将发生大变：

```
SQL> alter table t move;
表已更改。
SQL> select bytes/1024/1024 from user_segments where segment_name='T';
BYTES/1024/1024
---------------
    38
SQL> select count(*) from t;
  COUNT(*)
----------
    332096
统计信息
--------
          0  recursive calls
          0  db block gets
       4742  consistent gets
          0  physical reads
```

脚本 6-18　释放高水平位后性能飞跃

逻辑读从原先的 33350 降低为 4742，性能大幅度提升！

3. segment 之高水平位情况监控

接上面的问题，你在优化 SQL 的时候，你是如何知道系统中存在大表记录删除后，高水平位依然没释放的问题呢？

（1）num_rows 和 blocks 的正常比例

我们来构造一组试验，看看正常情况下从 user_tab_statistics 获取到的 num_rows 和 blocks 的正常比例，如下：

```
--构造表
drop table t purge;
create table t as select * from dba_objects;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
commit;
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'T',estimate_percent
=> 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;
select num_rows,blocks from user_tab_statistics where table_name='T';
 NUM_ROWS     BLOCKS
--------- ----------
  2320250      33583
```

脚本 6-19　num_rows 和 blocks 的正常比例

这里看出，num_rows 是 2320250，而 blocks 是 33583。2320250/33583 大致为 70 左右，表示 70 行装一个块，还算合理。

（2）num_rows 和 blocks 的异常比例

当删除了大量数据之后，情况又是如何呢？请继续看试验，如下：

```
set autotrace off
delete from t where rownum<=2300000;
commit;
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'T',estimate_percent
=> 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;

select num_rows,blocks from user_tab_statistics where table_name='T';
  NUM_ROWS     BLOCKS
---------- ----------
     32780      33583
```

脚本 6-20　num_rows 和 blocks 的异常比例

现在 num_rows 是 32780，而 blocks 是 33583，大致一行装一个块，这显然是不太可能的！

6.3.3　表空间的案例

1. 表空间查询慢与回收站关系

有一个案例，查询表空间语句非常慢，后来被查是由于回收站对象过多却没有释放的原因。这其实是 Oracle 执行计划中的一些 Bug，在 Oracle 查询表空间时，基表的查询主要是通过 NL 连接，对象过多则导致性能缓慢。具体细节就不说了，我们构造一个试验，让有兴趣的读者自行去体会。

首先是先建 1000 张表，如下：

```
sqlplus "/ as sysdba"
grant create any table to ljb;

connect ljb/ljb
drop table test purge;
create table test as select * from dba_objects where rownum<=2;

create or replace procedure p_create_tab
as
 l_sql VARCHAR2(32767);
BEGIN
  FOR i IN 1..10000
  LOOP
    l_sql := 'CREATE TABLE TEST_LJB_' || i ||'  as select * from test';
   -- dbms_output.put_line(l_sql);
   EXECUTE IMMEDIATE l_sql;
```

```
  END LOOP;
END p_create_tab;
/

create or replace procedure p_drop_tab
as
 l_sql VARCHAR2(32767);
BEGIN
  FOR i IN 1..10000
  LOOP
    l_sql := 'DROP TABLE TEST_LJB_' ||i;
   -- dbms_output.put_line(l_sql);
   EXECUTE IMMEDIATE l_sql;
  END LOOP;
END p_drop_tab;
/
```

脚本 6-21　构建 1000 张表

接下来通过执行 exec p_create_tab 完成建表 1000 张，并执行查询表空间语句，发现速度非常快，仅 0.12s，如下：

```
exec p_create_tab;
set timing on
 SELECT A.TABLESPACE_NAME "表空间名",
       A.TOTAL_SPACE "总空间(G)",
       NVL(B.FREE_SPACE, 0) "剩余空间(G)",
       A.TOTAL_SPACE - NVL(B.FREE_SPACE, 0) "使用空间(G)",
       CASE  WHEN  A.TOTAL_SPACE=0  THEN  0  ELSE  trunc(NVL(B.FREE_SPACE,  0)  /
A.TOTAL_SPACE * 100, 2) END "剩余百分比%" --避免分母为 0
  FROM (SELECT TABLESPACE_NAME, trunc(SUM(BYTES) / 1024 / 1024/1024 ,2) TOTAL_SPACE
          FROM DBA_DATA_FILES
         GROUP BY TABLESPACE_NAME) A,
       (SELECT TABLESPACE_NAME, trunc(SUM(BYTES / 1024 / 1024/1024  ),2) FREE_SPACE
          FROM DBA_FREE_SPACE
         GROUP BY TABLESPACE_NAME) B
 WHERE A.TABLESPACE_NAME = B.TABLESPACE_NAME(+)
 ORDER BY 5;
已用时间:  00:00:00.12
```

脚本 6-22　查询表空间速度很快

但是再执行 exec p_drop_tab;drop 这 1000 张表后，速度一下子就慢了（因为回收站里有太多对象了），花了 8s 多，如下：

```
exec p_drop_tab;
set timing on
 SELECT A.TABLESPACE_NAME "表空间名",
       A.TOTAL_SPACE "总空间(G)",
       NVL(B.FREE_SPACE, 0) "剩余空间(G)",
       A.TOTAL_SPACE - NVL(B.FREE_SPACE, 0) "使用空间(G)",
```

```
      CASE  WHEN  A.TOTAL_SPACE=0  THEN  0  ELSE  trunc(NVL(B.FREE_SPACE,  0)  /
A.TOTAL_SPACE * 100, 2) END "剩余百分比%" --避免分母为 0
  FROM (SELECT TABLESPACE_NAME, trunc(SUM(BYTES) / 1024 / 1024/1024 ,2) TOTAL_SPACE
          FROM DBA_DATA_FILES
         GROUP BY TABLESPACE_NAME) A,
       (SELECT TABLESPACE_NAME, trunc(SUM(BYTES / 1024 / 1024/1024  ),2) FREE_SPACE
          FROM DBA_FREE_SPACE
         GROUP BY TABLESPACE_NAME) B
 WHERE A.TABLESPACE_NAME = B.TABLESPACE_NAME(+)
 ORDER BY 5;

已用时间:  00:00:08.12
```

脚本 6-23　drop 大量表后，查询表空间速度变慢

2. 表空间频繁扩展与插入性能

环境搭建:分别建 2 个表空间，一个是固定尺寸的，另一个是自动扩展的，再分别在两个表空间上建相同结构的表，如下：

```
--- 分别建固定尺寸和自动扩展的两个表空间
set timing on
drop tablespace tbs_ljb_a including contents and datafiles;
drop tablespace tbs_ljb_b including contents and datafiles;
create tablespace TBS_LJB_A datafile  'D:\ORACLE\ORADATA\TEST11G\TBS_LJB_A.DBF' size
1M autoextend on uniform size 64k;
create tablespace TBS_LJB_B datafile  'D:\ORACLE\ORADATA\TEST11G\TBS_LJB_B.DBF' size
2G ;

---分别在两个不同表空间建表
connect ljb/ljb

set timing on
CREATE TABLE t_a (id int,contents varchar2(1000)) tablespace TBS_LJB_A;
CREATE TABLE t_b (id int,contents varchar2(1000)) tablespace TBS_LJB_B;
```

脚本 6-24　分别建固定尺寸和自动扩展的表空间及对应表

开始试验，分别往这两个表中插入数据：

```
---往可自动扩展表空间的表中插入数据
SQL> insert into t_a select rownum,LPAD('1', 1000, '*') from dual connect by
level<=200000;
已创建 200000 行。
已用时间:  00: 00: 52.41

---往固定大小的表空间的表中插入数据
SQL> insert into t_b select rownum,LPAD('1', 1000, '*') from dual connect by
level<=200000;
```

```
已创建 200000 行。
已用时间:  00: 00: 15.17
```

脚本 6-25 分别往两表插入相同量的数据

我们惊奇地发现，往固定尺寸的表空间的表中插入数据花了 15s，而往自动扩展的表空间的表中插入数据居然花了 52s！其本质原因就在于表空间初始分配的空间很小，而且每次新增的尺寸也很小。导致系统不断分配空间，自动扩展，多做事了。通过接下来的查询也可以一目了然地看出这一点，如下：

```
SQL> select count(*) from user_extents where segment_name='T_A';

  COUNT(*)
----------
      3610

SQL> select count(*) from user_extents where segment_name='T_B';

  COUNT(*)
----------
       100
```

脚本 6-26 两表的 extent 数量差异显著

T_A 表申请空间分配有 3610 次之多，而 T_B 表仅 100 次，性能差异能不大吗？

6.3.4 rowid

环境搭建，比较三种扫描方式的性能差异。

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id);
set linesize 1000
set autotrace traceonly
```

1. 方法 1（全表扫描）

```
SQL> select /*+full(t)*/ * from t where object_id=2;
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |   207 |   291   (1)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| T    |     1 |   207 |   291   (1)| 00:00:04 |
-------------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       1044  consistent gets
```

脚本 6-27 三种扫描方式之全表扫描

2. 方法 2（索引扫描）

```
SQL> select * from t where object_id=2;
-------------------------------------------------------------------------------------------
-------
|   0 | SELECT STATEMENT             |                |      1 |    207 |      2   (0)|
00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T              |      1 |    207 |      2   (0)|
00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_OBJECT_ID |      1 |        |      1   (0)|
00:00:01 |
-------------------------------------------------------------------------------------------
-------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
```

脚本 6-28　三种扫描方式之索引扫描

3. 方法 3（rowid 扫描）

```
SQL> select * from t where object_id=2 and rowid='AAAYiZAALAAAADLAAw';
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |      |      1 |    219 |      1   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY USER ROWID| T    |      1 |    219 |      1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          1  consistent gets
```

脚本 6-29　三种扫描方式之 rowid 扫描

扫描范围	扫描方式	SQL 写法	consistent gets
全扫描	方法1:全表扫描	select /*+full(t)*/ * from t where object_id=2;	1044
局部扫描	方法2:索引范围扫描	select * from t where object_id=2;	4
	方法3:rowid扫描	select * from t where object_id=2 and rowid='AAAYiZAALAAAADLAAw';	1

6.4　本章习题、总结与延伸

通过逻辑结构的调整来优化SQL		
原先怎样	调整了什么	变成怎样
某统计语句较慢	Block从2KB换成16KB	逻辑读从4511降低为517
某范围查询语句较慢	从普通表变成分区表	逻辑读从5923降低为3
某表删除大量记录后访问依然很慢	将高水平位释放了	逻辑读从33350降低为4742
某数据库的表空间查询结果返回很慢	将回收站对象大幅减少	表空间查询从 8 s 多提升到 0.1 s
某应用系统的表记录插入较慢	将频繁自动扩展的表空间改成了固定尺寸的	插入速度从 52 s 变成 15 s
某等值查询语句较慢	加索引改成索引扫描，再改变为rowid扫描	逻辑读从1044降低为4，再降低为1

习题 1：简要说说逻辑体系结构。

习题 2：说说 BLOCK 的大小设置问题。

习题 3：用自己的话说说，逻辑体系与 SQL 优化有什么关系。

习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

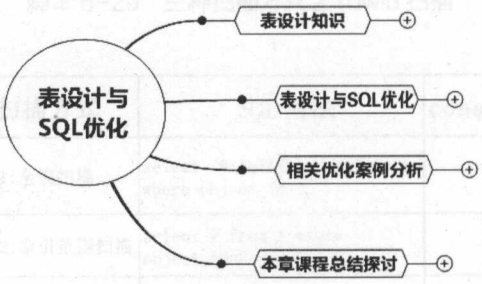
第 7 章　且慢，探寻表的设计让 SQL 飞

没想到表设计如此有用

表设计是一个非常重要的技术，这个技术使用得当，将会对应用优化起到巨大的作用。比如某张大表是按所在地区查询的，如果这表又按地区做分区，那 SQL 的性能马上就会大幅度提升，比如查福州的时候，就不会去访问厦门的其他 Segment，访问路径就会大大缩减。用一句专业点的话来描述就是：SQL 从全扫描转化成了局部扫描。当然，类似这样的例子很多，比如全局临时表、索引组织表等。这些是表的类型的设计，除此之外还有字段的设计，比如字段的类型、范式与反范式等。

如果我们只是站在表的特性上来研究，显然是不够的。我们还要紧抓业务，站在业务的层面来思考表设计，这才是最高境界。

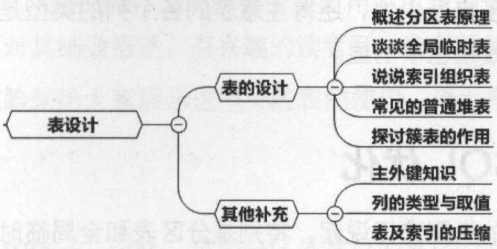
接下来我们从表设计知识、表设计与 SQL 优化、相关案例和总结 4 个方面展开本章的学习，总体学习思路如下图所示：



7.1　表设计

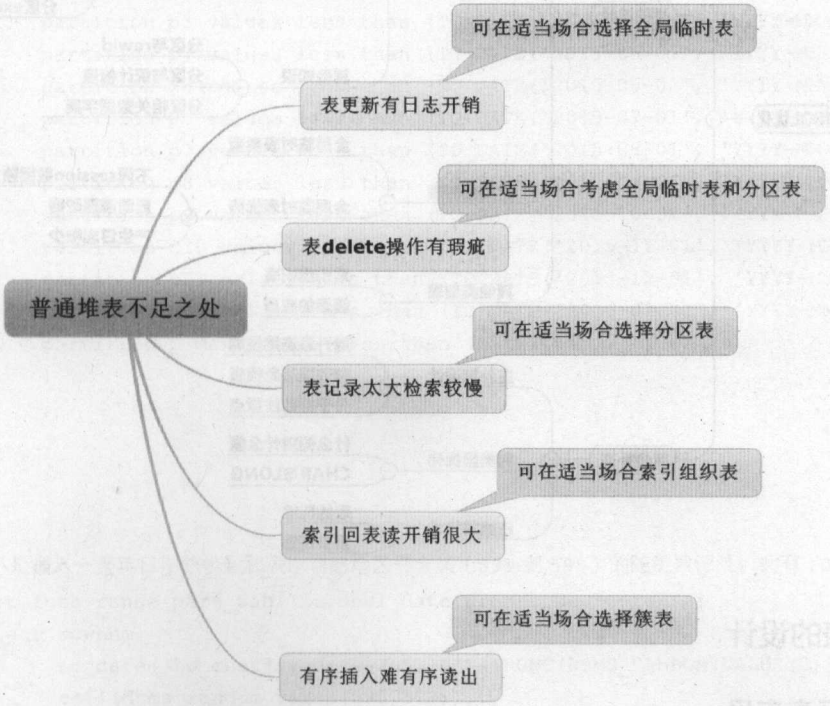
本节的大致内容分为表设计和补充模块。其中表设计主要从表的不同类型的选择展开讨

论，补充部分则从主外键、列设计等方面展开阐述，如下图所示：



7.1.1　表的设计

普通堆表是应用最为广泛的一种对象。不过由于其自身的缺陷从而让其他的一些“特殊对象”有了用武之地。比如表太大，检索数据困难，则可以考虑用分区表来将全扫描转换成较小范围的局部分区扫描；比如由于表的日志开销较大、delete 较耗性能等缺点，可考虑利用全局临时表日志开销极小和自动清理数据的特性；比如索引回表开销太大，可以考虑索引组织表避免索引回表；比如普通表的离散插入可导致顺序访问比较困难，则可以考虑用簇表等特性来解决。这些技术都是在特定的场合下的选择，往往对 SQL 优化很有帮助！



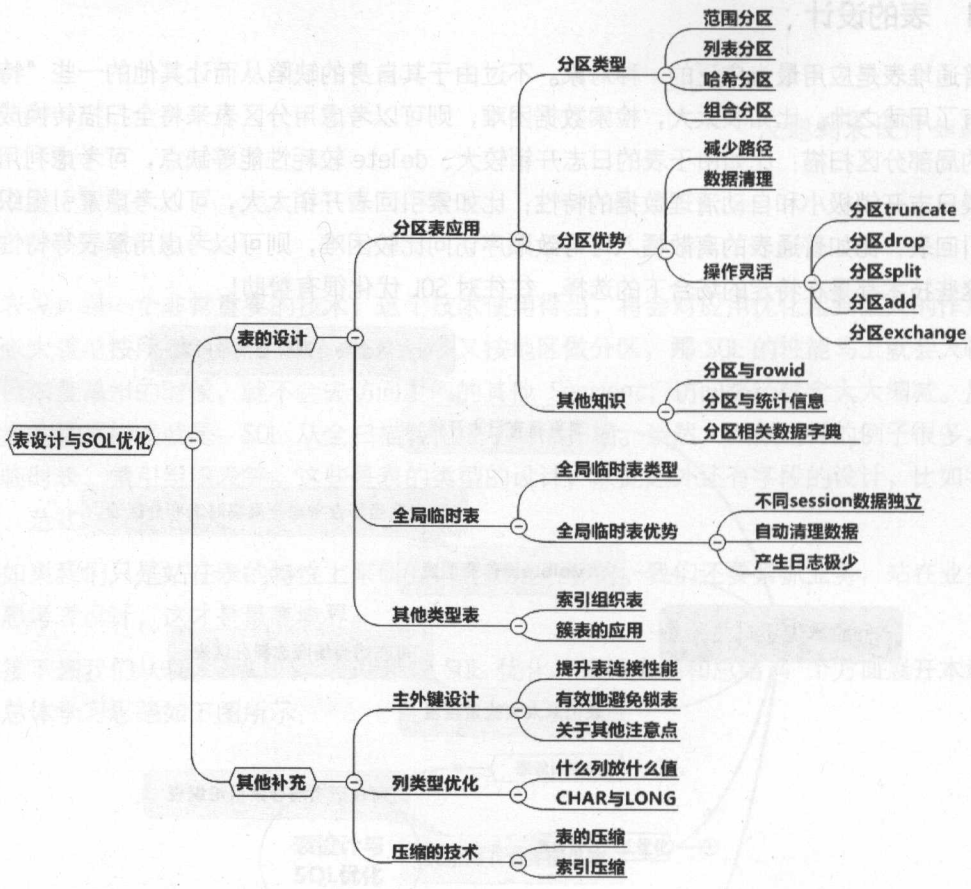
7.1.2　其他补充

除了上述由于普通堆表的不足而推出的系列“特殊对象”外，还有一些其他的表设计优化方式，比如表和表之间的主外键设计、表的列的类型的取值。还有就是针对表和索引本身进行

压缩，这样表占用的 Block 会减少，访问路径就减少，性能得以优化。不过关于表的压缩，除了注意使用场景要是查询多更新少外，还得注意表的各个列的类型是否相似，如果类型差异很大，压缩率将不高，优化效果也不明显。

7.2 表设计与 SQL 优化

关于表的设计需从表的类型选择说起，特别是分区表和全局临时表，被非常广泛地使用。以下是展开的详图：



7.2.1 表的设计

1. 分区表应用

（1）分区类型

现在开始描述分区表了，我们首先探讨分区表的类型及原理。分区表的类型有范围分区、列表分区、HASH 分区及组合分区 4 种，其中，范围分区应用最为广泛，需要重点学习和掌

握。而列表分区次之，在某些场合下也可以考虑使用组合分区（在 Oracle 11g 以前，组合分区的组合方式比较有限）。相对而言，HASH 分区在应用中适用的场景并不广泛，使用的频率比较低，故在这里暂且不对其细致描述，有兴趣的读者可以参考相关文档研究学习。

下面我们通过一组试验来给大家展示这些分区表的使用，请大家认真分析脚本。先来了解这些分区表的建立方法。

1）范围分区

首先看一组范围分区的例子，记住，范围分区最常见的是按时间列进行分区，如下：

```
SQL> drop table range_part_tab purge;

表已删除。

SQL> --注意，此分区为范围分区
SQL> create table range_part_tab (id number,deal_date date,area_code number,contents
varchar2(4000))
  2          partition by range (deal_date)
  3          (
  4          partition p1 values less than (TO_DATE('2015-02-01', 'YYYY-MM-DD')),
  5          partition p2 values less than (TO_DATE('2015-03-01', 'YYYY-MM-DD')),
  6          partition p3 values less than (TO_DATE('2015-04-01', 'YYYY-MM-DD')),
  7          partition p4 values less than (TO_DATE('2015-05-01', 'YYYY-MM-DD')),
  8          partition p5 values less than (TO_DATE('2015-06-01', 'YYYY-MM-DD')),
  9          partition p6 values less than (TO_DATE('2015-07-01', 'YYYY-MM-DD')),
 10          partition p7 values less than (TO_DATE('2015-08-01', 'YYYY-MM-DD')),
 11          partition p8 values less than (TO_DATE('2015-09-01', 'YYYY-MM-DD')),
 12          partition p9 values less than (TO_DATE('2015-10-01', 'YYYY-MM-DD')),
 13          partition p10 values less than (TO_DATE('2015-11-01', 'YYYY-MM-DD')),
 14          partition p11 values less than (TO_DATE('2015-12-01', 'YYYY-MM-DD')),
 15          partition p12 values less than (TO_DATE('2016-01-01', 'YYYY-MM-DD')),
 16          partition p_max values less than (maxvalue)
 17          )
 18          ;

表已创建。

SQL>
SQL> --以下是插入一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万条，如下：
SQL> insert into range_part_tab (id,deal_date,area_code,contents)
  2    select rownum,
  3           to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)),'J'),
  4           ceil(dbms_random.value(590,599)),
  5           rpad('*',400,'*')
  6      from dual
  7    connect by rownum <= 100000;

已创建 100000 行。
```

```
SQL> commit;

提交完成。
```

脚本 7-1　范围分区的例子

以上操作完成了范围分区的分区表建表，并且构造出 10 万条记录并将它们插入到分区表中，注意如下几点：

- 范围分区的关键字为 partition by range，即这三个关键字表示该分区为范围分区。
- values less than 是范围分区特定的语法，用于指明具体的范围，比如“partition p2 values less than (TO_DATE('2015-03-01', 'YYYY-MM-DD'))”，表示小于 3 月份的记录。
- partition p1 到 partition p_max 表示总共建立了 13 个分区。
- 最后还要注意 partition p_max values less than (maxvalue)部分，表示超出范围的记录全部落在这个分区中，免得出错。
- 分区表的分区可分别被指定在不同的表空间里，如果不写即为都在同一个默认表空间里。

请读者认真观察一分钟，再抽空运行笔者提供的这些脚本。

2）列表分区

接下来我们看一组列表分区的例子，表名虽然变为 list_part_tab，但是字段和插入记录的情况却是相同的，明显的区别在于建分区的字段不再是时间列，而是表示地区号的列。一般来说，列表分区最常见的分区列就是以地区列作为分区，如下：

```
SQL> drop table list_part_tab purge;
表已删除。
SQL> --注意，此分区为列表分区
SQL> create table list_part_tab (id number,deal_date date,area_code number,contents
varchar2(4000))
  2          partition by list (area_code)
  3          (
  4          partition p_591 values  (591),
  5          partition p_592 values  (592),
  6          partition p_593 values  (593),
  7          partition p_594 values  (594),
  8          partition p_595 values  (595),
  9          partition p_596 values  (596),
 10          partition p_597 values  (597),
 11          partition p_598 values  (598),
 12          partition p_599 values  (599),
 13          partition p_other values  (DEFAULT)
 14          )
 15          ;

表已创建。
```

```
SQL>
SQL>
SQL> --以下是插入一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万条，如下：
SQL> insert into list_part_tab (id,deal_date,area_code,contents)
  2    select rownum,
  3           to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)),'J'),
  4           ceil(dbms_random.value(590,599)),
  5           rpad('*',400,'*')
  6      from dual
  7    connect by rownum <= 100000;

已创建 100000 行。

SQL> commit;

提交完成。
```

脚本 7-2　列表分区的例子

以上操作完成了列表分区的分区表建表，构造出与之前相同的 10 万条记录并将它们插入到分区表中，同样需注意如下几点：

- 列表分区的关键字为 partition by list，即这三个关键字表示该分区为列表分区。
- 不同于之前范围分区的 values less than，列表分区仅需 values 即可确定范围。值得注意的是，partition p_592 values (592) 并不是说取值只能写一个，也可写为多个，比如 partition p_union values (592,593,594)。
- partition p_591 到 partition p_other 表示总共建立了 10 个分区。
- 最后还要注意 partition p_other values (default)部分，表示不在刚才 591～599 范围的记录全部落在这个默认分区中，避免应用出错。
- 分区表的分区可分别被指定在不同的表空间里，如果不写即为都在同一个默认表空间里。

3）散列分区

接下来我们继续看看散列分区（HASH 分区）的例子，和之前一样，表名虽然变为 hash_part_tab，但是字段和插入记录的情况却都是相同的。这次建散列分区时所取的列有意和之前范围分区的相同，都是时间列，如下：

```
SQL> drop table hash_part_tab purge;
表已删除。
SQL> --注意，此分区为 HASH 分区
SQL> create table hash_part_tab (id number,deal_date date,area_code number,contents
varchar2(4000))
  2          partition by hash (deal_date)
  3          PARTITIONS 12
  4          ;
表已创建。
```

```
SQL> --以下是插入 2012 年一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万
条，如下：
SQL> insert into hash_part_tab(id,deal_date,area_code,contents)
  2    select rownum,
  3           to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)),'J'),
  4           ceil(dbms_random.value(590,599)),
  5           rpad('*',400,'*')
  6      from dual
  7    connect by rownum <= 100000;

已创建 100000 行。
SQL> commit;
提交完成。
```

脚本 7-3　列表分区的例子

以上操作完成了散列分区表的建立和 10 万条记录的插入。是不是觉得这个散列分区好像比起之前的两种类型，要简单得多啊?

针对散列分区，我也说几个需要注意的重点，如下：

- 散列分区的关键字为 partition by hash，出现这三个关键字即表示当前分区为散列分区。
- 散列分区与之前两种分区的明显差别在于，没有指定分区名，而仅仅是指定了分区个数，如 PARTITIONS 12。
- 散列分区的分区个数尽量设置为偶数个，比如本例是 12 个分区，如果是 11 个或者 13 个就不妥了，具体原因和 Oracle 内部架构有关，这里不再细说。
- 可以指定散列分区的分区表空间，比如增加如下一小段：STORE IN (ts1,ts2,ts3,ts4,t5,t6,t7,t8,t9,t10,t11,t12)表示分别将它们存在 12 个不同的表空间里，当然不写出表空间就是都在同一个默认表空间里。

4）组合分区

接下来我们看看组合分区的例子。在 Oracle 11g 以前主要支持范围-列表和范围-散列这两种组合，在实际应用中最常用的组合是范围-列表（range-list）的组合，下面我们就来看一组与之相关的组合分区的试验，如下：

```
SQL> drop table range_list_part_tab purge;
表已删除。
SQL> --注意，此分区为范围分区
SQL> create table range_list_part_tab (id number,deal_date date,area_code number,
contents varchar2(4000))
  2          partition by range (deal_date)
  3            subpartition by list (area_code)
  4            subpartition TEMPLATE
  5            (subpartition p_591 values  (591),
  6             subpartition p_592 values  (592),
  7             subpartition p_593 values  (593),
```

```
 8            subpartition p_594 values  (594),
 9            subpartition p_595 values  (595),
10            subpartition p_596 values  (596),
11            subpartition p_597 values  (597),
12            subpartition p_598 values  (598),
13            subpartition p_599 values  (599),
14            subpartition p_other values (DEFAULT))
15          (
16           partition p1 values less than (TO_DATE('2015-02-01', 'YYYY-MM-DD')),
17           partition p2 values less than (TO_DATE('2015-03-01', 'YYYY-MM-DD')),
18           partition p3 values less than (TO_DATE('2015-04-01', 'YYYY-MM-DD')),
19           partition p4 values less than (TO_DATE('2015-05-01', 'YYYY-MM-DD')),
20           partition p5 values less than (TO_DATE('2015-06-01', 'YYYY-MM-DD')),
21           partition p6 values less than (TO_DATE('2015-07-01', 'YYYY-MM-DD')),
22           partition p7 values less than (TO_DATE('2015-08-01', 'YYYY-MM-DD')),
23           partition p8 values less than (TO_DATE('2015-09-01', 'YYYY-MM-DD')),
24           partition p9 values less than (TO_DATE('2015-10-01', 'YYYY-MM-DD')),
25           partition p10 values less than (TO_DATE('2015-11-01', 'YYYY-MM-DD')),
26           partition p11 values less than (TO_DATE('2015-12-01', 'YYYY-MM-DD')),
27           partition p12 values less than (TO_DATE('2016-01-01', 'YYYY-MM-DD')),
28           partition p_max values less than (maxvalue)
29          )
30          ;
表已创建。
SQL> --以下是插入一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万条，如下：
SQL> insert into range_list_part_tab(id,deal_date,area_code,contents)
  2    select rownum,
  3           to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)),'J'),
  4           ceil(dbms_random.value(590,599)),
  5           rpad('*',400,'*')
  6      from dual
  7    connect by rownum <= 100000;
已创建 100000 行。
SQL> commit;
提交完成。
```

脚本 7-4　组合分区的例子

以上试验完成了组合分区表（范围-列表）的建立和 10 万条记录的插入。大家看了貌似语法最简单的散列分区后，现在笔者给大家展现的是看上去写法最复杂的组合分区，是不是有些头昏脑胀？

不过大家仔细看看就会发现，其实主要也就增加了 subpartition by list (area_code)这个模块，其他部分和原先的范围分区没啥差异。这里大家要注意如下几个重点：

- 组合分区是由主分区和从分区组成的，比如范围-列表分区，就表示主分区是范围分区，而从分区是列表分区，从分区的关键字为 subpartition，比如本例中的 subpartition by list (area_code)。

- 为了避免在每个主分区中都写相同的从分区，可以考虑用模版方式，比如本例中的 subpartition TEMPLATE 关键字。
- 只要涉及子分区模块，都需要有 subpartition 关键字。
- 关于表空间和之前的没有差别，依然是可以指定，也可以不指定。

Oracle 11g 以前，除了支持上述范围-列表这个最常见的组合分区外，还支持范围-HASH 组合。在 Oracle 11g 后，还提供了 RANGE-RANGE、LIST-RANGE、LIST-HASH 和 LIST-LIST 这 4 种组合，这里我们就主要介绍范围-列表分区，其他组合适用的场合相对比较少，使用频率较低，就不再一一举例了。

接下来我们将在前面这些试验的基础上继续探讨分区表的原理以及分区表有哪些特性，并体会分区表设计到底能给我们带来什么好处。

5）分区原理

现在我们开始探讨分区表的原理是什么，实践出真知，我们再来看一组试验，最终结论还是从试验中得出。

前面建了 4 种不同类型的分区表并依次插入了相同的 10 万条记录，现在我们再建一张普通表，也插入相同的 10 万条记录，如下：

```
SQL> drop table norm_tab purge;
表已删除。
SQL> create table norm_tab (id number,deal_date date,area_code number,contents
varchar2(4000));
表已创建。
SQL> insert into norm_tab(id,deal_date,area_code,contents)
  2    select rownum,
  3           to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)),'J'),
  4           ceil(dbms_random.value(590,599)),
  5           rpad('*',400,'*')
  6      from dual
  7    connect by rownum <= 100000;
已创建 100000 行。
SQL> commit;
提交完成。
```

脚本 7-5　普通表也插入相同记录

接下来我们来观察普通表和分区表在段的分配上有何差异，这里仅以范围分区表来和普通表进行试验比较，如下：

```
SQL> SET LINESIZE 666
SQL> set pagesize 5000
SQL> column segment_name format a20
SQL> column partition_name format a20
SQL> column segment_type format a20
SQL> select segment_name,
```

```
  2         partition_name,
  3         segment_type,
  4         bytes / 1024 / 1024 "字节数(M)",
  5         tablespace_name
  6    from user_segments
  7   where segment_name IN('RANGE_PART_TAB','NORM_TAB');

SEGMENT_NAME     PARTITION_NAME       SEGMENT_TYPE         字节数(M) TABLESPACE_NAME
-------------------------------------------------------------------------------------
NORM_TAB                              TABLE                    47   TBS_LJB
RANGE_PART_TAB       P1               TABLE PARTITION           6   TBS_LJB
RANGE_PART_TAB       P2               TABLE PARTITION           4   TBS_LJB
RANGE_PART_TAB       P3               TABLE PARTITION           5   TBS_LJB
RANGE_PART_TAB       P4               TABLE PARTITION           4   TBS_LJB
RANGE_PART_TAB       P5               TABLE PARTITION           5   TBS_LJB
RANGE_PART_TAB       P6               TABLE PARTITION           4   TBS_LJB
RANGE_PART_TAB       P7               TABLE PARTITION           5   TBS_LJB
RANGE_PART_TAB       P8               TABLE PARTITION           4   TBS_LJB
RANGE_PART_TAB       P9               TABLE PARTITION           4   TBS_LJB
RANGE_PART_TAB       P10              TABLE PARTITION           5   TBS_LJB
RANGE_PART_TAB       P11              TABLE PARTITION           4   TBS_LJB
RANGE_PART_TAB       P12              TABLE PARTITION           3   TBS_LJB
RANGE_PART_TAB       P_MAX            TABLE PARTITION       .0625   TBS_LJB
已选择 14 行。
```

脚本 7-6　普通表和分区表的段的差异

比较看出普通表 NORM_TAB 就一个 SEGMENT，而分区表 RANGE_PART_TAB 则有 13 个 SEGMENT，这就是两者之间的差异。显而易见如果访问能落在指定的分区里，性能一定会比普通表的全表扫描快得多。

（2）分区优势

1）减少路径

分区表查询：

```
--观察范围分区表的分区消除带来的性能优势
set linesize 1000
set autotrace traceonly
set timing on
select *
      from range_part_tab
     where deal_date >= TO_DATE('2015-08-04', 'YYYY-MM-DD')
       and deal_date <= TO_DATE('2015-08-07', 'YYYY-MM-DD');
执行计划
-----------------------------------------------------------------------------------------
| Id |Operation             |Name          |Rows | Bytes |Cost (%CPU)| Time     |Pstart|Pstop|
-----------------------------------------------------------------------------------------
|   0|SELECT STATEMENT      |              |  958|  1917K|  171   (1)| 00:00:03 |      |     |
|   1| PARTITION RANGE SINGLE|              |  958|  1917K|  171   (1)| 00:00:03 |    8 |    8|
```

```
|*  2|  TABLE ACCESS FULL    |RANGE_PART_TAB|  958|  1917K|  171   (1)| 00:00:03 |    8 |    8|
-------------------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        657  consistent gets
          0  physical reads
          0  redo size
      42053  bytes sent via SQL*Net to client
       1240  bytes received via SQL*Net from client
         77  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1136  rows processed
```

脚本 7-7　分区表分区消除

普通表查询：

```
--比较相同语句，普通表无法用到 DEAL_DATE 条件进行分区消除的情况
select *
      from norm_tab
     where deal_date >= TO_DATE('2015-08-04', 'YYYY-MM-DD')
       and deal_date <= TO_DATE('2015-08-07', 'YYYY-MM-DD');

执行计划
----------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |   599 |  1199K|  1709   (1)| 00:00:21 |
|*  1 |  TABLE ACCESS FULL| NORM_TAB |   599 |  1199K|  1709   (1)| 00:00:21 |
----------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       6373  consistent gets
          0  physical reads
          0  redo size
      42222  bytes sent via SQL*Net to client
       1240  bytes received via SQL*Net from client
         77  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1137  rows processed
```

脚本 7-8　普通表全表扫描

普通表的逻辑读是 6373，而分区表是 657，分区消除后性能得以大幅度提升。

2）数据清理

```
connect ljb/ljb
drop table part_tab_trunc purge;
create table part_tab_trunc (id int,col2 int,col3 int,contents varchar2(4000))
        partition by range (id)
        (
        partition p1 values less than (10000),
        partition p2 values less than (20000),
        partition p3 values less than (maxvalue)
        )
        ;
insert into part_tab_trunc select rownum ,rownum+1,rownum+2, rpad('*',400,'*') from
dual connect by rownum <=50000;
commit;

--truncate 分区前
select count(*) from part_tab_trunc partition(p1);
 COUNT(*)
---------
  9999

alter table part_tab_trunc truncate partition p1 ;

--truncate 分区后
select count(*) from part_tab_trunc partition(p1);
 COUNT(*)
---------
    0
```

脚本 7-9　分区表数据清理

Drop 分区：

```
--分区表的 drop
sqlplus ljb/ljb
drop table part_tab_drop purge;
create table part_tab_drop (id int,col2 int ,col3 int,contents varchar2(4000))
        partition by range (id)
        (
        partition p1 values less than (10000),
        partition p2 values less than (20000),
        partition p3 values less than (maxvalue)
        )
        ;
insert into part_tab_drop select rownum ,rownum+1,rownum+2,rpad('*',400,'*') from
dual connect by rownum <=50000;
commit;

--drop 分区前
select partition_name, segment_type, bytes
```

```
  from user_segments
 where segment_name ='PART_TAB_DROP';

PARTITION_NAME                 SEGMENT_TYPE            BYTES
------------------------------ ------------------ ----------
P1                             TABLE PARTITION       5242880
P2                             TABLE PARTITION       5242880
P3                             TABLE PARTITION      15728640


alter table part_tab_drop drop partition p1 ;

--drop 分区后

select partition_name, segment_type, bytes
  from user_segments
 where segment_name ='PART_TAB_DROP';

PARTITION_NAME                 SEGMENT_TYPE            BYTES
------------------------------ ------------------ ----------
P2                             TABLE PARTITION       5242880
P3                             TABLE PARTITION      15728640
```

脚本 7-10　分区表 drop 分区

3）操作灵活

Split 分区：

```
---分区表的 SPLIT
drop table part_tab_split purge;
create table part_tab_split (id int,col2 int ,col3 int ,contents varchar2(4000))
        partition by range (id)
        (
        partition p1 values less than (10000),
        partition p2 values less than (20000),
        partition p_max values less than (maxvalue)
        )
        ;
insert into part_tab_split select rownum ,rownum+1,rownum+2,rpad('*',400,'*') from
dual connect by rownum <=90000;
commit;

--split 分区前
select partition_name, segment_type, bytes
  from user_segments
 where segment_name ='PART_TAB_SPLIT';

PARTITION_NAME                 SEGMENT_TYPE            BYTES
------------------------------ ------------------ ----------
```

```
P1                              TABLE PARTITION       5242880
P2                              TABLE PARTITION       5242880
P_MAX                           TABLE PARTITION      34603008

已选择 3 行。

alter table part_tab_split SPLIT PARTITION P_MAX at (30000) into (PARTITION
p3 ,PARTITION P_MAX);
alter table part_tab_split SPLIT PARTITION P_MAX at (40000) into (PARTITION
p4 ,PARTITION P_MAX);
alter table part_tab_split SPLIT PARTITION P_MAX at (50000) into (PARTITION
p5 ,PARTITION P_MAX);
alter table part_tab_split SPLIT PARTITION P_MAX at (60000) into (PARTITION
p6 ,PARTITION P_MAX);
alter table part_tab_split SPLIT PARTITION P_MAX at (70000) into (PARTITION
p7 ,PARTITION P_MAX);

--split 分区后
select partition_name, segment_type, bytes
  from user_segments
 where segment_name ='PART_TAB_SPLIT';

PARTITION_NAME                  SEGMENT_TYPE            BYTES
------------------------------ ------------------ ----------
P1                              TABLE PARTITION       5242880
P2                              TABLE PARTITION       5242880
P3                              TABLE PARTITION       5242880
P4                              TABLE PARTITION       5242880
P5                              TABLE PARTITION       5242880
P6                              TABLE PARTITION       5242880
P7                              TABLE PARTITION       5242880
P_MAX                           TABLE PARTITION      10485760

已选择 8 行。
```

脚本 7-11　分区表 split 分区

Add 分区：

```
drop table part_tab_add purge;
create table part_tab_add (id int,col2 int,col3 int,contents varchar2(4000))
        partition by range (id)
        (
        partition p1 values less than (10000),
        partition p2 values less than (20000),
        partition p3 values less than (30000),
        partition p4 values less than (40000),
        partition p5 values less than (50000),
        partition p_max values less than (maxvalue)
```

```
        )
        ;
insert into part_tab_add select rownum ,rownum+1,rownum+2,rpad('*',400,'*') from dual
connect by rownum <=45000;
commit;

--add 分区前
select partition_name, segment_type, bytes
  from user_segments
 where segment_name ='PART_TAB_ADD';

PARTITION_NAME                 SEGMENT_TYPE            BYTES
------------------------------ ------------------ ----------
P1                             TABLE PARTITION       5242880
P2                             TABLE PARTITION       5242880
P3                             TABLE PARTITION       5242880
P4                             TABLE PARTITION       5242880
P5                             TABLE PARTITION       3145728
P_MAX                          TABLE PARTITION         65536

已选择 6 行。

--注意：必须要把默认分区去掉，再 add 分区，再增加默认分区,这里可能丢数据!
alter table part_tab_add  drop partition p_max;
alter table part_tab_add  add PARTITION p6 values less than (60000);
alter table part_tab_add  add PARTITION p_max  values less than (maxvalue);


--add 分区后
select partition_name, segment_type, bytes
  from user_segments
 where segment_name ='PART_TAB_ADD';

PARTITION_NAME                 SEGMENT_TYPE            BYTES
------------------------------ ------------------ ----------
P1                             TABLE PARTITION       5242880
P2                             TABLE PARTITION       5242880
P3                             TABLE PARTITION       5242880
P4                             TABLE PARTITION       5242880
P5                             TABLE PARTITION       3145728
P6                             TABLE PARTITION         65536
P_MAX                          TABLE PARTITION         65536

已选择 7 行。
```

脚本 7-12　分区表 ADD 分区

exchange 分区：

```
--该操作会导致全局索引失效（要考虑 including indexes 或者重建索引），具体见索引章节

--分区表的 exchange
connect ljb/ljb
drop table part_tab_exch purge;
create table part_tab_exch (id int,col2 int,col3 int,contents varchar2(4000))
        partition by range (id)
        (
         partition p1 values less than (10000),
         partition p2 values less than (20000),
         partition p3 values less than (30000),
         partition p4 values less than (40000),
         partition p5 values less than (50000),
         partition p_max values less than (maxvalue)
        )
        ;
insert into part_tab_exch select rownum ,rownum+1,rownum+2, rpad('*',400,'*') from
dual connect by rownum <=60000;
commit;

create index idx_part_exch_col2 on part_tab_exch(col2) local;
create index idx_part_exch_col3 on part_tab_exch (col3);


--分区表的 EXCHANGE（某分区和普通表之间的数据进行交换）
drop table normal_tab purge;
create table normal_tab (id int,col2 int,col3 int,contents varchar2(4000));
create index idx_norm_col2  on normal_tab (col2);


--交换前数据情况
select count(*) from normal_tab;
  COUNT(*)
----------
     0
select count(*) from part_tab_exch partition(p1);
  COUNT(*)
----------
   999

--其中 including indexes  可选，为了保证全局索引不要失效
alter table part_tab_exch exchange partition p1 with table normal_tab including
indexes update global indexes;

--交换后数据情况
select count(*) from normal_tab;
  COUNT(*)
----------
    999
```

```
select count(*) from part_tab_exch partition(p1);
 COUNT(*)
----------
     0
```

脚本 7-13　分区表 EXCHANGE 分区

（3）其他知识

1）分区与 rowid

分区表的分区数据转移后会更改 ROWID（因为到了不同的分区后，所在的 segment 变了）。

构造环境：

```
---分区表的分区数据转移后会更改 ROWID
set autotrace off
drop table part_tab_rowid purge;
create table part_tab_rowid (id int,col2 int,col3 int,contents varchar2(4000))
        partition by range (id)
        (
        partition p1 values less than (10),
        partition p2 values less than (20),
        partition p_max values less than (maxvalue)
        )
        ;
insert into part_tab_rowid select rownum ,rownum+1,rownum+2,rpad('*',400,'*') from
dual connect by rownum <=30;
commit;


select t.id,t.rowid from part_tab_rowid partition (p1) t;

        ID ROWID
---------- ------------------
         1 AAAZdQAAGAAATxjAAA
         2 AAAZdQAAGAAATxjAAB
         3 AAAZdQAAGAAATxjAAC
         4 AAAZdQAAGAAATxjAAD
         5 AAAZdQAAGAAATxjAAE
         6 AAAZdQAAGAAATxjAAF
         7 AAAZdQAAGAAATxjAAG
         8 AAAZdQAAGAAATxjAAH
         9 AAAZdQAAGAAATxjAAI

已选择 9 行。

select t.id,t.rowid from part_tab_rowid partition (p2) t;

           ID ROWID
---------- ------------------
        10 AAAZdRAAGAAATxrAAA
```

```
        11 AAAZdRAAGAAATxrAAB
        12 AAAZdRAAGAAATxrAAC
        13 AAAZdRAAGAAATxrAAD
        14 AAAZdRAAGAAATxrAAE
        15 AAAZdRAAGAAATxrAAF
        16 AAAZdRAAGAAATxrAAG
        17 AAAZdRAAGAAATxrAAH
        18 AAAZdRAAGAAATxrAAI
        19 AAAZdRAAGAAATxrAAJ

已选择 10 行。


select dbms_rowid.rowid_object('AAAZdQAAGAAATxjAAA') data_object_id#,
           dbms_rowid.rowid_relative_fno('AAAZdQAAGAAATxjAAA') rfile#,
           dbms_rowid.rowid_block_number('AAAZdQAAGAAATxjAAA') block#,
           dbms_rowid.rowid_row_number('AAAZdQAAGAAATxjAAA') row# from dual;

DATA_OBJECT_ID#     RFILE#     BLOCK#        ROW#
--------------- ---------- ---------- ----------
         104272          6      80995          0
```

接下来执行如下：

```
--以下这个 enable row movement 必须操作，否则会出现 ORA-014402:更新分区关键字列导致分区更改
alter table part_tab_rowid   enable row movement;
update part_tab_rowid set id=12 where id=1;
commit;
```

发现行的 rowid 发生了改变：

```
select t.id,t.rowid from part_tab_rowid partition(p1) t;
        ID ROWID
---------- ------------------
         2 AAAZdQAAGAAATxjAAB
         3 AAAZdQAAGAAATxjAAC
         4 AAAZdQAAGAAATxjAAD
         5 AAAZdQAAGAAATxjAAE
         6 AAAZdQAAGAAATxjAAF
         7 AAAZdQAAGAAATxjAAG
         8 AAAZdQAAGAAATxjAAH
         9 AAAZdQAAGAAATxjAAI

已选择 8 行。

SQL> select t.id,t.rowid from part_tab_rowid partition(p2) t;
        ID ROWID
---------- ------------------
        10 AAAZdRAAGAAATxrAAA
        11 AAAZdRAAGAAATxrAAB
        12 AAAZdRAAGAAATxrAAC
```

```
        13 AAAZdRAAGAAATxrAAD
        14 AAAZdRAAGAAATxrAAE
        15 AAAZdRAAGAAATxrAAF
        16 AAAZdRAAGAAATxrAAG
        17 AAAZdRAAGAAATxrAAH
        18 AAAZdRAAGAAATxrAAI
        19 AAAZdRAAGAAATxrAAJ
        12 AAAZdRAAGAAATxrAAK
已选择 11 行。
```

脚本 7-14　分区表分区转移与 rowid

而普通表的变更是不会引起变更 rowid 的，这里就不做试验了，请读者自行体会。

2）分区与统计信息

- 传统的 analyze 方式对收集分区表统计信息不准确。
- 可以只收集某分区的统计信息。

准备脚本略去，请看如下：

```
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'RANGE_PART_TAB',
partname =>'p_201512', estimate_percent => 10,method_opt=> 'for all indexed
columns',cascade=>TRUE) ;

set linesize 366
col partition_count format 99999
col num_rows format 99999999
col partition_name format a28
col table_name format a30
col last_analyzed format date

---收集完 p_201512 分区的统计结构后，可以通过如下方式查询到结果：
select table_name,
       partition_name,
       last_analyzed,
       partition_position,
       num_rows
  from user_tab_statistics t
 where table_name ='RANGE_PART_TAB';

 TABLE_NAME          PARTITION_NAME     LAST_ANALYZED  PARTITION_POSITION  NUM_ROWS
------------------- ------------------ -------------- ------------------ ---------
RANGE_PART_TAB                         01-12 月-13                          201230
RANGE_PART_TAB      P_201501                                          1
RANGE_PART_TAB      P_201502                                          2
RANGE_PART_TAB      P_201503                                          3
RANGE_PART_TAB      P_201504                                          4
RANGE_PART_TAB      P_201505                                          5
RANGE_PART_TAB      P_201506                                          6
RANGE_PART_TAB      P_201507                                          7
```

```
RANGE_PART_TAB      P_201508                                    8
RANGE_PART_TAB      P_201509                                    9
RANGE_PART_TAB      P_201510                                   10
RANGE_PART_TAB      P_201511                                   11
RANGE_PART_TAB      P_201512            01-12 月-13            12       8474
RANGE_PART_TAB      P_201601                                   13
RANGE_PART_TAB      P_201602                                   14
RANGE_PART_TAB      P_MAX                                      15

已选择 16 行。
```

脚本 7-15　分区与统计信息

3）分区相关数据字典

该表是否是分区表，分区表的分区类型是什么，是否有子分区，分区总数有多少。如下：

```
select partitioning_type,
       subpartitioning_type,
       partition_count
  from user_part_tables
 where table_name ='RANGE_PART_TAB';
```

脚本 7-16　查询分区表信息

该分区表在哪一列上建分区,有无多列联合建分区。如下：

```
select column_name,
       object_type,
       column_position
  from user_part_key_columns
 where name ='RANGE_PART_TAB';
```

脚本 7-17　查询分区表哪列建分区

该分区表有多大？如下：

```
select sum(bytes) / 1024 / 1024
  from user_segments
 where segment_name ='RANGE_PART_TAB';
```

脚本 7-18　查询分区表尺寸

该分区表各分区分别有多大，各个分区名是什么。如下：

```
select partition_name,
       segment_type,
       bytes
  from user_segments
 where segment_name ='RANGE_PART_TAB';
```

脚本 7-19　查询分区表各分区的大小与分区名

该分区表的统计信息收集情况。如下：

```
select table_name,
```

```
       partition_name,
       last_analyzed,
       partition_position,
       num_rows
  from user_tab_statistics t
 where table_name ='RANGE_PART_TAB';
```

脚本 7-20　查询分区表统计信息收集情况

查该分区表有无索引，分别什么类型，全局索引是否失效，此外还可看统计信息收集情况。如下：

```
select table_name,
       index_name,
       last_analyzed,
       blevel,
       num_rows,
       leaf_blocks,
       distinct_keys,
       status
  from user_indexes
 where table_name ='RANGE_PART_TAB';
```

脚本 7-21　查询分区表索引情况

该分区表在哪些列上建了索引。如下：

```
select index_name,
       column_name,
       column_position
  from user_ind_columns
 where table_name = 'RANGE_PART_TAB';
```

脚本 7-22　查询分区表在哪些列有索引

该分区表上的各索引分别有多大。如下：

```
select segment_name,segment_type,sum(bytes)/1024/1024
  from user_segments
 where segment_name in
       (select index_name
          from user_indexes
         where table_name ='RANGE_PART_TAB')
group by segment_name,segment_type ;
```

脚本 7-23　查询分区表各索引大小

该分区表的索引段的分配情况。如下：

```
select segment_name
       partition_name,
       segment_type,
       bytes
  from user_segments
```

```
 where segment_name in
       (select index_name
          from user_indexes
         where table_name ='RANGE_PART_TAB');
```

脚本 7-24　查询分区表索引段的分配情况

查看分区索引相关信息及统计信息，是否失效。如下：

```
select t2.table_name,
       t1.index_name,
       t1.partition_name,
       t1.last_analyzed,
       t1.blevel,
       t1.num_rows,
       t1.leaf_blocks,
       t1.status
  from user_ind_partitions t1, user_indexes t2
where t1.index_name = t2.index_name
   and t2.table_name='RANGE_PART_TAB';
```

脚本 7-25　查询分区表索引相关统计信息

2. 全局临时表

（1）全局临时表类型

全局临时表分为两种类型：一种是基于会话的全局临时表（commit preserve rows）；一种是基于事务的全局临时表（on commit delete rows）。具体建表语法见下面例子：

```
SQL> drop table t_tmp_session purge;
表已删除。
SQL> drop table t_tmp_transaction purge ;
表已删除。
SQL> create global temporary table T_TMP_session on commit preserve rows as select  *
from dba_objects where 1=2;
表已创建。
SQL> select table_name,temporary,duration from user_tables  where table_name='T_TMP_
SESSION';
TABLE_NAME           T DURATION
---------------------------- - -------------------------
T_TMP_SESSION        Y SYS$SESSION
SQL> create global temporary table t_tmp_transaction on commit delete rows as select
* from dba_objects where 1=2;
表已创建。
SQL> select table_name, temporary, DURATION from user_tables   where table_name=
'T_TMP_TRANSACTION';
TABLE_NAME             T   DURATION
---------------------------- - ---------------------------------------
T_TMP_TRANSACTION      Y   SYS$TRANSACTION
```

脚本 7-26　全局临时表的两种类型

（2）全局临时表优势

1）高效删除记录

全局临时表最重要的特点有两个。一是高效删除记录，基于事务的全局临时表 COMMIT 或者 SESSION 连接退出后，临时表记录自动被删除；基于会话的全局临时表则是在 SESSION 连接退出后，临时表记录自动被删除，都无须我们动手去操作。二是针对不同会话数据独立，不同的 SESSION 访问全局临时表，看到的结果不同。

这两点既是全局临时表最重要的特点，也是最有用的特点，在工作中某些特定的场合，它们发挥出了巨大的作用。下面我们先从全局临时表的高效删除开始学习。

首先我们看看基于事务的全局临时表 COMMIT 后，记录是否已经被删除了，并查看产生日志的情况，如下：

```
SQL> select count(*) from t_tmp_transaction;
COUNT(*)
----------
      0
SQL> select * from v_redo_size;
NAME                          VALUE
---------------------------------------------------------------------------
redo size                         0
SQL> insert into t_tmp_transaction select * from dba_objects;
已创建 55721 行。
SQL> select * from v_redo_size;
NAME                                VALUE
---------------------------------------------------------------------------
redo size                                303976
SQL> commit;
提交完成。
SQL> select * from v_redo_size;
NAME                                VALUE
---------------------------------------------------------------------------
redo size                                304116
SQL> select count(*) from t_tmp_transaction;
COUNT(*)
----------
      0
```

脚本 7-27　基于事务的全局临时表的数据清理

可以发现，基于事务的全局临时表中插入了 55721 行，COMMIT 后，再查这个表时记录就没了。还有，用 COMMIT 方式删除全局临时表记录所产生的日志量才 304116-303976=140，比起之前直接用 delete 方式操作产生的日志量 16541396，几乎可以忽略不计了。

140 这个日志量其实是 COMMIT 动作本身产生的，所以我们基本可以理解为全局临时表的 COMMIT 或者退出 SESSION 的方式不会产生日志，这是 Oracle 的全局临时表的一种特性。

接下来我们再看看基于 SESSION 的全局临时表的运行情况，如下：

```
SQL> select * from v_redo_size;
NAME                          VALUE
----------------------------------------------------------------
redo size                     0
SQL> insert  into  t_tmp_session select * from dba_objects;
已创建 55721 行。
SQL> select * from v_redo_size;
NAME                          VALUE
----------------------------------------------------------------
redo size                     304036
SQL> commit;
提交完成。
SQL> select count(*) from t_tmp_session;
  COUNT(*)
----------
     55721
SQL> select * from v_redo_size;
NAME                          VALUE
----------------------------------------------------------------
redo size                     304176
```

脚本 7-28　基于 SESSION 的全局临时表的数据清理

现在，大家会发现，基于会话的全局临时表在 COMMIT 后表的记录依然存在，这和基于事务的临时表真是不一样。

那我们再观察一下，看看退出 SESSION，再重新连接后，记录还在吗？

```
SQL> exit
C:\Users\ljb>sqlplus ljb/ljb
SQL> select count(*) from  t_tmp_session;
  COUNT(*)
----------
      0
```

不用说大家也看得很清楚了，记录真的没有了。我们已经知道了基于事务和基于会话（SESSION）的全局临时表的差别了，那什么时候用基于事务的，什么时候用基于 SESSION 的呢？

如果全局临时表在程序的一次调用执行过程中需要多次清空记录再插入记录，就要考虑用基于事务的，这时 COMMIT 可以把结果快速清理了，否则用 DELETE 效率低下。如果不存在这种情况，就用基于 SESSION 的，这样更简单，连 COMMIT 的动作都省了。一般来说，基于 SESSION 的全局临时表的应用会更多一些，少数比较复杂的应用，涉及一次调用中需要记录清空再插入等复杂动作时，才考虑用基于事务的全局临时表。

笔者在实际工作中经常使用到全局临时表，并且全局临时表发挥了巨大的作用。在后面笔者还会分享与全局临时表有关的经典案例。接下来我们来看看全局临时表的第二个重要特点，

针对不同会话独立的有用特性。

2）不同会话独立

下面我们讨论全局临时表时，暂且就只讨论基于 SESSION 的临时表而不讨论基于事务的全局临时表了，因为它们基本上没什么大的差异。

首先进入一个 SID=975 的 SESSION，完成全局临时表的插入，查看当前记录，具体如下：

```
C:\Users\ljb>sqlplus ljb/ljb
SQL> select * from v$mystat where rownum=1;
       SID STATISTIC#      VALUE
---------- ---------- ----------
       975          0          1
SQL> select * from t_tmp_session;
未选定行
SQL> insert  into  t_tmp_session select * from dba_objects;
已创建 55721 行。
SQL> commit;
提交完成。
SQL> select count(*) from t_tmp_session;
  COUNT(*)
----------
   55721
```

接下来再登录一个 SID=973 的新连接，发现同样是这个 t_tmp_session 表，记录却为 0，对该表插入一条，在当前 SESSION 中我们查询到该表就是一条记录。

```
C:\Users\ljb>sqlplus ljb/ljb
SQL> select * from v$mystat where rownum=1;
  SID    STATISTIC#    VALUE
----------------------------------------------
  973       0          1
SQL> select count(*) from t_tmp_session;
  COUNT(*)
----------
       0
SQL> insert into t_tmp_session select * from  dba_objects where rownum=1;
已创建 1 行。
SQL> commit;
提交完成。
SQL> select count(*) from t_tmp_session;
  COUNT(*)
----------
      1
```

脚本 7-29　全局临时表不同的 session 数据独立

假如这时我们回到 SID=975 的 SESSION 再去查看 t_tmp_session 表，会发现那个 SESSION 查询的依然是 55721 条。

这是一个神奇的特性，结合高效删除和灵活应用这两个特性，它将会给相关工作带来巨大的帮助。大家将会在后续的章节中感受到这一点。

7.2.2　其他补充

1. 主外键设计

一般有一个原则，两个表有主外键关系时，外键的列上要有索引，这样有两大好处：1.两表关联查询一般更高效；2.外键所在的表更新不容易产生死锁。

2. 压缩的技术

（1）表的压缩

压缩表可减少表的尺寸，因此占更少的 BLOCK，从而减少 IO。

未压缩前：

```
DROP TABLE t purge;

CREATE TABLE t NOCOMPRESS AS
SELECT rownum AS n, rpad(' ',500,mod(rownum,15)) AS pad
FROM dual
CONNECT BY level <= 200000;

execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t');

--未压缩的表当前情况
SELECT table_name, blocks,compression  FROM user_tables WHERE table_name = 'T';
TABLE_NAME                      BLOCKS COMPRESS
------------------------------ ---------- --------
T                                14449 DISABLED
set autotrace traceonly
select count(*) from t;

执行计划
----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |  3922   (1)| 00:00:48 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    |   200K|  3922   (1)| 00:00:48 |
----------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      14297  consistent gets
          0  physical reads
```

```
          0  redo size
        423  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 7-30　表压缩前的逻辑读

压缩后：

```
set autotrace off
ALTER TABLE t MOVE COMPRESS;
execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t');
SELECT table_name, blocks,compression  FROM user_tables WHERE table_name = 'T';
TABLE_NAME                          BLOCKS COMPRESS
------------------------------ ---------- --------
T                                    2639 ENABLED

set autotrace traceonly
select count(*) from t;

执行计划
----------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   718   (1)| 00:00:09 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    |   200K|   718   (1)| 00:00:09 |
----------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       2595  consistent gets
          0  physical reads
          0  redo size
        423  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
 rows processed
```

脚本 7-31　表压缩后的逻辑读

压缩前 consistent gets 为 14297，压缩后为 2595。性能确实提升了不少。不过要注意，压缩后的表更新的开销会更大，查询耗费的 CPU 也更多。所以压缩表一般适合在更新比较少且 CPU 消耗不大，同时 IO 消耗很大的系统中使用。

（2）索引压缩

压缩索引时，联合索引的压缩度会高一些。

环境准备：

```
DROP TABLE t1 purge;

CREATE TABLE t1 AS select * from dba_objects;
alter table T1 modify owner not null;
alter table T1 modify object_name not null;
alter table T1 modify object_type not null;
insert  into t1 select * from t1;
insert  into t1 select * from t1;
commit;
create index idx1_object_union  on t1(owner,object_type,object_name);
execute dbms_stats.gather_index_stats(ownname=>user, indname=>'idx1_object_union');
```

未压缩索引的当前情况：

```
SELECT t.index_name,t.compression,t.leaf_blocks,t.blevel   FROM user_indexes t WHERE
index_name = 'IDX1_OBJECT_UNION';

INDEX_NAME                     COMPRESS LEAF_BLOCKS     BLEVEL
------------------------------ -------- ----------- ----------
IDX1_OBJECT_UNION              DISABLED        2043          2
```

开始压缩索引：

```
drop table t2 purge;
create table t2 as select * from t1;
alter table T2 modify owner not null;
alter table T2 modify object_name not null;
alter table T2 modify object_type not null;
create index idx2_object_union  on t2(owner,object_type,object_name);
ALTER index  idx2_object_union  rebuild  COMPRESS;
execute dbms_stats.gather_index_stats(ownname=>user, indname=>'idx2_object_union');

SELECT t.index_name,t.compression,t.leaf_blocks,t.blevel   FROM user_indexes t WHERE
index_name = 'IDX2_OBJECT_UNION';

INDEX_NAME                     COMPRESS LEAF_BLOCKS     BLEVEL
------------------------------ -------- ----------- ----------
IDX2_OBJECT_UNION              ENABLED          907          2
```

接下来比较一下两者的性能差异，首先是未压缩索引：

```
set linesize 1000
set autotrace traceonly
select count(*) from t1 ;

执行计划
----------------------------------------------------------------------------------
```

```
| Id  | Operation             | Name               | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |   572   (1)| 00:00:07 |
|   1 |  SORT AGGREGATE       |                    |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX1_OBJECT_UNION  |   251K|   572   (1)| 00:00:07 |
-------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       2067  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 7-32　索引压缩前的逻辑读

接下来是压缩索引：

```
select count(*) from t2 ;

执行计划
-------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |   258   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE       |                    |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX2_OBJECT_UNION  |   282K|   258   (1)| 00:00:04 |
-------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        922  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 7-33　索引压缩后的逻辑读

索引压缩前性能为 2067，压缩后为 922，性能差异不小。

3. 列类型优化

（1）什么列放什么值

什么类型就放什么值，否则会发生类型转换，导致系能问题！（存放字符的字段就设置为 varchar2 类型，存放数值的字段就设置为 number 类型，存放日期的字段就设置为 date 类型）。

如下：

```
drop table t_col_type purge;
create table t_col_type(id varchar2(20),col2 varchar2(20),col3 varchar2(20));
insert into t_col_type select rownum,'abc','efg' from dual connect by level<=10000;
commit;
create index idx_id on t_col_type(id);
set linesize 1000
set autotrace traceonly

select * from t_col_type where id=6;

执行计划
----------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     1 |    36 |     9   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T_COL_TYPE |     1 |    36 |     9   (0)| 00:00:01 |
----------------------------------------------------------------------------------
   1 - filter(TO_NUMBER("ID")=6)
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         32  consistent gets
          0  physical reads
          0  redo size
        540  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 7-34　访问索引列无法用到索引

实际上只有如下写法才可以用到索引，这个很不应该，应该是什么类型的取值就设置成什么样的字段。

```
select * from t_col_type where id='6';
执行计划
----------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
```

```
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     1 |    36 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_COL_TYPE |     1 |    36 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_ID      |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

   2 - access("ID"='6')
统计信息
----------------------------------------------------------

          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        544  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

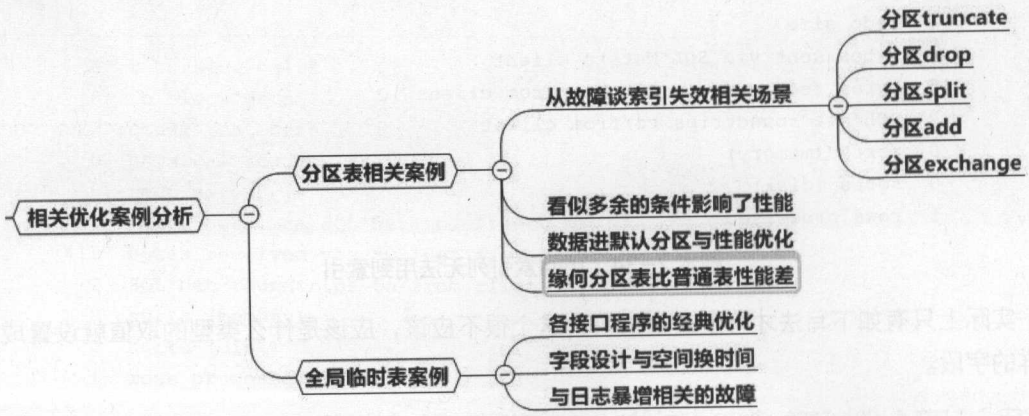
脚本 7-35　调整写法索引列才用到索引

（2）过时的字段

CHAR 和 LONG 类型基本要被 VARCHAR2 和 CLOB 等类型替换，它们属于已经过时的字段类型。其中 CHAR 不利扩展，且有可能浪费空间，而 LONG 的更新查询操作极为麻烦！

7.3　相关优化案例分析

本节案例主要和分区表及全局临时表有关，具体如下图所示：



7.3.1　分区表相关案例

1. 分区索引失效

某天，某系统某重要模块忽然变慢，导致系统处理工单产生大量积压，严重影响了生产。经过查询，发现原来是相关人员的误操作导致数据量巨大的分区表的全局索引失效了，结果超级大表在做全表扫描，噩梦从此开始。

到底是相关人员在做什么操作呢？别猜了！来，一起看看都有什么操作会让分区表的索引失效吧：

分区表索引失效的操作

操作动作	操作命令	全局索引		分区索引	
		是否失效	如何避免失效	是否失效	如何避免失效
truncate 分区	alter table part_tab_trunc truncate partition p1 ;	失效	alter table part_tab_trunc truncate partition p1 **Update GLOBAL indexes;**	没影响	
drop 分区	alter table part_tab_drop drop partition p1 ;	失效	alter table part_tab_drop drop partition p1 **Update GLOBAL indexes;**	没影响	
split 分区	alter table part_tab_split SPLIT PARTITION P_MAX at (30000) into (PARTITION p3, PARTITION P_MAX);	失效	alter table part_tab_split SPLIT PARTITION P_MAX at (30000) into (PARTITION p3, PARTITION P_MAX) **update global indexes;**	如果MAX区中已经有记录了，这个时候SPLIT就会导致有记录的新增分区的局部索引失效！	对局部索引进行重建索引alter index idx_part_split_col3 **rebuild;**
add 分区	alter table part_tab_add add PARTITION p6 values less than (60000);	没影响		没影响	
exchange 分区	alter table part_tab_exch exchange partition p1 with table normal_tab including indexes;	失效	alter table part_tab_exch exchange partition p1 with table normal_tab including indexes **update global indexes;**	没影响	

2. 看似多余的条件影响分区性能

这里不写代码，由读者自己去构造和体会，笔者只讲述一个场景：某 SQL 执行比较缓慢，后来查询该语句由于业务的特定需求，可以加上一个类似地区号的分区字段，这样就可以少访问别的分区，快速定位，提升性能。不过开发人员却没有加，导致性能出问题。好比查找建筑物为：福州市政府，那必然可以带上一个条件：福州。因为从客观事实上分析，福州市政府不可能建在厦门等别的地市。

3. 数据进默认分区与性能优化

某项目组发现最近系统运行越来越慢，事后发现，原来是一张分区表的访问较慢。经分析，发现原来是 maxvalue 分区在 2015 年 7 月前，导致后续几个月的数据全部都落到 p_max 分区中，造成分区不均匀，没有起到分区消除的作用，具体情况还原如下：

```
create table range_part_tab (id number,deal_date date,area_code number,nbr number,
contents varchar2(4000))
          partition by range (deal_date)
          (
          partition p_201501 values less than (TO_DATE('2015-02-01', 'YYYY-MM-DD')),
          partition p_201502 values less than (TO_DATE('2015-03-01', 'YYYY-MM-DD')),
          partition p_201503 values less than (TO_DATE('2015-04-01', 'YYYY-MM-DD')),
          partition p_201504 values less than (TO_DATE('2015-05-01', 'YYYY-MM-DD')),
          partition p_201505 values less than (TO_DATE('2015-06-01', 'YYYY-MM-DD')),
          partition p_201506 values less than (TO_DATE('2015-07-01', 'YYYY-MM-DD')),
          partition p_201507 values less than (TO_DATE('2015-08-01', 'YYYY-MM-DD')),
          partition p_max values less than (maxvalue)
          );
```

插入数据（注意，要根据实际情况调整 to_char(sysdate-365,'J')的具体值）。

```
insert into range_part_tab (id,deal_date,area_code,nbr,contents)
      select rownum,
             to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)),'J'),
             ceil(dbms_random.value(591,599)),
             ceil(dbms_random.value(18900000001,18999999999)),
             rpad('*',400,'*')
        from dual
      connect by rownum <= 100000;
commit;
```

经常有类似如下的案例：由于规划失误，数据都进默认分区，导致默认分区超大，失去分区表的意义。

```
SQL> select count(*) from range_part_tab partition (p_201501);
  COUNT(*)
----------
     16858
SQL> select count(*) from range_part_tab partition (p_201502);
  COUNT(*)
----------
      7664
SQL> select count(*) from range_part_tab partition (p_201503);
  COUNT(*)
----------
      8484
SQL> select count(*) from range_part_tab partition (p_201504);
  COUNT(*)
----------
      8177
```

```
SQL> select count(*) from range_part_tab partition (p_201505);
  COUNT(*)
----------
      8414
SQL> select count(*) from range_part_tab partition (p_201506);
  COUNT(*)
----------
      8245
SQL> select count(*) from range_part_tab partition (p_201507);
  COUNT(*)
----------
      8565
SQL> select count(*) from range_part_tab partition (p_max);
  COUNT(*)
----------
    133593
```

大部分数据进了默认分区，该分区基本上失去了分区的意义，而使用者可能还蒙在鼓里。后续进行 split 分区并重新规划后解决问题，速度提升了近 10 倍！另外我们还引进了监控手段，通过观察各分区数据量的比例来避免此类事情的发生。

4. 缘何分区表性能比普通表性能差

构造环境，分别建分区表和普通表及它们分别对应的索引：

```
--构造分区表，插入数据。
drop table part_tab purge;
create table part_tab (id int,col2 int,col3 int)
       partition by range (id)
       (
       partition p1 values less than (10000),
       partition p2 values less than (20000),
       partition p3 values less than (30000),
       partition p4 values less than (40000),
       partition p5 values less than (50000),
       partition p6 values less than (60000),
       partition p7 values less than (70000),
       partition p8 values less than (80000),
       partition p9 values less than (90000),
       partition p10 values less than (100000),
       partition p11 values less than (maxvalue)
       )
       ;
insert into part_tab select rownum,rownum+1,rownum+2 from dual connect by rownum
<=110000;
commit;
create  index idx_par_tab_col2 on part_tab(col2) local;
create  index idx_par_tab_col3 on part_tab(col3) ;
```

```
--构造普通表，表结构和数据量都与分区表一样。
drop table norm_tab purge;
create table norm_tab  (id int,col2 int,col3 int);
insert into norm_tab select rownum,rownum+1,rownum+2 from dual connect by rownum
<=110000;
commit;
create  index idx_nor_tab_col2 on norm_tab(col2) ;
create  index idx_nor_tab_col3 on norm_tab(col3) ;
```

第 1 组试验，首先查看分区表的性能：

```
select * from part_tab where col2=8 ;
执行计划
----------------------------------------------------------------------------------------------------
|Id |Operation                          |Name           |Rows|Bytes|Cost (%CPU)|Time    |Pstart|Pstop|
----------------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT                   |               |   1|   39|   13   (0)|00:00:01|      |     |
|  1| PARTITION RANGE ALL               |               |   1|   39|   13   (0)|00:00:01|    1 |   11|
|  2|  TABLE ACCESS BY LOCAL INDEX ROWID|PART_TAB       |   1|   39|   13   (0)|00:00:01|    1 |   11|
|* 3|   INDEX RANGE SCAN                |IDX_PAR_TAB_COL2|   1|     |   12   (0)|00:00:01|    1 |   11|
----------------------------------------------------------------------------------------------------
  3 - access("COL2"=8)
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         24  consistent gets
          0  physical reads
          0  redo size
        539  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
```

再查看普通表的性能：

```
select * from norm_tab where col2=8 ;
执行计划
----------------------------------------------------------------------------------------------
| Id |Operation                    | Name           |Rows  |Bytes |Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |                |    1 |   39 |    2   (0)| 00:00:01 |
|   1| TABLE ACCESS BY INDEX ROWID| NORM_TAB       |    1 |   39 |    2   (0)| 00:00:01 |
|*  2|  INDEX RANGE SCAN           | IDX_NOR_TAB_COL2|    1 |      |    1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------
 2 - access("COL2"=8)
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
```

```
         0  redo size
       543  bytes sent via SQL*Net to client
       415  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

为什么走普通表的索引居然比走分区表的 local 索引性能要好？岂不是不需要学什么知识了，不懂分区表的人 SQL 还跑得更快？

```
SQL> set timing on
SQL> set autotrace off
SQL> select index_name,
  2           blevel,
  3           leaf_blocks,
  4           num_rows,
  5           distinct_keys,
  6           clustering_factor
  7      from user_ind_statistics
  8     where table_name in( 'NORM_TAB');

INDEX_NAME         BLEVEL LEAF_BLOCKS   NUM_ROWS DISTINCT_KEYS CLUSTERING_FACTOR
-------------------- ---------- ---------- ---------- ----------------
IDX_NOR_TAB_COL2        1         244     110000        110000              299
IDX_NOR_TAB_COL3        1         244     110000        110000              299
SQL>     select index_name,
  2           blevel,
  3           leaf_blocks,
  4           num_rows,
  5           distinct_keys,
  6               clustering_factor FROM USER_IND_PARTITIONS where index_name like
'IDX_PAR_TAB%';

INDEX_NAME           BLEVEL LEAF_BLOCKS   NUM_ROWS DISTINCT_KEYS CLUSTERING_FACTOR
-------------------- ---------- ---------- ---------- ----------------
IDX_PAR_TAB_COL2          1          21       9999          9999                24
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10000         10000                28
IDX_PAR_TAB_COL2          1          23      10001         10001                28
已选择 11 行。
```

脚本 7-36　使用分区表性能比普通表还更差的场景

本质原因在于，分区表的分区索引虽然大小要比全局索引小很多，但是索引的范围检索性能是由索引的高度（BLEVEL）来决定的，而不是由索引的大小来决定的。这里分区表的检索由于没有分区条件，p_start 和 p_stop 从 1 到 11 访问了 11 个分区，访问了 11 个小索引。由于分区表的索引高度居然和普通表的索引高度一样（BLEVL 都是 1，表示高度是 2 层），11×2 当然远大于 2，故性能差异很明显！

后续将局部索引调整为全局索引，逻辑读从 24 减少到 3，性能大幅提升。

7.3.2 全局临时表案例

1. 统计信息引发性能血案

某系统上线以来运行非常平稳，半年后系统忽然遭遇性能问题，接下来通过定位，发现是某核心模块的 SQL 开销很大。不过单独将该 SQL 提取出来执行时，却发现非常快！再仔细一分析，才发现其中有一个表是全局临时表。涉及全局临时表的执行计划分析必须要在应用程序执行的 Session 中跟踪才有意义。单独提取出来分析执行计划是无意义的，因为全局临时表的数据在程序运行的 Session 中才可见。

进一步分析发现，原来有人把全局临时表的统计信息给收集了。这下，出大问题了！因为在系统层面收集全局临时表的统计信息，收集到的数据量一定是 0，而实际情况是 Session 里有多少数据量，就是多少数据量。我们来还原一下这个案例场景。

环境准备：

```
DROP TABLE t1 CASCADE CONSTRAINTS PURGE;
DROP TABLE t2 CASCADE CONSTRAINTS PURGE;
CREATE TABLE t1 (
     id NUMBER NOT NULL,
     n NUMBER,
     contents VARCHAR2(4000)
   )
   ;
CREATE global temporary table t2 (
     id NUMBER NOT NULL,
     t1_id NUMBER NOT NULL,
     n NUMBER,
     contents VARCHAR2(4000)
   ) on commit preserve rows
   ;
execute dbms_random.seed(0);
INSERT INTO t1
     SELECT  rownum,  rownum, dbms_random.string('a', 50)
       FROM dual
     CONNECT BY level <= 10
      ORDER BY dbms_random.random;
INSERT INTO t2 SELECT rownum, rownum, rownum, dbms_random.string('b', 50) FROM dual
```

```
CONNECT BY level <= 100000
    ORDER BY dbms_random.random;
COMMIT;
select count(*) from t1;
  COUNT(*)
----------
     10
select count(*) from t2;
  COUNT(*)
----------
  100000
```

正常的执行计划如下：

```
set linesize 1000
alter session set statistics_level=all ;
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
--------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |     10 |00:00:00.05 |     791 |
|*  1 |  HASH JOIN         |      |      1 |     10 |     10 |00:00:00.05 |     791 |
|   2 |   TABLE ACCESS FULL| T1   |      1 |     10 |     10 |00:00:00.01 |       7 |
|   3 |   TABLE ACCESS FULL| T2   |      1 |  90564 |    100K|00:00:00.02 |     784 |
--------------------------------------------------------------------------------
```

这个时候执行计划是对的，因为 T2 在当前 session 中确实是一个大表，T2 表处于被驱动的位置，是正确的。那么，我们新开一个 Session，收集全局临时表 T2 表的统计信息，如下：

```
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'T2',estimate_percent
=> 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;
PL/SQL 过程已成功完成。

select table_name,
       partition_name,
       last_analyzed,
       partition_position,
       num_rows
  from user_tab_statistics t
 where table_name ='T2';
TABLE_NAME  PARTITION_NAME    LAST_ANALYZED  PARTITION_POSITION   NUM_ROWS
--------------------------- -------------- ------------------ ----------
T2                           27-3月 -16                              0
```

然后回到刚才的 Session，发现执行计划变了，T2 和 T1 表的驱动顺序变了，如下：

```
set linesize 1000
alter session set statistics_level=all ;
SELECT *
```

```
FROM t1, t2
WHERE t1.id = t2.t1_id;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |     10 |00:00:00.10 |    9970 |
|*  1 |  HASH JOIN         |      |      1 |      1 |     10 |00:00:00.10 |    9970 |
|   2 |   TABLE ACCESS FULL| T2   |      1 |      1 |    100K|00:00:00.02 |    9962 |
|   3 |   TABLE ACCESS FULL| T1   |      1 |     10 |     10 |00:00:00.01 |       8 |
-------------------------------------------------------------------------------------
```

脚本 7-37　全局临时表收集统计信息导致 Oracle 执行计划错误

显然这个执行计划是错误低效的，Buffers 高达 9970，远超过 791，性能差异明显。 所以对全局临时表收集统计信息是一个非常错误的行为。我们也应该对是否在对全局临时表收集统计信息这事进行监控，具体我们稍后还有描述。

2. 各接口程序的经典优化

某应用系统在设计中遇到了一些麻烦事，为各种业务配置了十多张不同的接口表，比如短信、彩信、彩铃……这些表的结构、字段等大致都一样，数据模型中的表显得非常重复。如果设计成同一张表，用字段来区分，对并发却有影响。此外由于是接口表，数据输送过来后，处理结束后还要把接口表的数据清理了。

实际上，这个时候是最适合应用全局临时表的。我们后续经过系列改造和提取，将这些接口表统一改成一张全局临时表。由于不同的程序在不同的 Session 中数据互不可见，也互不影响，因此并发和自动清理机制的好处在这个案例中得到尽情的发挥！

具体案例简单还原如下，供参考：

```
create global temporary table t_global on commit preserve rows as select  * from
dba_objects where 1=2;
select table_name,temporary,duration from user_tables  where table_name='T_GLOBAL';

--不同 session 的例子，只试验一个基于 sesssion 的临时表就可以了，不用试验另外一个了。
---连上 session 1(比如是业务 1 的进程 )
 connect ljb/ljb

 insert into t_global select * from dba_objects  where rownum<=10;
 --可以体会提交，基于 session 的提交并清理数据
 commit;
 select count(*) from t_global;

  COUNT(*)
----------
    10

继续登录 session 2(比如是业务 2 的进程 )
```

```
connect yxl/yxl

insert into t_global select * from dba_objects  where rownum<=20;
commit;
select count(*) from t_global;

  COUNT(*)
----------
    20
--业务 n 的进程.....
```

3. 字段设计与空间换时间

某次在一个生产系统中发现一条语句执行得很慢，经过查看发现该语句 10 表关联， 执行计划甚为复杂，类似如下：

```
   SELECT *
  FROM t1           a,
       t_global_tab b,
       t2           c,
       t3           d,
       t4           e,
       t5           f,
  ...
 WHERE a.id=b.id
    and ...
    and ...
```

优化思路：对用于中间运算的全局临时表增加字段，从而减少了在一些业务场景下的表连接次数，后来经过业务的推敲和确认，发现将这个 t_gloabl_tab 临时表增加几个字段，就无须到那么多表中获取其他相关信息了。改造后的 SQL，从原来的 10 表关联瞬间变成 4 表关联。后来性能提升了 5 倍左右!

这是一个空间换时间的概念，不过也需要权衡利弊。

4. 与日志暴增相关的故障

某次系统出现 REDO 暴增的情况，经过查询，发现有大量的 delete 语句在操作，而该语句实质是在做中间运算，将临时数据先存在中间表，处理完毕后删除该表。

后来把这个表改为全局临时表后，REDO 暴增的情况得到很大缓解，系统性能同时也得以提升。

环境准备：

```
---试验准备工作，建观察 redo 的视图
sqlplus "/ as sysdba"
grant all on v_$mystat to ljb;
grant all on v_$statname to ljb;
connect  ljb/ljb
```

```
drop table t purge;
create table t as select * from dba_objects ;
--以下创建视图，方便后续直接用 select * from v_redo_size 进行查询
create or replace view v_redo_size as
    select a.name,b.value
    from v$statname a,v$mystat b
    where a.statistic#=b.statistic#
    and a.name='redo size';
```

方案 1

```
drop table t_tmp purge;
create table t_tmp (id int,col2 int ,col3 int,contents varchar2(4000));

select * from v_redo_size;
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
redo size                                                           9988

begin
   insert into t_tmp select rownum ,rownum+1,rownum+2,rpad('*',400,'*') from dual
connect by rownum <=10000;
   --临时插入 t_tmp 表后，接下来删除该临时表记录，中间略去了大部分逻辑
   delete from t_tmp ;
   commit;
end;
/

select * from v_redo_size;
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
redo size                                                         11385896
总共产生日志量为：11385896-9988=11375908
```

方案 2

```
--退出 session,连到新的 session 上完成如下操作：
drop table t_global purge;
create global temporary table t_global (id int,col2 int ,col3 int,contents
varchar2(4000)) on commit delete rows;

select * from v_redo_size;
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
redo size                                                           42272

begin
   insert into t_global select rownum ,rownum+1,rownum+2,rpad('*',400,'*') from dual
connect by rownum <=10000;
   --临时插入 t_global 表后，如下删除临时表记录的 delete 动作可以不做，commit 后数据自动清理
   --delete from t_global ;
  commit;
```

```
end;
/

select * from v_redo_size;
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
redo size                                                             209152
总共产生日志量为：209152-42272=166880
```

脚本 7-38　全局临时表减少大量日志，优化了生产系统

7.3.3　监控异常的表设计

1. 监控分区表相关设计

（1）监控失效分区索引

```
prompt <p>查询当前用户下，失效-普通索引
select t.index_name,
       t.table_name,
       blevel,
       t.num_rows,
       t.leaf_blocks,
       t.distinct_keys
  from user_indexes t
where status = 'INVALID';

prompt <p>查询当前用户下，失效-分区索引
select t1.blevel,
       t1.leaf_blocks,
       t1.INDEX_NAME,
       t2.table_name,
       t1.PARTITION_NAME,
       t1.STATUS
  from user_ind_partitions t1, user_indexes t2
where t1.index_name = t2.index_name
   and t1.STATUS = 'UNUSABLE';
```

脚本 7-39　监控失效分区索引

（2）监控未建分区的大表

```
prompt <p>当前用户下，表大小超过 10 个 GB 未建分区的
select segment_name,
       segment_type,
       sum(bytes) / 1024 / 1024 / 1024 object_size
  from user_segments
WHERE segment_type = 'TABLE'
group by  segment_name, segment_type
```

```
having sum(bytes) / 1024 / 1024 / 1024 >= 10
order by object_size desc;
```

脚本 7-40　监控未建分区的大表

（3）监控分区数过多的表

```
prompt <p>当前用户下，分区最多的前 10 个对象
select *
  from (select  table_name, count(*) cnt
          from user_tab_partitions
         group by  table_name
         order by cnt desc)
where rownum <= 10;

prompt <p>当前用户下，分区个数超过 100 的表
select  table_name, count(*) cnt
  from user_tab_partitions
 having count(*) >= 100
group by table_name, table_name
order by cnt desc;

--或者如下更方便
select table_name, partitioning_type, subpartitioning_type
  from user_part_tables
 where partition_count > 100;
```

脚本 7-41　监控分区数过多的表

（4）监控分区表各分区大小严重不均匀情况

```
set linesize 266
col table_name format a20
select table_name,
       max(num_rows),
       trunc(avg(num_rows),0),
       sum(num_rows),
       case when sum(num_rows),0 then 0,else trunc(max(num_rows) / sum(num_rows),2)
end,
       count(*)
  from user_tab_partitions
 group by table_name
having max(num_rows) / sum(num_rows) > 2 / count(*);

--也可用来作为判断查询当前用户下有因为疏于分区管理导致大量数据进了默认分区的参考。

select table_name,
       partition_name,
       num_rows
 from user_tab_partitions
```

```
 where table_name = 'RANGE_PART_TAB'
 order by num_rows desc;
```

脚本 7-42　监控分区表各分区大小严重不均匀情况

（5）监控当前有多少带子分区的分区表

```
select table_name,
       partitioning_type,
       subpartitioning_type,
       partition_count
  from user_part_tables
 where subpartitioning_type <> 'NONE';


select count(*) from user_part_tables where  subpartitioning_type <> 'NONE';
```

脚本 7-43　监控当前有多少带子分区的分区表

2. 监控哪些全局临时表被收集统计信息

```
prompt <p>当前用户下，被收集统计信息的临时表
select owner,
       table_name,
       t.last_analyzed,
       t.num_rows,
       t.blocks
  from user_tables t
where t.temporary = 'Y'
   and last_analyzed is not null;
```

脚本 7-44　监控哪些全局临时表被收集统计信息

3. 监控哪些外键未建索引

```
--查看当前数据库哪些对象外键没建索引
select table_name,
       constraint_name,
       cname1 || nvl2(cname2, ',' || cname2, null) ||
       nvl2(cname3, ',' || cname3, null) ||
       nvl2(cname4, ',' || cname4, null) ||
       nvl2(cname5, ',' || cname5, null) ||
       nvl2(cname6, ',' || cname6, null) ||
       nvl2(cname7, ',' || cname7, null) ||
       nvl2(cname8, ',' || cname8, null) columns
  from (select b.table_name,
               b.constraint_name,
               max(decode(position, 1, column_name, null)) cname1,
               max(decode(position, 2, column_name, null)) cname2,
               max(decode(position, 3, column_name, null)) cname3,
```

```
               max(decode(position, 4, column_name, null)) cname4,
               max(decode(position, 5, column_name, null)) cname5,
               max(decode(position, 6, column_name, null)) cname6,
               max(decode(position, 7, column_name, null)) cname7,
               max(decode(position, 8, column_name, null)) cname8,
               count(*) col_cnt
          from (select substr(table_name, 1, 30) table_name,
                       substr(constraint_name, 1, 30) constraint_name,
                       substr(column_name, 1, 30) column_name,
                       position
                  from user_cons_columns) a,
               user_constraints b
         where a.constraint_name = b.constraint_name
           and b.constraint_type = 'R'
         group by b.table_name, b.constraint_name) cons
 where col_cnt > ALL
 (select count(*)
          from user_ind_columns i
         where i.table_name = cons.table_name
           and i.column_name in (cname1, cname2, cname3, cname4, cname5,
                cname6, cname7, cname8)
           and i.column_position <= cons.col_cnt
         group by i.index_name)
```

脚本 7-45　监控哪些外键未建索引

查询所有含外键的表：

```
select count(*),TABLE_NAME,c_constraint_name from (
select a.table_name,
       substr(a.constraint_name, 1, 30) c_constraint_name,
       substr(a.column_name, 1, 30) column_name,
       position,
       b.owner,
       b.constraint_name,
       b.constraint_type
  from user_cons_columns a, user_constraints b
 where a.constraint_name = b.constraint_name
   and b.constraint_type = 'R' )
   group by TABLE_NAME,c_constraint_name
```

脚本 7-46　查询所有含外键的表

4. 监控表中有没有过时类型的字段

```
select table_name,
       column_name,
       data_type
  from user_tab_columns
```

```
where data_type in ( 'LONG','CHAR');
```

脚本 7-47　监控表中有没有过时类型的字段

7.3.4　表设计优化相关案例总结

通过表设计来优化SQL系列手段		
原先怎样	调整了什么	变成怎样
索引失效导致分区大表用不到索引	明白了让分区索引失效的所有操作	索引生效，性能提升
全分区扫描	增加一个地区为福州的等价改写（因为福州市政府就在福州），让查询落在了福州这个分区里	从全分区扫描变成局部分区扫描，性能提升
数据全部落入默认分区中，查询缓慢	将默认分区做split分区，同时增加分区表的各分区比例监控	性能提升近10倍！
分区表建立局部索引，但是SQL用不上分区条件，导致扫描了许多小索引	将分区表的局部索引调整成全局索引	逻辑读从24下降到3
对全局临时表收集统计信息，导致执行计划错误，SQL性能低下	不收集全局临时表统计信息	Buffer从9970降低到791
接口程序中根据不同业务设计了多张接口表，导致并发受到影响，清理数据开销也大	改造后接口表统一成一张全局临时表	并发能力大幅增加，性能提升明显
一条SQL的表关联有10个之多，性能较低	通过将某关键表的字段进行扩充，将表连接中需要的字段都融入到这个表中，于是表关联从10个减少为4个	性能提升5倍左右

7.4　本章习题、总结与延伸

习题 1：说说分区表的主要好处是什么，为什么会有这些好处？

习题 2：说说全局临时表的主要好处是什么。

习题 3：有人说分区表肯定是建局部索引，这个对吗？

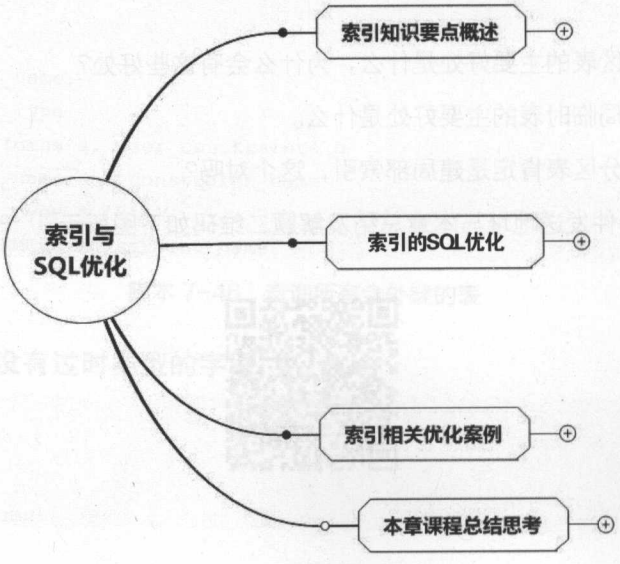
习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

第8章　且慢，学习索引如何让SQL飞

原来索引的学习如此重要

索引是 SQL 优化使用频度最高的优化武器，甚至可以说你对索引有了深刻的理解，你基本上可以优化身边 60%以上的 SQL。如何做到深刻地理解索引呢？最关键是知道索引的结构，并且明白这些结构有什么特点，再思考这些特点和哪些类型的 SQL 优化有关系。这样，索引的学习就算毕业了。

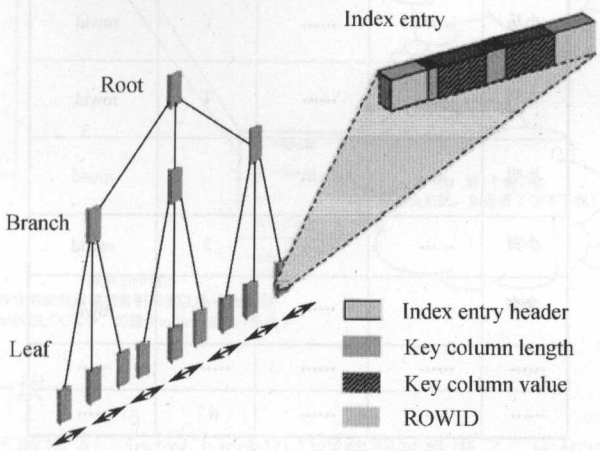
本章我们先从索引的知识要点入手，得出索引的结构后我们开始尝试让索引的原理和 SQL 结合起来。后面给大家分析一些经典案例，让大家从中得到更多的启发。最后对索引知识进行一系列延伸。总体学习思路如下图所示：



8.1 索引知识要点概述

8.1.1 索引结构的推理

接下来我们从通过系列推导来学习索引结构原理，让读者对索引有一个深刻的认识。然后再对结构进行分析研究，推导出和 SQL 优化有关的三大特性，将其应用到优化工作中去。



索引由 Root（根块）、Branch（茎块）和 Leaf（叶子块）三部分组成，其中 Leaf（叶子块）主要存储 key column value（索引列具体值），以及能具体定位到数据块所在位置的 rowid（注意区分索引块和数据块）。

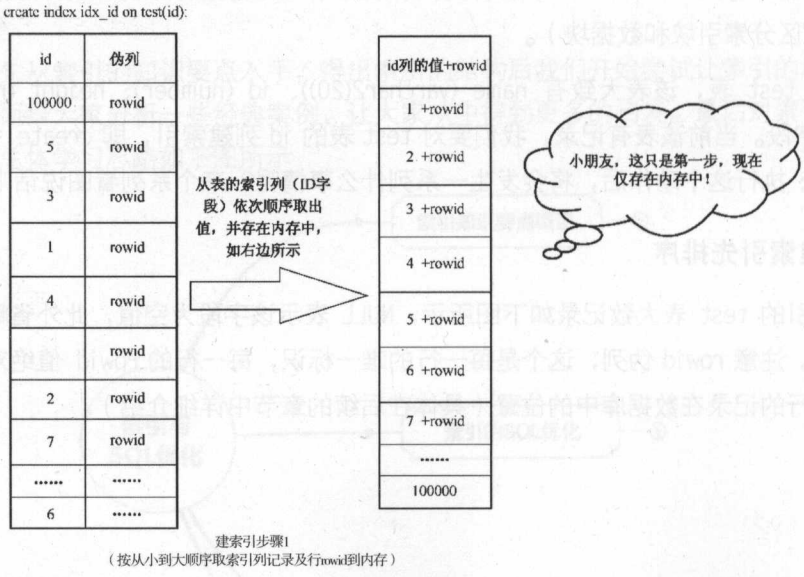
有一张 test 表，该表大致有 name (varchar2(20))、id (number)、height (number)、age (number)等字段。当前该表有记录，我们要对 test 表的 id 列建索引，即 create index idx_id on test(id)；执行这个动作后，将会发生一系列什么事情呢？来个系列看图说话吧。

1. 要建索引先排序

未建索引的 test 表大致记录如下图所示，NULL 表示该字段为空值，此外省略号表示略去不显示内容。注意 rowid 伪列，这个是每一行的唯一标识，每一行的 rowid 值绝对不重复，由它可定位到行的记录在数据库中的位置（具体在后续的章节中详细介绍）。

name	age	height	id	伪列
小余	……	……	100000	rowid
老张	……	……	5	rowid
老王	……	……	3	rowid
小马	……	……	1	rowid
大刘	……	……	4	rowid
小明	……	……		rowid
小黄	……	……	2	rowid
老李	……	……	7	rowid
……	……	……	……	……
……	……	……	6	……

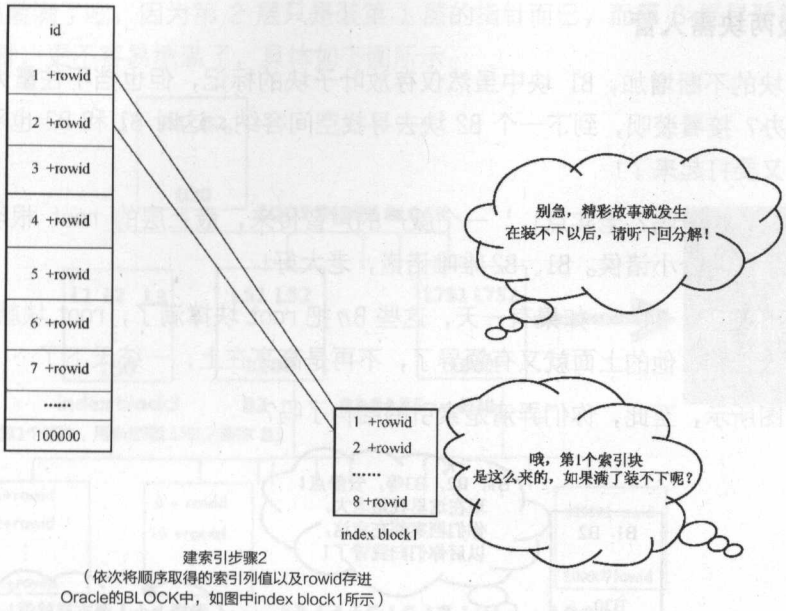
建索引后，先从 test 表的 id 列由小到大顺序取出数据放在内存中（这里需注意，除了 id 列的值外，还要注意取该列值的同时，该行的 rowid 也被一并取出），如下图所示：



建索引步骤1
（按从小到大顺序取索引列记录及行rowid到内存）

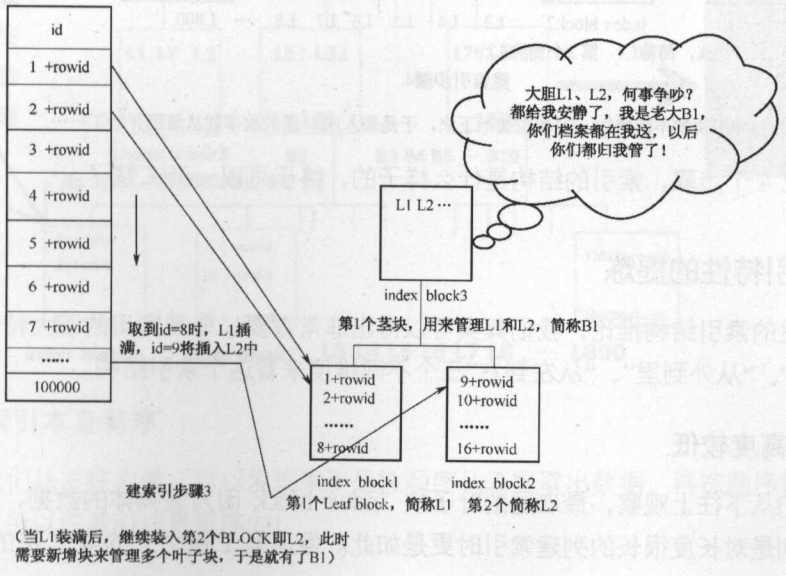
2. 列值入块成索引

依次将内存中的顺序存放的列值和对应的 rowid 存进 Oracle 空闲的 BLOCK 中，形成索引块，具体如下图所示：



3. 填满一块接一块

随着索引列的值不断插入，index block1(L1)很快就被插满了，比如接下来取出的 id=9 的记录将无法插入 index block1(L1)中，只有插入到新的 Oracle 块中，如下图所示的 index block2(L2)。与此同时，发生了一件非常重要的事情，就是新写数据到另一个块 index block3(B1)，这是为啥呢？原来 L1 和 L2 平起平坐，谁都不服谁，打起来了，不得了了，无组织无纪律哪能行，赶紧得有人管啊，于是 index block3(B1)就担负起管理的角色，这个 BLOCK 记录了 L1 和 L2 的信息，并不记录具体的索引列的键值，目前只占用了 B1 一点点空间。具体细节如下图所示。



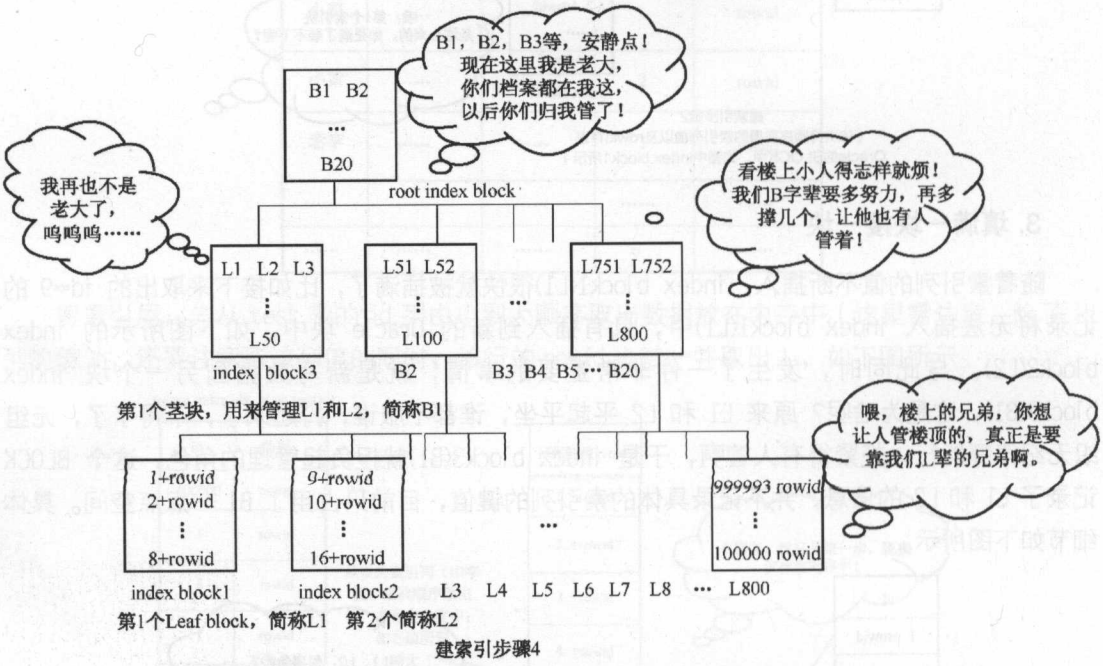
4. 同级两块需人管

随着叶子块的不断增加，B1 块中虽然仅存放叶子块的标记，但也挡不住量大，最终也容纳不下了。咋办？接着装呗，到下一个 B2 块去寻找空间容纳。这时 B1 和 B2 也平起平坐了，谁都不服谁，又要打起来了！

‘B1、B2，你们干吗？老实点！’一个威严的声音传来，最上层的 root 根块诞生了，并且有效地管住了这两个小诸侯。B1、B2 唯唯诺诺，老大好！

后续还会出现 B3、B4……如果有一天，这些 B*n* 把 root 块撑满了，root 块就不是 root 块了，他也要被人管着，他的上面就又有领导了，不再是高高在上，一统天下了……

具体如下图所示，至此，你们弄清楚索引的结构了吗？



建索引步骤4

随着叶子块的不断增加，B1块也装不下来，于是装入B2，新的故事就从这里开始了……

通过以上 4 个步骤，索引的结构是什么样子的，终于可以一清二楚了。

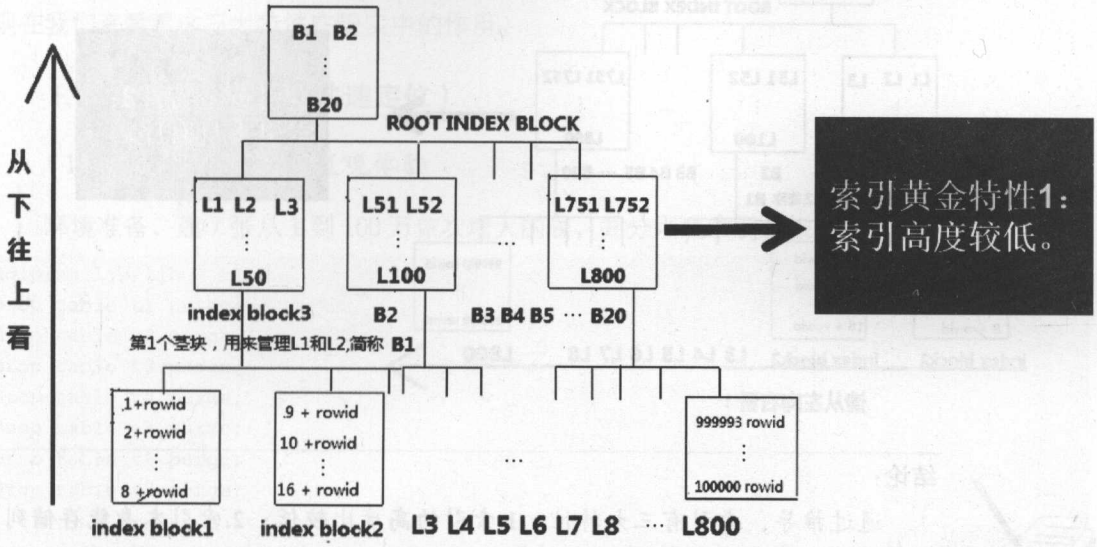
8.1.2 索引特性的提炼

通过上述的索引结构推论，我们其实可以得出非常重要、非常实用的三大特性，大家可通过“从下往上”、“从外到里”、“从左到右”三个不同维度来看这个索引结构。

1. 索引高度较低

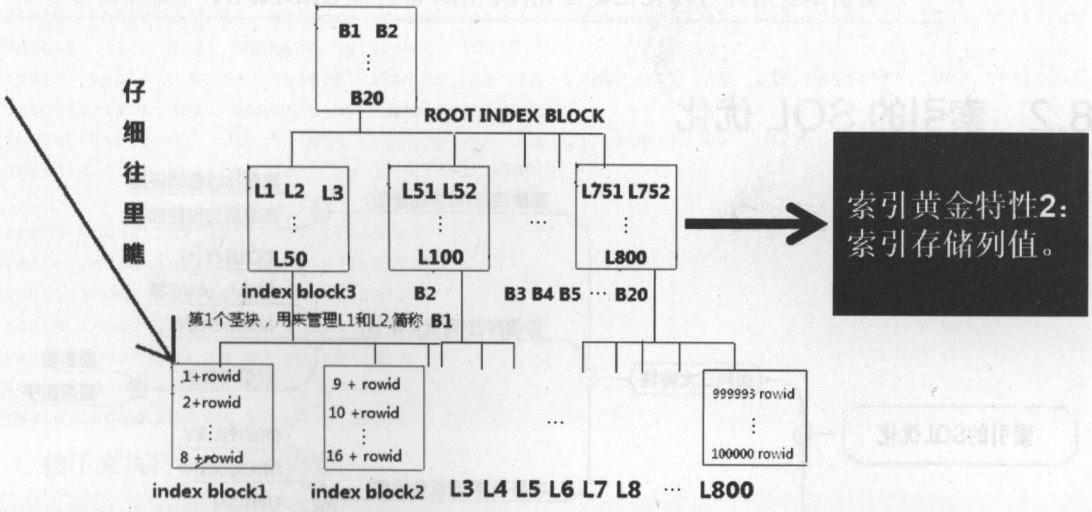
首先我们从下往上观察，最底层的叶子块 index block 因为装具体的数据，所以比较容易被填满，特别是对长度很长的列建索引时更是如此。但是第 1 层之上的第 2 层的 index block

就很不容易装满了吧，因为第 2 层只是装第 1 层的指针而已，而第 3 层是装第 2 层的 index block 的指针，更不容易填满了，具体如下图所示：



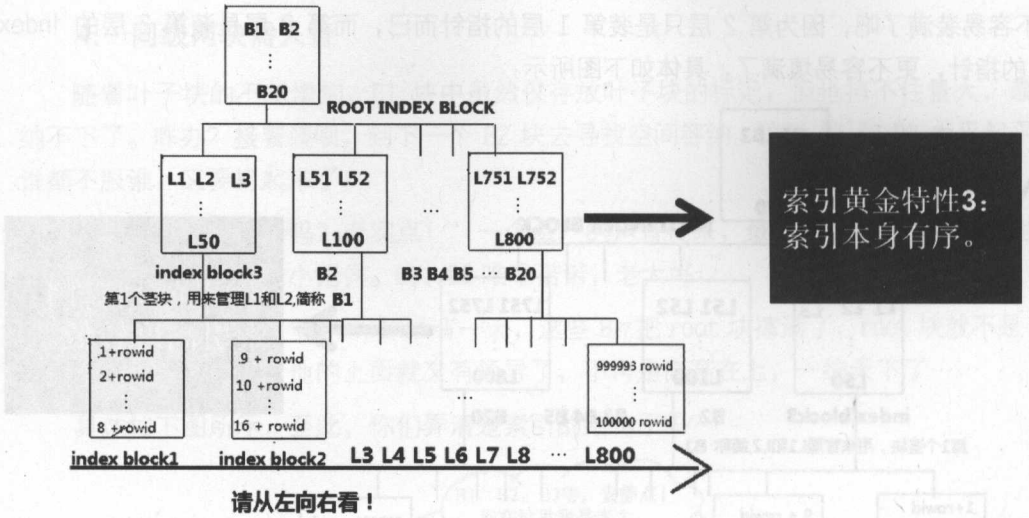
2. 索引存储列值

接下来我们从外往里瞧，我们很清楚，索引块就是存放索引的列值以及对应的 rowid。这里的 rowid 也是非常重要的，要想通过定位到的索引信息再返回到表中查到具体的其他列信息，还非得靠这个 rowid。如下图所示：



3. 索引本身有序

最后我们从左往右看，可以发现索引是按顺序从表里取出数据，再按顺序插入到块里形成索引块的，所以说索引块是有序的！



结论：

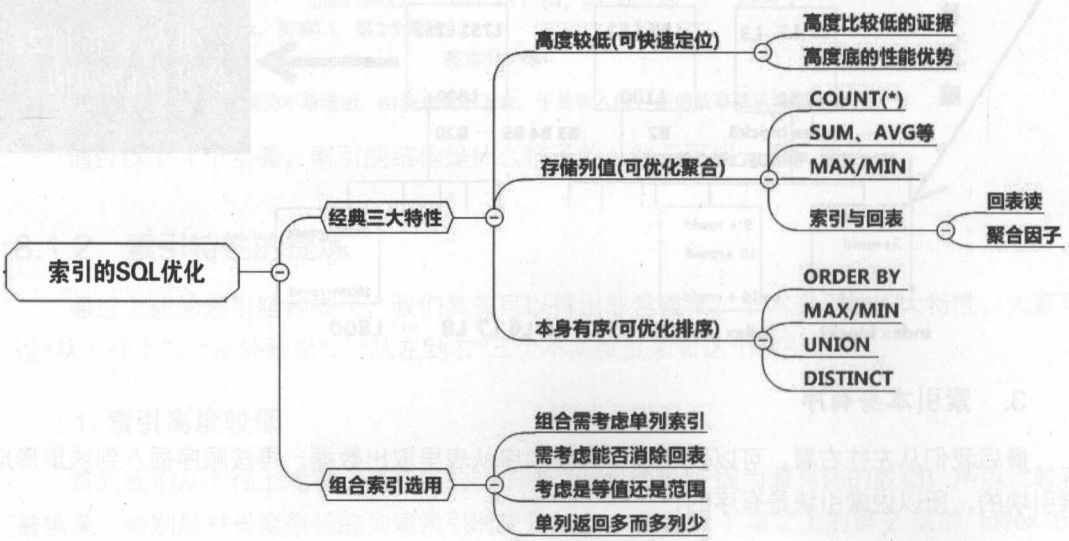
通过推导，索引有三大特性：**1.索引的高度比较低；2.索引本身能存储列值；3.索引本身有序。**

索引高度比较低这个特性告诉我们，为什么通过索引从海量数据中获取少量数据可以如此地快速！

索引本身能存储列值能够优化类似 COUNT(*) SUM(索引列)等聚合语句，因为 SQL 要的数据都在该列，无须从表中得到信息。

索引本身有序可以优化类似 MAX/MIN 语句及 ORDER BY 等排序语句。

8.2 索引的 SQL 优化



8.2.1 经典三大特性

我们通过一定的推理得出索引的真实结构，再通过索引的真实结构分析出索引三大特性，现在我们来看看这三大特性在现实中的作用。

1. 索引高度较低（可快速定位）

（1）索引高度较低的直观体验

环境准备，建7张从1到100万依次增大的表，并分别建索引，如下：

```
sqlplus ljb/ljb
drop table t1 purge;
drop table t2 purge;
drop table t3 purge;
drop table t4 purge;
drop table t5 purge;
drop table t6 purge;
drop table t7 purge;

create table t1 as select rownum as id ,rownum+1 as id2,rpad('*',1000,'*') as
contents from dual connect by level<=1;
create table t2 as select rownum as id ,rownum+1 as id2,rpad('*',1000,'*') as
contents from dual connect by level<=10;
create table t3 as select rownum as id ,rownum+1 as id2,rpad('*',1000,'*') as
contents from dual connect by level<=100;
create table t4 as select rownum as id ,rownum+1 as id2,rpad('*',1000,'*') as
contents from dual connect by level<=1000;
create table t5 as select rownum as id ,rownum+1 as id2,rpad('*',1000,'*') as
contents from dual connect by level<=10000;
create table t6 as select rownum as id ,rownum+1 as id2,rpad('*',1000,'*') as
contents from dual connect by level<=100000;
create table t7 as select rownum as id ,rownum+1 as id2,rpad('*',1000,'*') as
contents from dual connect by level<=1000000;

create index idx_id_t1 on t1(id);
create index idx_id_t2 on t2(id);
create index idx_id_t3 on t3(id);
create index idx_id_t4 on t4(id);
create index idx_id_t5 on t5(id);
create index idx_id_t6 on t6(id);
create index idx_id_t7 on t7(id);
```

接下来执行如下语句查询：

```
set linesize 1000
set autotrace off
select index_name,
         blevel,
         leaf_blocks,
         num_rows,
         distinct_keys,
```

```
          clustering_factor
     from user_ind_statistics
    where table_name in( 'T1','T2','T3','T4','T5','T6','T7');

INDEX_NAME   BLEVEL LEAF_BLOCKS   NUM_ROWS DISTINCT_KEYS CLUSTERING_FACTOR
------------------ ----------- ---------- ------------- -----------------
IDX_ID_T1         0           1          1             1                 1
IDX_ID_T2         0           1         10            10                 2
IDX_ID_T3         0           1        100           100                15
IDX_ID_T4         1           3       1000          1000               143
IDX_ID_T5         1          21      10000         10000              1429
IDX_ID_T6         1         222     100000        100000             14286
IDX_ID_T7         2        2226    1000000       1000000            142858

已选择 7 行。
```

脚本 8-1　索引高度较低的直观体验

这里的 BLEVEL=0 表示索引仅有叶子块，高度为 1。大家发现了什么没有？从 T1 到 T7 表的记录 NUM_ROWS 是以 10 倍的速度不断增加，而索引 IDX_ID_T1 到 IDX_ID_T7 的高度却以极其缓慢的速度增加，这说明了索引的高度确实是比较低。

（2）高度低有利于索引范围扫描

还是上面的环境，我们做如下 7 组试验，首先是语句 1 针对 t1 表的索引访问和全表扫描访问，如下：

```
select * from t1 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
          2  consistent gets
select /*+full(t1)*/ * from t1 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
          3  consistent gets
```

语句 2，针对 t2 表的索引访问和全表扫描访问，如下：

```
select * from t2 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
          3  consistent gets
select /*+full(t2)*/ * from t2 where id=1;
统计信息
```

```
--------
          0  recursive calls
          0  db block gets
          5  consistent gets
```

语句3，针对t3表的索引访问和全表扫描访问，如下：

```
select * from t3 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
          3  consistent gets
select /*+full(t3)*/ * from t3 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
         19  consistent gets
```

语句4，针对t4表的索引访问和全表扫描访问，如下：

```
select * from t4 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
select /*+full(t4)*/ * from t4 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
        148  consistent gets
```

语句5，针对t5表的索引访问和全表扫描访问，如下：

```
select * from t5 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
select /*+full(t5)*/ * from t5 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
       1435  consistent gets
```

语句6，针对t6表的索引访问和全表扫描访问，如下：

```
select * from t6 where id=1;
```

```
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
select /*+full(t6)*/ * from t6 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
      14298  consistent gets
```

语句 7，针对 t7 表的索引访问和全表扫描访问，如下：

```
select * from t7 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
          5  consistent gets
select /*+full(t7)*/ * from t7 where id=1;
统计信息
--------
          0  recursive calls
          0  db block gets
     142866  consistent gets
```

脚本 8-2　高度低有利于索引范围扫描

规律：从 t1 到 t7，表记录依次增大 10 倍，从 1 到 1000000，索引扫描的逻辑读是 2、3、3、4、4、4、5。从 t1 到 t7，表记录依次增大 10 倍，从 1 到 1000000，全表扫描的逻辑读是 3、5、19、148、1435、14298、142866，说明随着记录的增加，索引访问的优势越来越明显！

2. 索引存储列值（可优化聚合）

（1）索引特性之存列值优化 count

要领：只要索引能回答问题，索引就可以被当成一个“瘦表”，这样访问路径就会减少。另外切记不存储空值。

```
drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
commit;
create index idx1_object_id on t(object_id);
set autotrace on
select count(*) from t;
执行计划
--------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
```

```
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     1 |   292    (1)| 00:00:04 |
|   1 |  SORT AGGREGATE     |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 69485 |   292    (1)| 00:00:04 |
-----------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1048  consistent gets
```

脚本 8-3　count 无法用到索引

为啥用不到索引，因为索引不能存储空值，所以加上一个 is not null，再试验看看：

```
select count(*) from t where object_id is not null;
执行计划
--------
| Id  | Operation             | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                |     1 |    13 |    50   (2)| 00:00:01 |
|   1 |  SORT AGGREGATE       |                |     1 |    13 |            |          |
|*  2 |   INDEX FAST FULL SCAN| IDX1_OBJECT_ID | 69485 |   882K|    50   (2)| 00:00:01 |
-------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        170  consistent gets
```

脚本 8-4　修改代码让 count 用到索引

也可以不加 is not null，直接把列的属性设置为 not null，也可以，继续试验如下：

```
alter table t modify OBJECT_ID not null;
select count(*) from t;
执行计划
--------
| Id  | Operation             | Name           | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                |     1 |    49   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |                |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX1_OBJECT_ID | 69485 |    49   (0)| 00:00:01 |
-----------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        170  consistent gets
```

脚本 8-5　修改代码让 count 用到索引

当然，如果是主键就无须定义列是否允许为空了。

```
drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
alter table t add constraint pk1_object_id primary key (OBJECT_ID);
set autotrace on
select count(*) from t;

执行计划
--------
| Id  | Operation             | Name          | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               |     1 |    46   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |               |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| PK1_OBJECT_ID | 69485 |    46   (0)| 00:00:01 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        160  consistent gets
```

脚本 8-6　主键让 count 用到索引

（2）索引特性之存列值优化 sum avg

```
SUM/AVG 的优化
drop table t purge;
create table t as select * from dba_objects;
create index idx1_object_id on t(object_id);
set autotrace on
set linesize 1000
set timing on

select sum(object_id) from t;
执行计划
--------
| Id  | Operation             | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               |     1 |    13 |    49   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |               |     1 |    13 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX1_OBJECT_ID | 92407 |  1173K|    49   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        170  consistent gets
          0  physical reads
```

```
          0  redo size
        432  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-7　索引特性之存列值优化 sum avg

比较一下不走索引的代价，体会一下这个索引的重要性：

```
select /*+full(t)*/ sum(object_id) from t;
SUM(OBJECT_ID)
--------------
  2732093100
执行计划
--------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    13 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |    13 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 92407 |  1173K|   292   (1)| 00:00:04 |
-------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1047  consistent gets
          0  physical reads
          0  redo size
        432  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-8　sum avg 不走索引的代价

类似的比如 AVG，和 SUM 的表现是一样的，就不再做试验了。

3. 索引本身有序（可优化排序）

（1）索引特性之有序优化 order by

以下语句没有索引有 order by，必然产生排序。

```
set autotrace traceonly
set linesize 1000
drop table t purge;
```

```
create table t as select * from dba_objects;
select * from t where object_id>2 order by object_id;
执行计划
--------
| Id  | Operation          | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 92407 |    18M|       |  4454   (1)| 00:00:54 |
|   1 |  SORT ORDER BY     |      | 92407 |    18M|    21M|  4454   (1)| 00:00:54 |
|*  2 |   TABLE ACCESS FULL| T    | 92407 |    18M|       |   294   (2)| 00:00:04 |
-----------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1047  consistent gets
          0  physical reads
          0  redo size
    3513923  bytes sent via SQL*Net to client
      54029  bytes received via SQL*Net from client
       4876  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
      73117  rows processed
```

脚本 8-9　无索引的 order by 语句必然会排序

新增索引后，Oracle 就有可能利用索引本身就有序的特点，来避免排序，如下：

```
create index idx_t_object_id on t(object_id);
set autotrace traceonly

select * from t where object_id>2 order by object_id;
执行计划
--------
| Id  | Operation                   | Name           | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                | 92407|    18M|  1302   (1)| 00:00:16 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T              | 92407|    18M|  1302   (1)| 00:00:16 |
|*  2 |   INDEX RANGE SCAN          | IDX_T_OBJECT_ID | 92407|       |   177   (1)| 00:00:03 |
-------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
      10952  consistent gets
          0  physical reads
          0  redo size
    8115221  bytes sent via SQL*Net to client
      54029  bytes received via SQL*Net from client
       4876  SQL*Net roundtrips to/from client
```

```
          0  sorts (memory)
          0  sorts (disk)
      73117  rows processed
```

脚本 8-10 索引让 order by 语句排序消失

观察黑体部分的 sorts (memory)，果然排序消失了。

（2）索引特性之有序优化 MAX/MIN

```
--MAX/MIN 的索引优化
drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
alter table t add constraint pk_object_id primary key (OBJECT_ID);
set autotrace on
set linesize 1000

select max(object_id) from t;
执行计划
--------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |     1 |    13 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE              |              |     1 |    13 |            |          |
|   2 |   INDEX FULL SCAN (MIN/MAX)| PK_OBJECT_ID |     1 |    13 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
        431  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-11 MAX/MIN 的索引优化

假设没用到索引的情况如下，看看执行计划有何不同，再看看代价和逻辑读的差异！

```
select /*+full(t)*/ max(object_id) from t;
执行计划
--------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    13 |   292   (1)| 00:00:04 |
```

```
|   1 |  SORT AGGREGATE    |       |      1 |    13 |            |          |
|   2 |   TABLE ACCESS FULL| T     | 92407 |  1173K|   292   (1)| 00:00:04 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1047  consistent gets
          0  physical reads
          0  redo size
        431  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-12　MAX/MIN 语句用不到索引性能低下

另外，可以做如下试验观察在有索引的情况下，随着记录数增加，性能差异是否明显？

```
set autotrace off
drop table t_max purge;
create table t_max as select * from dba_objects;
insert into t_max select * from t_max;
insert into t_max select * from t_max;
insert into t_max select * from t_max;
insert into t_max select * from t_max;
insert into t_max select * from t_max;
select count(*) from t_max;
create index idx_t_max_obj on t_max(object_id);
set autotrace on
select max(object_id) from t_max;

执行计划
--------
| Id  | Operation                  | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |               |     1 |    13 |     3   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE            |               |     1 |    13 |            |          |
|   2 |   INDEX FULL SCAN (MIN/MAX)| IDX_T_MAX_OBJ |     1 |    13 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        431  bytes sent via SQL*Net to client
```

```
       415  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

脚本 8-13　MAX/MIN 用索引与数据量增加的影响

做这个试验的目的是为了说明，由于索引的结构原因，导致查询最小值在左边，最大值在右边，索引无论多大，性能影响都不大，实际情况果然如此。具体如下表所示：

索引三大特性应用整理		
索引特性	**带来的好处**	**应用的常见 SQL**
高度比较低	高度低有利于索引范围扫描	select * from t7 where id=1;
索引存储列值	在索引本身可以回答问题的情况下，索引比表一般要小得多	select count(*) from t; select sum(object_id) from t;
索引本身有序	能够利用原有的排序消除排序	select * from t where object_id>2 order by object_id;
	能够用索引有序的特性在叶子块的最左边或最右边找到最小和最大值	select max(object_id) from t;

8.2.2　组合索引选用

1. 适用单列查询返回多、组合查询返回少的场景

比如 where 学历=硕士以上，返回不少的记录；再如 where 职业=收银员，同样返回不少的记录。于是无论哪个条件查询做索引，都不合适。可是，如果学历为硕士以上，同时职业又是收银员的，返回的就少之又少了。于是联合索引就可以这么开始建了。

2. 组合查询的组合顺序，要考虑单独的前缀查询

比如你在建 id,object_type 联合索引时，要考虑是单独 where id=xxx 查询返回的多，还是单独 where object_type 查询返回的多。这里细节就暂时略去了，在案例的部分中还有描述。

3. 仅等值无范围查询时，组合的顺序不影响性能

环境准备：

```
drop table t purge;
create table t as select * from dba_objects;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
update t set object_id=rownum ;
```

```
commit;
create index idx_id_type on t(object_id,object_type);
create index idx_type_id on t(object_type,object_id);
set autotrace off
alter session set statistics_level=all ;
set linesize 366
```

看看写法 1，用 type_id，id 顺序组合索引 idx_id_type，如下：

```
select /*+index(t,idx_id_type)*/ * from  t  where object_id=20  and object_type='TABLE';
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------
| Id |Operation                    | Name       |Starts |E-Rows|A-Rows|   A-Time   |Buffers |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |            |     1 |      |     1|00:00:00.01|      5 |
|   1| TABLE ACCESS BY INDEX ROWID| T          |     1 |    57|     1|00:00:00.01|      5 |
|*  2|  INDEX RANGE SCAN           | IDX_ID_TYPE|     1 |     9|     1|00:00:00.01|      4 |
-------------------------------------------------------------------------------------------
```

脚本 8-14　type_id，id 顺序组合索引

看看写法 2，用 id,type_id 顺序组合索引 idx_type_id，如下：

```
select /*+index(t,idx_type_id)*/ * from  t  where object_id=20  and object_type='TABLE';
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
Plan hash value: 3420768628
---------------------------
| Id |Operation                    | Name       |Starts |E-Rows |A-Rows|   A-Time   |Buffers|
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |            |     1 |       |     1|00:00:00.01|      5|
|   1| TABLE ACCESS BY INDEX ROWID| T          |     1 |    57 |     1|00:00:00.01|      5|
|*  2|  INDEX RANGE SCAN           | IDX_TYPE_ID|     1 |     9 |     1|00:00:00.01|      4|
-------------------------------------------------------------------------------------------
```

脚本 8-15　id、type_id 顺序组合索引

发现两个语句的性能是一样的，Buffers 都为 5。

4. 组合索引最佳顺序一般是将等值查询的列置前

```
select /*+index(t,idx_id_type)*/ *  from    t where object_id>=20 and object_id<2000 and
object_type='TABLE';
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------
| Id |Operation                    |Name       |Starts |E-Rows|A-Rows |   A-Time   |Buffers |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |           |     1 |      |   469 |00:00:00.01 |     86 |
|   1| TABLE ACCESS BY INDEX ROWID|T          |     1 |    14|   469 |00:00:00.01 |     86 |
|*  2|  INDEX RANGE SCAN           |IDX_ID_TYPE|     1 |     1|   469 |00:00:00.01 |     40 |
-------------------------------------------------------------------------------------------
```

脚本 8-16　将等值查询的列置前

```
select /*+index(t,idx_type_id)*/ *  from  t  where object_id>=20 and object_id<2000
and object_type='TABLE';
-------------------------------------------------------------------------------------------
| Id |Operation                    | Name       |Starts|E-Rows |A-Rows |   A-Time  | Buffers |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |            |     1|       |   469 |00:00:00.01|     81 |
|   1| TABLE ACCESS BY INDEX ROWID| T          |     1|   469 |   469 |00:00:00.01|     81 |
|*  2|  INDEX RANGE SCAN           | IDX_TYPE_ID|     1|   469 |   469 |00:00:00.01|     35 |
-------------------------------------------------------------------------------------------
```

脚本 8-17 将等值查询的列置后

通过脚本 8-16 和脚本 8-17 可以看出，where object_id>=20 and object_id<2000 表示 object_id 列是用于范围查询的，所以建议将 object_id 这个列置后。而 and object_type='TABLE' 表明 object_type 列用于等值查询，因此建议将 object_type 列置前，所以脚本 8-17 的 Buffers 是 81，比脚本 8-16 更低，性能更好。

8.2.3 索引扫描类型的分类与构造

1. INDEX RANGE SCAN

```
--请记住这个 INDEX RANGE SCAN 扫描方式

drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
commit;
create  index idx_object_id on t(object_id);
set autotrace traceonly
set linesize 1000
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'T',estimate_percent
=> 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;

select * from t where object_id=8;
执行计划
--------
| Id  | Operation                    | Name           | Rows  |Cost (%CPU)|
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |    2   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| T              |     1 |    2   (0)|
|*  2 |   INDEX RANGE SCAN           | IDX_OBJECT_ID |     1 |    1   (0)|
-----------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
```

```
          0  redo size
       1394  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-18　INDEX RANGE SCAN

2. INDEX UNIQUE SCAN

```
--请注意这个 INDEX UNIQUE SCAN 扫描方式,在唯一索引情况下使用。

drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
commit;
create unique index idx_object_id on t(object_id);
set autotrace traceonly
set linesize 1000

select * from t where object_id=8;
执行计划
--------
| Id  | Operation                    | Name          | Rows  |Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    2   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| T             |     1 |    2   (0)|
|*  2 |   INDEX UNIQUE SCAN          | IDX_OBJECT_ID |     1 |    1   (0)|
-------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
       1298  bytes sent via SQL*Net to client
        404  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-19　INDEX UNIQUE SCAN

3. TABLE ACCESS BY USER ROWID

```
--请注意这个 TABLE ACCESS BY USER ROWID 扫描方式,直接根据 rowid 来访问，最快的访问方式!
```

```
drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
commit;
--注意，这里连索引都没建!
--create  index idx_object_id on t(object_id);
set autotrace off
select rowid from t where object_id=8;
ROWID
-----
AAAZxiAAGAAAB07AAH

set autotrace traceonly
set linesize 1000

select * from t where object_id=8 and rowid='AAAZxiAAGAAAB07AAH';
执行计划
--------
|Id |Operation                   | Name|Rows | Bytes|Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|  0|SELECT STATEMENT            |     |    1|   219|    1   (0)| 00:00:01|
|* 1| TABLE ACCESS BY USER ROWID| T   |    1|   219|    1   (0)| 00:00:01|
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          1  consistent gets
          0  physical reads
          0  redo size
       1391  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本8-20 TABLE ACCESS BY USER ROWID

4. INDEX FULL SCAN

```
--请记住这个 INDEX FULL SCAN 扫描方式，并体会与 INDEX FAST FULL SCAN 的区别

drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
commit;
alter table T modify object_id not null;
create  index idx_object_id on t(object_id);
set autotrace traceonly
```

```
set linesize 1000

select * from t  order by object_id;
执行计划
--------
|Id  |Operation                    | Name          | Rows  |Bytes |Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT             |               | 88780 |   17M| 1208   (1)| 00:00:15|
|  1 | TABLE ACCESS BY INDEX ROWID| T             | 88780 |   17M| 1208   (1)| 00:00:15|
|  2 |  INDEX FULL SCAN            | IDX_OBJECT_ID| 88780 |      |  164   (1)| 00:00:02|
-------------------------------------------------------------------------------------------
统计信息
---------
          0  recursive calls
          0  db block gets
      10873  consistent gets
          0  physical reads
          0  redo size
    8116181  bytes sent via SQL*Net to client
      54040  bytes received via SQL*Net from client
       4877  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      73130  rows processed
```

脚本 8-21　INDEX FULL SCAN

5. INDEX FAST FULL SCAN

```
---请记住这个 INDEX FAST FULL SCAN 扫描方式，并体会与 INDEX FULL SCAN 的区别

drop table t purge;
create table t as select * from dba_objects ;
update t set object_id=rownum;
commit;
alter table T modify object_id not null;
create  index idx_object_id on t(object_id);
set autotrace traceonly
set linesize 1000

select count(*) from t;
执行计划
--------
|Id |Operation             |Name          |Rows  |Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|  0|SELECT STATEMENT      |              |    1 |   49   (0)| 00:00:01|
|  1| SORT AGGREGATE       |              |    1 |           |         |
|  2|  INDEX FAST FULL SCAN|IDX_OBJECT_ID|88780 |   49   (0)| 00:00:01|
-------------------------------------------------------------------------
统计信息
```

```
--------
          0  recursive calls
          0  db block gets
        170  consistent gets
          0  physical reads
          0  redo size
        425  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-22 INDEX FAST FULL SCAN

6. INDEX FULL SCAN (MINMAX)

```
--请注意这个 INDEX FULL SCAN (MIN/MAX)扫描方式

drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
commit;
create  index idx_object_id on t(object_id);
set autotrace traceonly
set linesize 1000

select max(object_id) from t;
执行计划
--------
|Id  |Operation                  |Name          |Rows  |Bytes |Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT           |              |    1 |   13 |    2   (0)| 00:00:01 |
|  1 | SORT AGGREGATE            |              |    1 |   13 |           |          |
|  2 |  INDEX FULL SCAN (MIN/MAX)|IDX_OBJECT_ID|    1 |   13 |    2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
        431  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-23 INDEX FULL SCAN (MINMAX)

7. INDEX SKIP SCAN

```
--请记住这个 INDEX SKIP SCAN 扫描方式

drop table t purge;
create table t as select * from dba_objects;
update t set object_type='TABLE' ;
commit;
update t set object_type='VIEW' where rownum<=30000;
commit;
create  index idx_type_id on t(object_type,object_id);
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'T',estimate_percent
=> 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;
set autotrace traceonly
set linesize 1000
select * from t where object_id=8;

执行计划
--------
| Id |Operation                    |Name        |Rows  |Bytes |Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |            |    1 |   94 |    4   (0)| 00:00:01 |
|   1| TABLE ACCESS BY INDEX ROWID|T           |    1 |   94 |    4   (0)| 00:00:01 |
|*  2|  INDEX SKIP SCAN            |IDX_TYPE_ID|    1 |      |    3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OBJECT_ID"=8)
       filter("OBJECT_ID"=8)

统计信息
--------
          1  recursive calls
          0  db block gets
          7  consistent gets
          0  physical reads
          0  redo size
       1401  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-24　INDEX SKIP SCAN

8. TABLE ACCESS BY INDEX ROWID

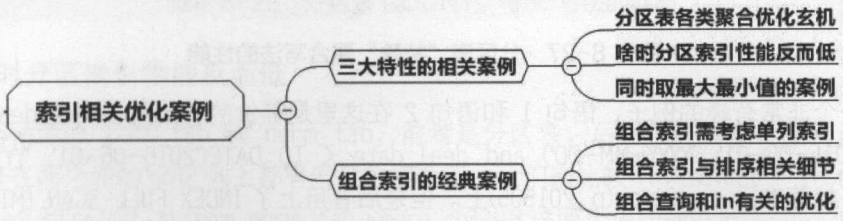
```
--好好地体会前后两个试验，记住这个 TABLE ACCESS BY INDEX ROWID
drop table t purge;
```

```
create table t as select * from dba_objects;
update t set object_id=rownum;
commit;
create  index idx_object_id on t(object_id);
set autotrace traceonly explain
set linesize 1000

select object_id from t where object_id=2 and object_type='TABLE';
------------------------------------------------------------------
|Id |Operation                    |Name          |Rows  |Bytes |Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|  0|SELECT STATEMENT             |              |    9 |  216 |    2   (0)| 00:00:01 |
|* 1| TABLE ACCESS BY INDEX ROWID|T             |    9 |  216 |    2   (0)| 00:00:01 |
|* 2|  INDEX RANGE SCAN           |IDX_OBJECT_ID|   12 |      |    1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

--在接下来的试验中，你会看到，哇塞，TABLE ACCESS BY INDEX ROWID 消失了。
create  index idx_id_type on t(object_id,object_type);
select object_id from t where object_id=2 and object_type='TABLE';

执行计划
--------
| Id  | Operation         | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |              |     9 |   216 |     1   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| IDX_ID_TYPE |     9 |   216 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

脚本 8-25 TABLE ACCESS BY INDEX ROWID

8.3 索引相关优化案例

通过前面一系列基本原理的讲解和索引优化方法的介绍，现在我们再举出一些经典案例，希望给读者更多启发。如下图所示：



8.3.1 三大特性的相关案例

1. 分区表各类聚合优化玄机

环境准备略去，请看语句1：

```
select max(nbr) max_nbr
  from range_part_tab
 where deal_date >= TO_DATE('2015-05-01', 'YYYY-MM-DD')
   and deal_date < TO_DATE('2015-06-01', 'YYYY-MM-DD');
执行计划
--------
| Id  | Operation                | Name           | Rows  |Cost (%CPU)| Pstart| Pstop
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                |     1 |  170   (0)|       |       |
|   1 |  SORT AGGREGATE          |                |     1 |           |       |       |
|   2 |   PARTITION RANGE SINGLE|                |    22 |  170   (0)|     5 |     5 |
|   3 |    TABLE ACCESS FULL     | RANGE_PART_TAB |    22 |  170   (0)|     5 |     5 |
-----------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        568  consistent gets
```

脚本 8-26　分区表普通聚合写法的性能

接下来看语句 2：

```
select max(nbr) max_nbr from range_part_tab partition(p_201505);
执行计划
--------
| Id  | Operation                    | Name          | Rows  | Cost (%CPU)|Pstart| Pstop |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |     2   (0)|      |       |
|   1 |  SORT AGGREGATE              |               |     1 |            |      |       |
|   2 |   PARTITION RANGE SINGLE     |               |     1 |     2   (0)|    5 |     5 |
|   3 |    INDEX FULL SCAN (MIN/MAX)| IDX_PART_NBR  |     1 |     2   (0)|    5 |     5 |
-----------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          2  consistent gets
```

脚本 8-27　分区表“特殊”聚合写法的性能

这是一个非常有趣的例子，语句 1 和语句 2 在这里是等价的，因为 where deal_date >= TO_DATE('2015-05-01', 'YYYY-MM-DD') and deal_date < TO_DATE('2015-06-01', 'YYYY-MM-DD') 的条件正好就落在了 partition(p_201505)上。但是后者用上了 INDEX FULL SCAN (MIN/MAX)，而前者走了全表扫描。当然，不止是这个 SUM 的写法，其他分区表的聚合写法也有类似效果，比如接下来我们再看看 COUNT(*)的语句 3 和语句 4。

语句 3：

```
select count(*) max_nbr
  from range_part_tab
 where deal_date >= TO_DATE('2015-05-01', 'YYYY-MM-DD')
```

```
   and deal_date < TO_DATE('2015-06-01', 'YYYY-MM-DD');
执行计划
--------
| Id  | Operation                | Name           | Rows  |Cost (%CPU)|Pstart| Pstop |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                |     1 |  170   (0)|      |       |
|   1 |  SORT AGGREGATE          |                |     1 |           |      |       |
|   2 |   PARTITION RANGE SINGLE|                |    22 |  170   (0)|    5 |     5 |
|   3 |    TABLE ACCESS FULL     | RANGE_PART_TAB |    22 |  170   (0)|    5 |     5 |
-------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        568  consistent gets
```

脚本 8-28 分区表 COUNT 普通写法的性能

语句 4：

```
select count(*) max_nbr from range_part_tab partition(p_201505);
执行计划
--------
| Id  | Operation                | Name          | Rows  | Cost (%CPU)|Pstart| Pstop |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |               |     1 |     8   (0)|      |       |
|   1 |  SORT AGGREGATE          |               |     1 |            |      |       |
|   2 |   PARTITION RANGE SINGLE|               |  8716 |     8   (0)|    5 |     5 |
|   3 |    INDEX FAST FULL SCAN | IDX_PART_NBR  |  8716 |     8   (0)|    5 |     5 |
-------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
         29  consistent gets
```

脚本 8-29 分区表 COUNT “特殊”写法的性能

2. 啥时分区索引性能反而低

假设有两张表 part_tab 和 norm_tab，前者是分区表，后者是普通表，并且记录数都一样。接下来在两个表的 col2 列上都有索引的情况下，我们来比较一下 select * from part_tab where col2=8 和 select * from norm_tab where col2=8 这两个语句的性能。

首先是环境准备，分别建分区表和普通表，并且在分区表的 col2 列建局部索引，在普通表的 col2 列建索引，如下：

```
drop table part_tab purge;
create table part_tab (id int,col2 int,col3 int)
       partition by range (id)
       (
```

```
        partition p1 values less than (10000),
        partition p2 values less than (20000),
        partition p3 values less than (30000),
        partition p4 values less than (40000),
        partition p5 values less than (50000),
        partition p6 values less than (60000),
        partition p7 values less than (70000),
        partition p8 values less than (80000),
        partition p9 values less than (90000),
        partition p10 values less than (100000),
        partition p11 values less than (maxvalue)
        )
        ;
insert into part_tab select rownum,rownum+1,rownum+2 from dual connect by rownum
<=110000;
commit;
create  index idx_par_tab_col2 on part_tab(col2) local;
create  index idx_par_tab_col3 on part_tab(col3) ;

drop table norm_tab purge;
create table norm_tab  (id int,col2 int,col3 int);
insert into norm_tab select rownum,rownum+1,rownum+2 from dual connect by rownum
<=110000;
commit;
create  index idx_nor_tab_col2 on norm_tab(col2) ;
create  index idx_nor_tab_col3 on norm_tab(col3) ;
```

脚本 8-30　构造分区表和普通表的环境

接下来，我们针对 part_tab 进行查询。如下：

```
set autotrace traceonly
set linesize 1000
set timing on
select * from part_tab where col2=8 ;
执行计划
----------------------------------------------------------------------------------------------------
|Id|Operation                          |Name           |Rows|Bytes|Cost (%CPU)|Time    |Pstart|Pstop|
----------------------------------------------------------------------------------------------------
| 0|SELECT STATEMENT                   |               |   1|   39|   13   (0)|00:00:01|      |     |
| 1| PARTITION RANGE ALL               |               |   1|   39|   13   (0)|00:00:01|    1 |   11|
| 2|  TABLE ACCESS BY LOCAL INDEX ROWID|PART_TAB       |   1|   39|   13   (0)|00:00:01|    1 |   11|
|*3|   INDEX RANGE SCAN                |IDX_PAR_TAB_COL2|   1|     |   12   (0)|00:00:01|    1 |   11|
----------------------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         24  consistent gets
          0  physical reads
```

```
          0  redo size
        539  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-31　分区表局部分区扫描的情况

继续针对 norm_tab 查询，如下：

```
select * from norm_tab where col2=8 ;
执行计划
----------------------------------------------------------------------------------------------
| Id  |Operation                    | Name            |Rows  |Bytes |Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT             |                 |    1 |   39 |    2   (0)| 00:00:01|
|   1 | TABLE ACCESS BY INDEX ROWID| NORM_TAB        |    1 |   39 |    2   (0)| 00:00:01|
|*  2 |  INDEX RANGE SCAN           | IDX_NOR_TAB_COL2|    1 |      |    1   (0)| 00:00:01|
----------------------------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        543  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-32　普通表索引扫描的情况

我们发现，普通表 norm_tab 对应的 SQL 的 consistent gets 为 4，而分区表 part_tab 对应的 SQL 的 consistent gets 为 24，这是怎么回事？学会用分区表，性能反而更差了，还不如不学啊！

其实真正的原因是，这个分区表有分区，但是 SQL 语句却没有分区条件，导致该分区表的局部索引从 Pstart 1 扫描到 Pstop 11，遍历 11 个分区。等于访问了 11 个小索引。索引的大小和高度是有差异的，大小差别很大，高度差距非常小（甚至不变），这在前面已经提起过了。假设普通索引的高度是 3，每个分区小索引的高度是 2，那 11×2 也远大于 3，这就是性能有巨大差异的原因。

所以当分区表的分区条件无法加上时，全局索引性能要好于分区索引。

3. 同时取最大值和最小值的案例

环境准备，如下：

```
drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum;
commit;
alter table t add constraint pk_object_id primary key (OBJECT_ID);
set autotrace on
set linesize 1000
```

接下来我们同时取最大和最小值，看看执行计划是什么：

```
set linesize 1000
set autotrace on

select max(object_id),min(object_id) from t;
执行计划
--------
| Id  | Operation             | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |              |     1 |    13 |    46   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |              |     1 |    13 |            |          |
|   2 |   INDEX FAST FULL SCAN| PK_OBJECT_ID | 74796 |   949K|    46   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        160  consistent gets
          0  physical reads
          0  redo size
        502  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-33　同时取最大值和最小值的语句写法

发现执行计划并没有走高效的 INDEX FULL SCAN (MIN/MAX)扫描方式，而只是走了 INDEX FAST FULL SCAN 模式。主要是因为 Oracle 不能同时在索引根的两段寻找最大值和最小值，而且分开写成 select min(object_id) from t 和 select max(object_id) from t 两条语句也是不行的，它们是不等价的。那该怎么优化呢？

这里我们可以利用笛卡儿乘积来完成如下编写，由于各自都是 1 条记录，故相乘也还是 1，这可以确保记录不会出错，如下：

```
select max, min
  from (select max(object_id) max from t ) a,
       (select min(object_id) min from t ) b;
执行计划
--------
|Id |Operation                    |Name         |Rows  | Bytes |Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT             |             |    1 |    26 |    4   (0)| 00:00:01|
| 1 | NESTED LOOPS                |             |    1 |    26 |    4   (0)| 00:00:01|
| 2 |  VIEW                       |             |    1 |    13 |    2   (0)| 00:00:01|
| 3 |   SORT AGGREGATE            |             |    1 |    13 |           |         |
| 4 |    INDEX FULL SCAN (MIN/MAX)|PK_OBJECT_ID|    1 |    13 |    2   (0)| 00:00:01|
| 5 |  VIEW                       |             |    1 |    13 |    2   (0)| 00:00:01|
| 6 |   SORT AGGREGATE            |             |    1 |    13 |           |         |
| 7 |    INDEX FULL SCAN (MIN/MAX)|PK_OBJECT_ID|    1 |    13 |    2   (0)| 00:00:01|
-------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        480  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 8-34 同时取最大值和最小值的语句的改造写法

这下在执行计划中出现了两个 INDEX FULL SCAN (MIN/MAX)，总的逻辑读才 4，远低于 INDEX FAST FULL SCAN 的 160，性能大幅度提升了。

8.3.2 组合索引的经典案例

1. 组合索引的写法

```
drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum ;
create index idx_id_type on t(object_id,object_type);

UPDATE t SET OBJECT_ID=20 WHERE ROWNUM<=26000;
UPDATE t SET OBJECT_ID=21 WHERE OBJECT_ID<>20;
COMMIT;
```

```
set linesize 1000
set pagesize 1
alter session set statistics_level=all ;
select  /*+index(t,idx1_object_id)*/ * from t  where object_TYPE='TABLE'  AND OBJECT_ID >= 20
AND OBJECT_ID<= 21;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------
| Id |Operation                    |Name       |Starts |E-Rows |A-Rows |   A-Time   |Buffers |
-------------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |           |     1 |       |  2939 |00:00:00.02 |   1117 |
|   1| TABLE ACCESS BY INDEX ROWID|T          |     1 |  3411 |  2939 |00:00:00.02 |   1117 |
|*  2|  INDEX RANGE SCAN           |IDX_ID_TYPE|     1 |   299 |  2939 |00:00:00.02 |    736 |
-------------------------------------------------------------------------------------------------
2 - access("OBJECT_ID">=20 AND "OBJECT_TYPE"='TABLE' AND "OBJECT_ID"<=21)
    filter("OBJECT_TYPE"='TABLE')
已选择 25 行。
```

脚本 8-35　组合索引与范围写法

请看写法 2，差异在于 OBJECT_ID IN (20,21)这部分，性能差别比较明显，Buffers 从 1117 下降到 598，性能提升不少，如下：

```
select  /*+index(t,idx_id_type)*/ * from t t where object_TYPE='TABLE'  AND  OBJECT_ID IN
(20,21);
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------
| Id|Operation                    | Name       |Starts |E-Rows |A-Rows |   A-Time   |Buffers|
-------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT             |            |     1 |       |  2939 |00:00:00.01|    598|
|  1| INLIST ITERATOR             |            |     1 |       |  2939 |00:00:00.01|    598|
|  2|  TABLE ACCESS BY INDEX ROWID| T          |     2 |  3411 |  2939 |00:00:00.01|    598|
|* 3|   INDEX RANGE SCAN          | IDX_ID_TYPE|     2 |     1 |  2939 |00:00:00.01|    217|
-------------------------------------------------------------------------------------------3
- access((("OBJECT_ID"=20 OR "OBJECT_ID"=21)) AND "OBJECT_TYPE"='TABLE')
已选择 25 行。
```

脚本 8-36　组合索引与 In 写法

这里需要注意的是，OBJECT_ID IN (20,21)与 object_TYPE='TABLE' AND OBJECT_ID >= 20 AND OBJECT_ID<= 21 实际并不等价，但是从业务的角度来说，不会有小数，所以是等价的。这里为啥性能会有差距，主要是因为当第 1 列是等值的时候，第 2 列用到索引检索到数据，可以停止检索的步伐（比如查到 LJB,发现下一个是 MBB，由于 M 排序在 L 后面，知道再也不会有 LJB 了）。而第 1 列是范围的时候，第 2 列用索引检索到数据也不能停下来

2. 组合索引与增加检索条件

依然是关于 IN 优化（col1、col2、col3 的索引情况，如果没有为 COL2 赋予查询条件，则 COL3 只能起到检验作用)。

首先是环境准备，如下：

```
drop table t purge;
create table t as select * from dba_objects;
UPDATE t SET OBJECT_ID=20 WHERE ROWNUM<=26000;
UPDATE t SET OBJECT_ID=21 WHERE OBJECT_ID<>20;
Update t set object_id=22 where rownum<=10000;
COMMIT;

create index idx_union on t(object_type,object_id,owner);
set autotrace off
alter session set statistics_level=all ;
set linesize 1000
```

写法1，请看如下SQL：

```
select * from t where object_type='VIEW' and OWNER='LJB'
Plan hash value: 1570829420
-------------------------------------------------------------------------------------------------
| Id |Operation                    |Name      |Starts |E-Rows |A-Rows |   A-Time    |Buffers |
-------------------------------------------------------------------------------------------------
|   1| TABLE ACCESS BY INDEX ROWID|T         |     1 |    11 |     5 |00:00:00.01 |     24 |
|*  2|  INDEX RANGE SCAN          |IDX_UNION |     1 |    50 |     5 |00:00:00.01 |     20 |
-------------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("OBJECT_TYPE"='VIEW' AND "OWNER"='LJB')
       filter("OWNER"='LJB')
Note
-----
   - dynamic sampling used for this statement
```

脚本 8-37 未增加 OBJECT_ID 列的写法

写法2，这时增加and OBJECT_ID IN (20,21,22)条件后，效果如下：

```
select /*+index(T IDX_UNION)*/
 * from t T where object_type='VIEW'
and OBJECT_ID IN (20,21,22)
AND OWNER='LJB';
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
Plan hash value: 306189815
-------------------------------------------------------------------------------------------------
| Id  |Operation                    | Name     |Starts |E-Rows |A-Rows |   A-Time    |Buffers|
-------------------------------------------------------------------------------------------------
|   1 | INLIST ITERATOR             |          |     1 |       |     5 |00:00:00.01|     14|
|   2 |  TABLE ACCESS BY INDEX ROWID| T        |     3 |     1 |     5 |00:00:00.01|     14|
```

```
|*  3 |   INDEX RANGE SCAN          | IDX_UNION|     3 | 12452 |     5 |00:00:00.01|     10|
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("OBJECT_TYPE"='VIEW' AND (("OBJECT_ID"=20 OR "OBJECT_ID"=21 OR
              "OBJECT_ID"=22)) AND "OWNER"='LJB')
Note
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------
   - dynamic sampling used for this statement
```

脚本 8-38　增加 OBJECT_ID 列的写法

写法 1 的 Buffers 是 24，写法 2 是 14，性能有明显差异，问题在哪里呢？两个语句等价吗？

很显然，当表中 OBJECT_ID 列的记录只有 20、21、22 三个取值时，两个语句是等价的。而 3 列组合索引的特点是，第 2 列脱离第 1 列无意义，第 3 列脱离第 2 列无意义。现在正好是中间这列无条件，补上后，第 2 列有意义，第 3 列也有意义，所以性能大幅提升了。

8.4　本章习题、总结与延伸

习题 1：说说索引三大特性是什么，能应用在哪些 SQL 上？

习题 2：说说用组合索引需要考虑什么问题。

习题 3：分区表中的聚合语句有什么特别之处？

习题及疑问的邮件发送地址与本章总结及解题二维码如下图：

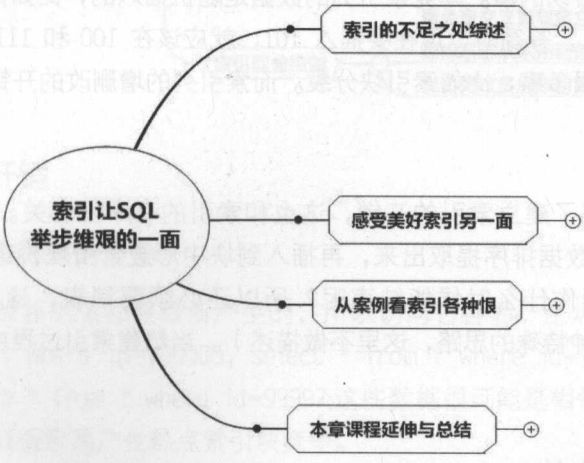
第 9 章　且慢，弄清索引之阻碍让 SQL 飞

原来索引是一把双刃剑

任何事物都有它的两面性，索引也不例外，上一章中我们介绍了索引的各种好处，本章我们主要来阐述索引的坏处。

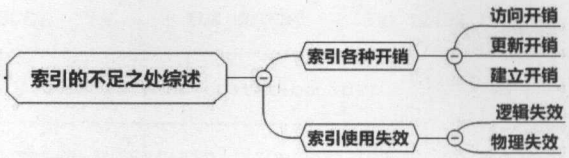
大家可能没注意到，上一章中主要都是一些查询语句，那如果更新语句出现，索引的缺点将很明显。因为索引本身是有序的，而更新数据的过程中也要更新索引，更新完后还要保持索引的有序性，这就需要付出很大的开销了，索引不好的一面就从这里开始显示出来了。

本章我们先简要综述一下索引的不足之处有哪些，接下来详细阐述，随后一起和大家探索各种工作中的案例，最后为思考回顾。总体学习思路如下图所示：



9.1　索引的不足之处

关于索引的不足之处我们可以从索引的开销和容易失效这两个方面来讨论，如下图所示：



9.1.1 索引的各种开销

还记得前面关于索引结构的分析吗？通过系列步骤，我们明白了索引的结构，推导出索引的三大特性，并应用这些特性让 SQL 跑得更快。

这只是索引好的一面。真正有问题的一面被掩盖了。都有什么问题呢？

1. 热块竞争

你看，索引最新的数据块一般是在最右边，而我们访问数据时正常来说也是访问比较新的数据，历史数据很少有人关注。然后问题来了，大家都一起访问最新的数据，不是都集中于同一个目标来访问了吗？这就很容易产生热块竞争。

2. 回表开销

另外，大家都知道索引存储索引列的值和 rowid，通过 rowid 来定位回到表中。其实这个回到表中的开销也是很大，具体情况我们随后可以了解到。

3. 更新开销

索引的有序性是一个非常重要的特性，这个特性能够消除排序等开销，但是索引块要保持有序性，可不是一件容易的事。毕竟索引列的数据是随机插入的，比如你在原来的索引列中存储的是 100、110、111，等等时，现在要插入 101，就应该在 100 和 111 之间插入，为了保证这个顺序索引需要做很多事，比如索引块分裂。而索引列的增删改的开销是很大的。

4. 建立开销

还有千万别忽略了建立索引的开销，这也和索引的有序性有关。我们在建索引的过程中，首先把索引列的数据排序提取出来，再插入到块中形成索引块，这时如果数据不断地插入，排序提取这个动作什么时候能结束呢？所以还必须要锁表，这就是一个很大的开销（online 建索引是一种特殊的思路，这里不做描述）。当然建索引过程中排序这个动作本身也是一个不小的开销。

9.1.2 索引使用失效

索引的不足之处除了上述的几点外，从另一个维度看，还会有失效的可能。我们现在知道建索引对查询一般比较有利，对更新一般比较有害。不过有的时候，虽然建了索引，但其对查询毫无帮助，这种情况还是有的。比如索引失效了，这分为逻辑失效和物理失效两种。

1. 逻辑失效

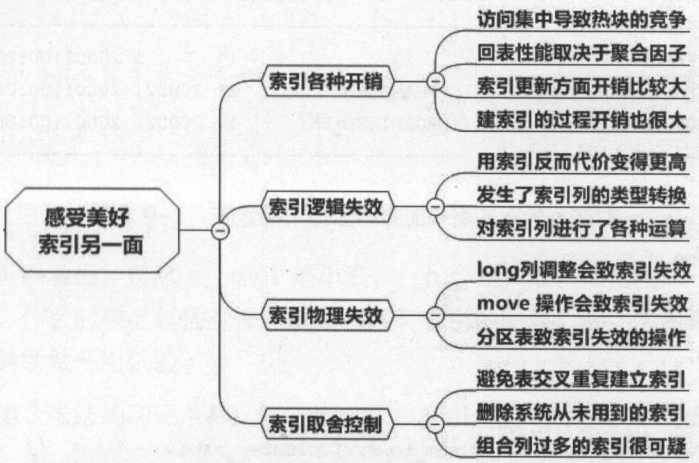
逻辑失效是索引本身并没有真正失效，只是由于写法的问题导致索引用不上，比如对 SQL 的条件列进行运算，类似 select * from t where upper(name)='ABC'等，这时在 name 列上建了 Btree 索引是用不上的。再或者比如被人强制用了全表扫描的 Hint 等导致数据库被迫不用索引，等等。

2. 物理失效

物理失效就是索引真的失效了，比如被人误设了 unusable 动作，或者是一些类似分区表的不规范操作导致的索引失效。对此后续有详细的例子说明。

9.2　感受美好索引另一面

前面简要描述了索引的不足之处，接下来我们进行更加详细的展开说明，具体细节如下：



9.2.1　索引各种开销

1. 访问集中导致热块的竞争

由于一般来说，最新的值都是最新产生的，所以访问它容易产生热块竞争。举例来说，如：select * from t where id=100000，select * from t where id=99999;select * from t where id=99998;select * from t where id=99997;这些数据很可能是相邻的，那么它们就会在同一个索引块上，这样很容易产生热点索引块竞争。

2. 回表性能取决于聚合因子

结论：索引查询要尽可能避免回表，如果不可避免，则需要关注聚合因子是否过大。（注：这个例子在前面的章节已经说过了，这里就不再详述了。）在该例子中，构造脚本

ORGANIZED 表的聚合因子比较小，回表的代价较低，产生了 2900 个 BUFFER，如下：

```
select /*+ index( colocated colocated_pk ) */ * from colocated where x between 20000 and 40
SELECT * FROM table(dbms_xplan.display_cursor(NULL,NULL,'runstats_last'));
-------------------------------------------------------------------------
| Id |Operation                    |Name         |Starts|E-Rows|A-Rows|   A-Time   |Buffers |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |             |     1|      | 20001|00:00:00.05 |   2900 |
|   1| TABLE ACCESS BY INDEX ROWID|COLOCATED    |     1| 20002| 20001|00:00:00.05 |   2900 |
|*  2|  INDEX RANGE SCAN           |COLOCATED_PK|     1| 20002| 20001|00:00:00.03 |   1375 |
-------------------------------------------------------------------------------------------
```

而 DISORGANIZED 表的聚合因子比较大，回表的代价很高，如下，产生 21360 个 BUFFER：

```
---两者性能差异显著,DISORGANIZED 表的聚合因子比较大，回表的代价很高，如下，产生 21360 个 BUFFER
select /*+ index( disorganized disorganized_pk ) */* from disorganized where x
between 20000 and 40000;
SELECT * FROM table(dbms_xplan.display_cursor(NULL,NULL,'runstats_last'));
-------------------------------------------------------------------------
| Id |Operation                    |Name           |Starts|E-Rows|A-Rows|   A-Time   |Buffers|
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |               |     1|      | 20001|00:00:00.09|  21360|
|   1| TABLE ACCESS BY INDEX ROWID|DISORGANIZED   |     1| 20002| 20001|00:00:00.09|  21360|
|*  2|  INDEX RANGE SCAN           |DISORGANIZED_PK|     1| 20002| 20001|00:00:00.03|   1375|
-------------------------------------------------------------------------------------------
```

```
select a.table_name,
       a.index_name,
       a.blevel,
       a.leaf_blocks,
       b.num_rows,
       b.blocks,
       a.clustering_factor,
       trunc(a.clustering_factor / b.num_rows,2) cluster_rate
  from user_indexes a, user_tables b
 where a.table_name = b.table_name
     and index_name in ('COLOCATED_PK', 'DISORGANIZED_PK' )
     and a.clustering_factor is not null
 order by cluster_rate desc;

TABLE_NAME   INDEX_NAME     BLEVEL LEAF_BLOCKS  NUM_ROWS BLOCKS CLUSTERING_FACTOR CLUSTER_RATE
----------   ----------     ------ ----------- ---------- ------ ----------------- ------------
DISORGANIZED DISORGANIZED_PK     1         208     100000   1219             99927          .99
COLOCATED    COLOCATED_PK        1         208     100000   1252              1190          .01
```

3. 索引更新方面的开销比较大

环境搭建：

```
drop table t_big purge;
drop table t purge;
```

```
create table t as select * from dba_objects;
set autotrace off
create table t_big as select * from t ;
insert into  t_big select * from t_big;
insert into  t_big select * from t_big;
insert into  t_big select * from t_big;
insert into  t_big select * from t_big;
insert into  t_big select * from t_big;
insert into  t_big select * from t_big;
commit;

drop table  t_small purge;
create table t_small as select * from t where rownum<=1000;
```

```
set timing on
insert into t_small select * from t_big;
已创建 4684096 行。
已用时间:  00: 00: 28.46
commit;

insert into t_big select * from t_big;
已创建 4684096 行。
已用时间:  00: 00: 28.22
commit;
```

脚本 9-1　无索引，表记录增加，插入不怎么变慢

由上面代码可以看出，虽然 t_small 是小表，t_big 是大表。但是插入一般不会随着记录的增加越插越慢。什么时候会越插越慢，就是当表有索引的时候。因为索引需要维护，越大维护越困难。我们继续做一组试验。

环境准备（建 3 张结构和记录都一样的表，只是索引分别是 6 个、2 个及无索引）：

```
drop table test1 purge;
drop table test2 purge;
drop table test3 purge;
drop table t purge;
create table t as select * from dba_objects;
create table test1 as select * from t;
create table test2 as select * from t;
create table test3 as select * from t;
create index idx_owner on test1(owner);
create index idx_object_name on test1(object_name);
create index idx_data_obj_id on test1(data_object_id);
create index idx_created on test1(created);
create index idx_last_ddl_time on test1(last_ddl_time);
create index idx_status on test1(status);
create index idx_t2_sta on test2(status);
create index idx_t2_objid on test2(object_id);
```

分别往这三张表里插记录：

```
SQL> set timing on
SQL> --语句 1(test1 表有 6 个索引)
SQL> insert into test1 select * from t;

已创建 111128 行。

已用时间:  00: 00: 06.08
SQL> commit;

提交完成。

已用时间:  00: 00: 00.00
SQL> --语句 2(test2 表有 2 个索引)
SQL> insert into test2 select * from t;

已创建 111128 行。

已用时间:  00: 00: 03.59
SQL> commit;

提交完成。

已用时间:  00: 00: 00.00
SQL> --语句 3(test3 表有无索引)
SQL> insert into test3 select * from t;

已创建 111128 行。

已用时间:  00: 00: 00.37
SQL> commit;

提交完成。

已用时间:  00: 00: 00.00
```

脚本 9-2　有索引的表，记录越多，插入越慢

表记录越大，索引越多，插入越慢，从试验结果来看，这一点还是非常明显的。

4. 建索引的过程开销也很大

（1）建索引过程会产生全表锁

```
drop table t purge;
create table t as select * from dba_objects;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
```

```
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
commit;
select sid from v$mystat where rownum=1;
--12
set timing on
create index idx_object_id on t(object_id);
索引已创建。
```

```
session 2
sqlplus ljb/ljb
set linesize 1000
select sid from v$mystat where rownum=1;
--134
--以下执行居然被阻塞，要直至建索引结束后，才能执行
update t set object_id=99999 where object_id=8;
```

可以通过如下方式查看被锁的情况：

```
打开一个新的窗口
set linesize 1000
select * from v$lock where sid in (12,134);

SQL> select * from v$lock where sid in (134,12);

ADDR     KADDR      SID TY        ID1        ID2   LMODE    REQUEST      CTIME   BLOCK
-------- -------- ----- -- ---------- ---------- ------- ---------- ---------- -------
2EB79320 2EB7934C    12 AE        100          0       4          0        409       0
2EB79394 2EB793C0   134 AE        100          0       4          0        254       0
2EB79408 2EB79434    12 TO      65921          1       3          0        402       0
2EB79574 2EB795A0    12 DL     106831          0       3          0         12       0
2EB795E8 2EB79614    12 DL     106831          0       3          0         12       0
0EDD7A9C 0EDD7ACC   134 TM     106831          0       0          3         10       0
0EDD7A9C 0EDD7ACC    12 TM     106831          0       4          0         12       1
0EDD7A9C 0EDD7ACC    12 TM         18          0       3          0         12       0
2C0D2844 2C0D28B0    12 TS          8   25202162       6          0          4       0
2C1A2A8C 2C1A2ACC    12 TX     393223      31633       6          0         12       0

select  /*+no_merge(a) no_merge(b) */
(select username from v$session where sid=a.sid) blocker,a.sid, 'is blocking',
(select username from v$session where sid=b.sid) blockee,b.sid
from v$lock a,v$lock b
where a.block=1 and b.request>0
and a.id1=b.id1
and a.id2=b.id2;

BLOCKER        SID 'ISBLOCKING BLOCKEE         SID
---------------- ---------- ------------------
LJB             12 is blocking LJB            134
```

脚本 9-3　建索引产生锁

（2）建索引过程会产生全表排序

未建索引前，观察一下数字字典中记录的系统排序情况，如下：

```
set linesize 266
drop table t purge;
create table t as select * from dba_objects;

select t1.name, t1.STATISTIC#, t2.VALUE
  from v$statname t1, v$mystat t2
 where t1.STATISTIC# = t2.STATISTIC#
   and t1.name like '%sort%';

NAME                                                             STATISTIC#      VALUE
---------------------------------------------------------------- ---------- ----------
sorts (memory)                                                          565        462
sorts (disk)                                                            566          0
sorts (rows)                                                            567       2174
```

建索引后，继续观察，发现排序次数 sorts (memory)增加了，如下：

```
create index idx_object_id on t(object_id);

select t1.name, t1.STATISTIC#, t2.VALUE
  from v$statname t1, v$mystat t2
 where t1.STATISTIC# = t2.STATISTIC#
   and t1.name like '%sort%';

NAME                                                             STATISTIC#      VALUE
---------------------------------------------------------------- ---------- ----------
sorts (memory)                                                          565        463
sorts (disk)                                                            566          0
sorts (rows)                                                            567      75292
```

脚本 9-4　建索引产生排序

索引各种开销的总结：

索引开销分析		
索引开销		**原因**
热块竞争		最新索引块都集中在最右边，而最新数据被访问频率最高，所以热块竞争产生
回表开销		索引存储索引列的值和 rowid，并可以通过 rowid 定位回到表中，得到这个索引列以外的列信息，要做这类事，必然有开销
更新开销		索引本身有序，更新后数据变化了，还要保持有序就必须做很多事来保证有序，所以有开销
建立开销	产生全表锁	加锁确保有序取索引列的数据的操作不会随着记录的不断插入而永远无法结束
	产生排序	索引本身是有序的，所以要顺序取出索引列的值并将它们插入 Block 中，这些动作当然要排序

9.2.2　索引使用失效

1. 索引逻辑失效

（1）用索引反而代价变得更高

这个道理比较简单，如果应用索引范围检索数据，返回大量记录且几乎是所有的记录，这时候用索引肯定有错，索引范围查询访问一般适合返回少量记录的情况，否则用全表扫描或者全索引扫描就可以。

（2）发生索引列的类型转换

在表字段设计的时候有一个非常重要的原则，什么类型的字段存什么类型的值，否则就会发生类型转化，具体请看如下例子：

```
drop table t_col_type purge;
create table t_col_type(id varchar2(20),col2 varchar2(20),col3 varchar2(20));
insert into t_col_type select rownum,'abc','efg' from dual connect by level<=10000;
commit;
create index idx_id on t_col_type(id);
set linesize 1000
set autotrace traceonly

select * from t_col_type where id=6;

执行计划
----------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     1 |    36 |     9   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T_COL_TYPE |     1 |    36 |     9   (0)| 00:00:01 |
----------------------------------------------------------------------------------
   1 - filter(TO_NUMBER("ID")=6)
统计信息
--------
          0  recursive calls
          0  db block gets
         32  consistent gets
          0  physical reads
          0  redo size
        540  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

实际上只有如下写法才可以用到索引，这个很不应该，如果什么类型的取值就设置什么样的字段，把 id 字段类型改为 Number，就顺畅了，如下：

```
select * from t_col_type where id='6';
执行计划
--------
| Id | Operation                    | Name       |Rows  |Bytes |Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0| SELECT STATEMENT             |            |    1 |   36 |    2   (0)| 00:00:01 |
|   1|  TABLE ACCESS BY INDEX ROWID| T_COL_TYPE |    1 |   36 |    2   (0)| 00:00:01 |
|*  2|   INDEX RANGE SCAN           | IDX_ID     |    1 |      |    1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
   2 - access("ID"='6')
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        544  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-5　索引列的类型转换

（3）对索引进行了各种运算

对索引列进行了各种运算，详见后面的案例部分。

2. 索引物理失效

（1）long 列调整导致索引失效

环境准备（建表，建 long 字段）：

```
drop table t purge;
create table t (object_id number,object_name long);
create index idx_object_id on t(object_id);
insert into t values (1,'ab');
commit;

select t.status,t.index_name from user_indexes t where index_name='IDX_OBJECT_ID';
STATUS   INDEX_NAME
-------- ---------------
VALID    IDX_OBJECT_ID
```

接下来将 long 修改为 clob，发现索引失效了，必须重建索引，如下：

```
alter table T modify object_name clob;
set autotrace off
```

```
select t.status,t.index_name from user_indexes t where index_name='IDX_OBJECT_ID';
STATUS   INDEX_NAME
-------- --------------
UNUSABLE IDX_OBJECT_ID

alter index  idx_object_id rebuild;
set autotrace off
select t.status,t.index_name from user_indexes t where index_name='IDX_OBJECT_ID';
STATUS   INDEX_NAME
-------- ---------------
VALID    IDX_OBJECT_ID
```

脚本 9-6　long 列调整导致索引失效

（2）move 操作会导致索引失效

Move 是一个危险系数非常高的操作，它可以收缩表降低高水平位，却会导致索引失效，因而需要重建索引，请看下面例子：

```
drop table t purge;
create table t as select * from dba_objects where object_id is not null;
alter table t modify object_id not null;
set autotrace off
insert into t select * from t;
insert into t select * from t;
commit;
create index idx_object_id on t(object_id);
select index_name,status from user_indexes where index_name='IDX_OBJECT_ID';
INDEX_NAME                     STATUS
-------------------------------------
IDX_OBJECT_ID                  VALID
alter table t move;
select index_name,status from user_indexes where index_name='IDX_OBJECT_ID';
INDEX_NAME                      STATUS
--------------------------------------
IDX_OBJECT_ID                  UNUSABLE
```

脚本 9-7　move 操作导致索引失效

（3）分区表导致索引失效的操作

这在前面已经描述过了，这里就不再重复了，请读者自行回到前面的章节进行复习总结。归纳如下：

- truncate 分区会导致全局索引失效，不会导致局部索引失效。如果对 truncate 增加 update global indexes，则全局索引不会失效。
- drop 分区会导致全局索引失效，局部索引因为 drop 分区，所以也不存在该分区的局部索引了。如果对 drop 分区增加 update global indexes，全局索引不会失效。

- split 分区会导致全局索引失效，也会导致局部索引失效。如果对 split 分区增加 update global indexes，则全局索引不会失效。
- add 分区不会导致全局索引失效，也不会导致局部索引失效。
- exchange 会导致全局索引失效，不会导致局部索引失效。如果对 exchange 分区增加 update global indexes，则全局索引不会失效。

重要结论：

- 所有的全局索引，只要用到 update global indexes ,都不会失效，其中 add 分区甚至不需要增加 update global indexes 都可以生效。
- 局部索引的操作都不会失效，除了 split 分区。切记 split 分区的时候，要将局部索引进行 rebuild。

索引失效分析		
失效模式		说明
逻辑失效	用索引反而代价变得更高	索引范围查询访问一般适合返回少量记录的情况，如果返回大量记录，用索引反而更慢！
	发生索引列的类型转换	select * from t_col_type where id=6; 1 - filter(TO_NUMBER("ID")=6)
	对索引进行了各种运算	select * from t where object_id/2=:id;
物理失效	long 列调整导致索引失效	alter table T modify object_name clob; select t.status,t.index_name from user_indexes t where index_name='IDX_OBJECT_ID'; STATUS INDEX_NAME --------- --------------- UNUSABLE IDX_OBJECT_ID
	move 操作会导致索引失效	alter table t move; select index_name,status from user_indexes where index_name='IDX_OBJECT_ID'; INDEX_NAME STATUS ---------- IDX_OBJECT_ID UNUSABLE
	分区表导致索引失效的操作	在关于表设计的章节中有过详细介绍

9.2.3 索引取舍控制

1. 避免表交叉重复建立索引

假如 t 表有 nbr、area_code 两列的联合索引，单列的 nbr 索引就显得多余，因为 nbr、area_code 索引可以用在单列 nbr 索引上，具体如下：

```
drop table t purge;
create table t as select * from dba_objects;
```

```
create index idx_object_id on t(object_id,object_type);
set autotrace traceonly
set linesize 1000
--以下就能用到索引，因为 object_id 列是前缀

select * from t where object_id=19;
执行计划
--------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |   207 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T              |     1 |   207 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_OBJECT_ID |     1 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
       1392  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-8　组合索引前缀与单列索引

2. 删除系统从未用到的索引

环境搭建，建表建索引并完成某列索引的监控：

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_t_id on t (object_id);
alter index idx_t_id monitoring usage;
select * from v$object_usage;
INDEX_NAME TABLE_NAME MONITORING USED      START_MONITORING           END_MONITORING
---------- ---------- ---------- ---------- ------------------------- -------------------
IDX_T_ID   T          YES        NO         12/14/2015 07:44:36
```

接下来继续执行一个用索引的查询，然后再观察 USED 字段，发现索引被用过：

```
set autotrace traceonly
select * from t where object_id=10;
set autotrace off
select * from v$object_usage;
INDEX_NAME TABLE_NAME MONITORING USED       START_MONITORING           END_MONITORING
---------- ---------- ---------- ---------- ------------------------- -------------------
IDX_T_ID   T          YES        YES        12/14/2015 07:44:36
```

停止对索引的监控，观察 v$object_usage 状态变化，发现 MONITORING 的值为 NO，且 END_MONITORING 记录了停止监控的时间，如下：

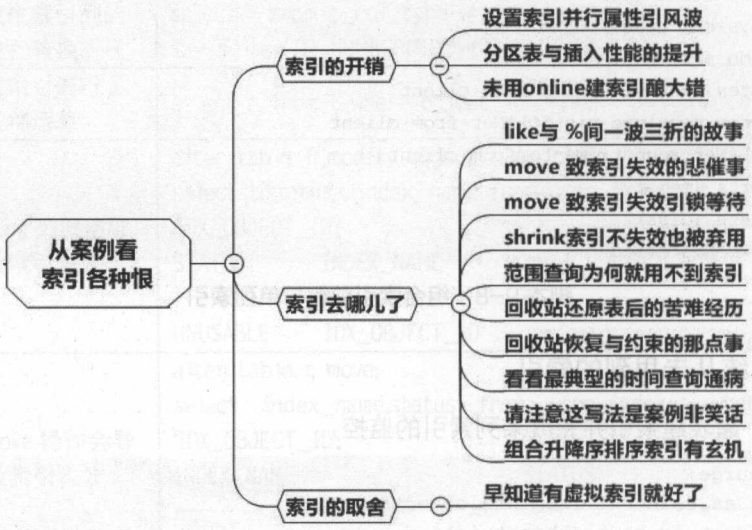
```
alter index idx_t_id nomonitoring usage;

INDEX_NAME TABLE_NAME MONITORING USED       START_MONITORING          END_MONITORING
------------------------------------------------------------------------------------------
IDX_T_ID       T          NO         YES      12/14/2015 07:44:36     12/14/2015 07:46:45
```

脚本 9-9　删除系统从未用到的索引

3. 组合列过多的索引很可疑

组合索引一般不宜过多，如果组合索引列达到 4 个以上，那这个索引本身就很大，就不一定高效。另外，索引更新也会出现比较大的性能问题。

9.3　从案例看索引各种恨



9.3.1　索引的开销

1. 设置索引并行属性引风波

某系统，为了提高建索引的效率，相关人员采用了并行建索引的方式，最终索引建成功后，并行度设到了索引的属性中去了，然后忘记将并行度取消。结果很多查询都调用并行，引起资源争用，最终引发了性能故障。

```
drop table t purge;
create table t as select * from dba_objects where object_id is not null;
alter table T modify object_id not null;
insert into t  select * from t;
```

```
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
commit;

set timing on
create index idx_object_id on t(object_id) parallel 8;
select index_name,degree from user_indexes where table_name='T';
INDEX_NAME                     DEGREE
------------------------------ -------
IDX_OBJECT_ID                  8
```

对应的执行计划如下：

```
set linesize 1000
set autotrace traceonly

select count(*) from t;
执行计划
--------
|Id|Operation                 |Name         |Rows |Cost(%CPU)|  TQ |IN-OUT|PQ Distrib|
-------------------------------------------------------------------------------------
| 0|SELECT STATEMENT          |             |   1 |5797   (2)|     |      |          |
| 1| SORT AGGREGATE           |             |   1 |          |     |      |          |
| 2|  PX COORDINATOR          |             |     |          |     |      |          |
| 3|   PX SEND QC (RANDOM)    |:TQ10000     |   1 |          |Q1,00| P->S |QC (RAND) |
| 4|    SORT AGGREGATE        |             |   1 |          |Q1,00| PCWP |          |
| 5|     PX BLOCK ITERATOR    |             |8100K|5797   (2)|Q1,00| PCWC |          |
| 6|      INDEX FAST FULL SCAN|IDX_OBJECT_ID|8100K|5797   (2)|Q1,00| PCWP |          |
-------------------------------------------------------------------------------------
统计信息
--------
         24  recursive calls
          0  db block gets
      25365  consistent gets
      20769  physical reads
          0  redo size
        426  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-10　索引的属性被设置为并行

这里可以看出，一个普通查询产生了并行操作，所有调用该表的语句都会产生并行动作，

最终必然会导致争用！一般来说，如果我们要进行并行操作，建议用 HINT 的方式给语句加并行属性，比如/*+parallel n*/。

2. 分区表与插入性能的提升

结论：

如果表上没有索引，插入的速度一般都不会慢，只有在有索引的情况下，才要考虑插入速度的优化。如果表有大量索引，一般来说，对于分区表的局部索引，由于只需要更新局部分区的索引，所以索引的开销会比较小，所以插入性能比有着相同的记录数、列及索引的普通表更快。

信不，觉得有道理？嗯，来，看看试验过程，首先是构造环境，如下：

```
--构造分区表，插入数据。
drop table range_part_tab purge;
create table range_part_tab (id number,deal_date date,area_code number,nbr1 number,
nbr2 number,nbr3 number,contents varchar2(4000))
          partition by range (deal_date)
          (
          partition p_201501 values less than (TO_DATE('2015-02-01', 'YYYY-MM-DD')),
          partition p_201502 values less than (TO_DATE('2015-03-01', 'YYYY-MM-DD')),
          partition p_201503 values less than (TO_DATE('2015-04-01', 'YYYY-MM-DD')),
          partition p_201504 values less than (TO_DATE('2015-05-01', 'YYYY-MM-DD')),
          partition p_201505 values less than (TO_DATE('2015-06-01', 'YYYY-MM-DD')),
          partition p_201506 values less than (TO_DATE('2015-07-01', 'YYYY-MM-DD')),
          partition p_201507 values less than (TO_DATE('2015-08-01', 'YYYY-MM-DD')),
          partition p_201508 values less than (TO_DATE('2015-09-01', 'YYYY-MM-DD')),
          partition p_201509 values less than (TO_DATE('2015-10-01', 'YYYY-MM-DD')),
          partition p_201510 values less than (TO_DATE('2015-11-01', 'YYYY-MM-DD')),
          partition p_201511 values less than (TO_DATE('2015-12-01', 'YYYY-MM-DD')),
          partition p_201512 values less than (TO_DATE('2016-01-01', 'YYYY-MM-DD')),
          partition p_201601 values less than (TO_DATE('2016-02-01', 'YYYY-MM-DD')),
          partition p_201602 values less than (TO_DATE('2016-03-01', 'YYYY-MM-DD')),
          partition p_201603 values less than (TO_DATE('2016-04-01', 'YYYY-MM-DD')),
          partition p_201604 values less than (TO_DATE('2016-05-01', 'YYYY-MM-DD')),
          partition p_max values less than (maxvalue)
          );
create index idx_parttab_id   on range_part_tab(id)         local;
create index idx_parttab_nbr1 on range_part_tab(nbr1)       local;
create index idx_parttab_nbr2 on range_part_tab(nbr2)       local;
create index idx_parttab_nbr3 on range_part_tab(nbr3)       local;
create index idx_parttab_area on range_part_tab(area_code)  local;
drop table normal_tab purge;

--接下来建普通表
create table normal_tab (id number,deal_date date,area_code number,nbr1 number,nbr2
number,nbr3 number,contents varchar2(4000));
```

```
create index idx_tab_id on normal_tab(id) ;
create index idx_tab_nbr1 on normal_tab(nbr1) ;
create index idx_tab_nbr2 on normal_tab(nbr2) ;
create index idx_tab_nbr3 on normal_tab(nbr3) ;
create index idx_tab_area on normal_tab(area_code) ;
```

接下来分别插入，比较性能：

```
SQL> set timing on
SQL> insert into range_part_tab
  2       select rownum,
  3
to_date( to_char(sysdate+60,'J')+TRUNC(DBMS_RANDOM.VALUE(0,60)),'J'),
  4               ceil(dbms_random.value(591,599)),
  5               ceil(dbms_random.value(18900000001,18999999999)),
  6               ceil(dbms_random.value(18900000001,18999999999)),
  7               ceil(dbms_random.value(18900000001,18999999999)),
  8               rpad('*',400,'*')
  9          from dual
 10        connect by rownum <= 400000;

已创建 400000 行。
已用时间:  00: 00: 51.20

SQL> insert into normal_tab
  2       select rownum,
  3
to_date( to_char(sysdate+60,'J')+TRUNC(DBMS_RANDOM.VALUE(0,60)),'J'),
  4               ceil(dbms_random.value(591,599)),
  5               ceil(dbms_random.value(18900000001,18999999999)),
  6               ceil(dbms_random.value(18900000001,18999999999)),
  7               ceil(dbms_random.value(18900000001,18999999999)),
  8               rpad('*',400,'*')
  9          from dual
 10        connect by rownum <= 400000;

已创建 400000 行。
已用时间:  00: 01: 20.04
```

脚本 9-11　分区表与插入性能的提升

普通表插入的时间是 1 分 20 秒，而分区表是 51 秒，看来结论果真不假。

3. 未用 online 建索引酿大错

结论：普通的对表建索引的操作将会导致针对该表的更新操作无法进行，需要等待索引建完。更新操作将会被建索引动作阻塞。而 ONLINE 建索引的方式却不会阻止针对该表的更新操作，与建普通索引相反的是，ONLINE 建索引的动作是反过来被更新操作阻塞。如下：

```
drop table t purge;
create table t as select * from dba_objects;
```

```
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
insert into t  select * from t;
commit;
select sid from v$mystat where rownum=1;
--12
set timing on
create index idx_object_id on t(object_id) online;
索引已创建。
session 2
sqlplus ljb/ljb
set linesize 1000
select sid from v$mystat where rownum=1;
--134
--以下执行居然不会被阻塞
update t set object_id=99999 where object_id=8;
```

接着分析：

```
session 3
set linesize 1000
SQL> select * from v$lock where sid in (134,12);
ADDR     KADDR             SID TY         ID1         ID2    LMODE   REQUEST     CTIME
BLOCK
-------- -------- ------ -- ---------- ---------- ------- -------- ---------- -------
2EB79320 2EB7934C     12 AE        100          0       4        0        278       0
2EB79394 2EB793C0    134 AE        100          0       4        0        303       0
2EB79408 2EB79434     12 DL     106831          0       3        0         25       0
2EB79574 2EB795A0     12 DL     106831          0       3        0         25       0
2EB795E8 2EB79614     12 OD     106831          0       4        0         25       0
2EB7965C 2EB79688     12 TX     131079      31688       0        4         11       0
0EDD7A9C 0EDD7ACC    134 TM     106831          0       3        0         23       0
0EDD7A9C 0EDD7ACC     12 TM     106831          0       2        0         25       0
0EDD7A9C 0EDD7ACC     12 TM     106834          0       4        0         25       0
2C17C3B8 2C17C3F8    134 TX     131079      31688       6        0         23       1
2C1A2448 2C1A2488     12 TX     589853      31754       6        0         25       0
已选择 11 行。

select  /*+no_merge(a) no_merge(b) */
(select username from v$session where sid=a.sid) blocker,
a.sid, 'is blocking',
(select username from v$session where sid=b.sid) blockee,
b.sid
from v$lock a,v$lock b
where a.block=1 and b.request>0
and a.id1=b.id1
```

```
and a.id2=b.id2;
BLOCKER          SID 'ISBLOCKING BLOCKEE          SID
------------------- ----------- --------------------
LJB              134 is blocking LJB               12
```

脚本 9-12　关于 online 方式建索引

9.3.2　索引去哪儿了

1. like 与 %间一波三折的故事

曾听说用 like 不能用索引，真是如此吗？看下面：

```
drop table t purge;
create table t as select * from dba_objects where object_id is not null;
set autotrace off
update t set object_id=rownum;
update t set object_name='AAALJB' where object_id=8;
update t set object_name='LJBAAA' where object_id=10;
commit;
create index idx_object_name on t(object_name);

SET AUTOTRACE ON
SET LINESIZE 1000

select object_name,object_id from t where object_name like 'LJB%';

OBJECT_NAME              OBJECT_ID
---------------------------------
LJBAAA                           10
LJB_TMP_SESSION               72521
LJB_TMP_SESSION               72910
LJB_TMP_TRANSACTION           72522
LJB_TMP_TRANSACTION           72911
已选择 5 行。

执行计划
--------
| Id  |Operation                    | Name            |Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT             |                 |    5|   395 |     6   (0)| 00:00:01 |
|   1 | TABLE ACCESS BY INDEX ROWID| T               |    5|   395 |     6   (0)| 00:00:01 |
|*  2 |  INDEX RANGE SCAN           | IDX_OBJECT_NAME |    5|       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          9  consistent gets
```

```
          0  physical reads
          0  redo size
        602  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
0 sorts (disk)
5  rows processed
```

脚本 9-13　like 不能用到索引吗

like 'LJB%'这种类型是可以用到索引的，比如检索到 M 打头，就知道不可能再有 L 打头出现了，当然能用到索引。不过如果是'%LJB%'就不行了，因为 Oracle 根本不知道自己的检索什么时候能停下来。

```
SQL> select object_name,object_id from t where object_name like '%LJB%';
执行计划
--------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    12 |   948 |   292   (1)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| T    |    12 |   948 |   292   (1)| 00:00:04 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1049  consistent gets
          0  physical reads
          0  redo size
        653  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          6  rows processed
```

当然，并不是说 Oracle 遇到这个情况一定要全表扫描，还有一种模式叫作全文检索，是可以用到索引的，这是后话，暂且不说。我们再看另外一种情况，就是'%LJB'，能用到索引吗？

其实其原理和'%LJB%'差不多，正常情况下检索也无法停下来，所以走索引也不太现实。不过我们有变通方法，可以通过 reverse 函数将'%LJB'转成 BJL%，然后用函数索引检索，就 OK 了，如下：

```
create index idx_reverse_objname on t(reverse(object_name));
set autotrace on
select object_name,object_id from t where reverse(object_name) like reverse('%LJB');

OBJECT_NAME                    OBJECT_ID
```

```
--------------------------- --
AAALJB                       8

执行计划
--------
|Id |Operation                    |Name                |Rows |Bytes |Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|  0|SELECT STATEMENT             |                    | 3596|  509K|  290   (0)| 00:00:04 |
|  1| TABLE ACCESS BY INDEX ROWID|T                    | 3596|  509K|  290   (0)| 00:00:04 |
|* 2|  INDEX RANGE SCAN           |IDX_REVERSE_OBJNAME|  647|      |    6   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
        496  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-14　like 用索引的巧妙例子

2. move 致索引失效的悲催事

在生产系统中，索引的失效往往意味着灾难性的故障。某大型制造业系统不小心由于表的 move 操作，导致索引失效。大致情况如下：

```
drop table t_p cascade constraints purge;
drop table t_c cascade constraints purge;

CREATE TABLE T_P (ID NUMBER, NAME VARCHAR2(30));
ALTER TABLE T_P ADD CONSTRAINT  T_P_ID_PK  PRIMARY KEY (ID);
CREATE TABLE T_C (ID NUMBER, FID NUMBER, NAME VARCHAR2(30));

ALTER TABLE T_C ADD CONSTRAINT FK_T_C FOREIGN KEY (FID) REFERENCES T_P (ID);

INSERT INTO T_P SELECT ROWNUM, TABLE_NAME FROM ALL_TABLES;
INSERT INTO T_C SELECT ROWNUM, MOD(ROWNUM, 1000) + 1, OBJECT_NAME  FROM ALL_OBJECTS;
COMMIT;

CREATE INDEX IND_T_C_FID ON T_C (FID);

SELECT TABLE_NAME,INDEX_NAME,STATUS FROM USER_INDEXES WHERE INDEX_NAME='IND_T_C_FID';
TABLE_NAME                     INDEX_NAME                     STATUS
------------------------------ ------------------------------ -------
```

```
T_C                              IND_T_C_FID                    VALID

--不小心失效了，比如操作了
ALTER TABLE T_C MOVE;

SELECT TABLE_NAME,INDEX_NAME,STATUS FROM USER_INDEXES WHERE INDEX_NAME='IND_T_C_FID';
TABLE_NAME                       INDEX_NAME                     STATUS
------------------------------------------------------------------------
T_C                              IND_T_C_FID                    UNUSABLE
```

脚本 9-15　move 表导致索引失效

结果查询性能成这样：

```
SET LINESIZE 1000
SET AUTOTRACE TRACEONLY
SELECT A.ID, A.NAME, B.NAME FROM T_P A, T_C B WHERE A.ID = B.FID AND A.ID = 880;
执行计划
--------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |    25 |  1500 |   111   (1)| 00:00:02 |
|   1 |  NESTED LOOPS                |            |    25 |  1500 |   111   (1)| 00:00:02 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T_P        |     1 |    30 |     0   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | T_P_ID_PK  |     1 |       |     0   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS FULL          | T_C        |    25 |   750 |   111   (1)| 00:00:02 |
-----------------------------------------------------------------------------------------
   3 - access("A"."ID"=880)
   4 - filter("B"."FID"=880)
统计信息
--------
          0  recursive calls
          0  db block gets
        394  consistent gets
          0  physical reads
          0  redo size
       3602  bytes sent via SQL*Net to client
        459  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         72  rows processed
```

脚本 9-16　索引失效后查询性能低下

将失效索引重建后，执行计划也改变了，逻辑读也从原来的 394 缩减为 81，性能大幅度提升。这里只是构造例子，实际情况表的记录巨大，索引访问变成全表扫描后，原先在 0.0 几秒瞬间可查出的记录变成超过 1 分钟才可以查出，严重影响了系统的使用，系统业务近乎停止。故障持续 15 分钟直至索引恢复正常，造成了难以估量的经济损失。

```
ALTER INDEX IND_T_C_FID   REBUILD;
查询性能是这样的:
SELECT A.ID, A.NAME, B.NAME FROM T_P A, T_C B WHERE A.ID = B.FID AND A.ID = 880;
执行计划
--------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    72 |  4320 |    87   (0)| 00:00:02 |
|   1 |  NESTED LOOPS                |             |    72 |  4320 |    87   (0)| 00:00:02 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T_P         |     1 |    30 |     0   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | T_P_ID_PK   |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| T_C         |    72 |  2160 |    87   (0)| 00:00:02 |
|*  5 |    INDEX RANGE SCAN          | IND_T_C_FID |    72 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
   3 - access("A"."ID"=880)
   5 - access("B"."FID"=880)
统计信息
--------
          0  recursive calls
          0  db block gets
         81  consistent gets
          0  physical reads
          0  redo size
       3602  bytes sent via SQL*Net to client
        459  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         72  rows processed
```

脚本 9-17　索引重建后性能恢复

结论：

又是一次 move table 引发的血案。这个案例，涉及有主外键的两表关联查询的性能问题，索引失效导致 NL 连接性能下降。关于 NL 连接，我们将在后续的章节中描述。

3. move 致索引失效引发锁等待

这是 move 表会导致索引失效的又一个故事。由于 move 外键所在的表，导致外键所在的表的索引失效，导致主外键的表更新起来举步维艰，频频被锁，如下：

```
--外键索引性能研究之准备
drop table t_p cascade constraints purge;
drop table t_c cascade constraints purge;

CREATE TABLE T_P (ID NUMBER, NAME VARCHAR2(30));
ALTER TABLE T_P ADD CONSTRAINT  T_P_ID_PK  PRIMARY KEY (ID);
```

```
CREATE TABLE T_C (ID NUMBER, FID NUMBER, NAME VARCHAR2(30));

ALTER TABLE T_C ADD CONSTRAINT FK_T_C FOREIGN KEY (FID) REFERENCES T_P (ID);
--以下操作导致外键索引失效
ALTER TABLE T_C MOVE;

外键索引删除后，立即有锁相关问题
--首先开启会话 1
select sid from v$mystat where rownum=1;
DELETE T_C WHERE ID = 2;
--接下来开启会话 2，也就是开启一个新的连接
select sid from v$mystat where rownum=1;

--然后执行如下操作进行观察
DELETE T_P WHERE ID = 2000;
--居然发现卡住半天不动了！
INSERT INTO T_P SELECT ROWNUM, TABLE_NAME FROM ALL_TABLES;
INSERT INTO T_C SELECT ROWNUM, MOD(ROWNUM, 1000) + 1, OBJECT_NAME  FROM ALL_OBJECTS;
COMMIT;
create index idx_IND_T_C_FID on T_C(FID);
```

脚本 9-18　外键索引失效导致锁表

假如外键有索引，就不会产生死锁情况，你可自行试验一下。

4. shrink 索引不失效也被弃用

可以把 alter table t move 降低高水平位导致索引失效的例子放在这里一起思考。这里用 alter table t shrink 的方式降低高水平位，结果避免了索引的失效，不过索引不失效了，是否索引就一定会被用到呢？请看下面的例子：

```
drop table t purge;
create table t as select * from dba_objects where object_id is not null;
alter table t modify object_id not null;
set autotrace off
insert into t select * from t;
insert into t select * from t;
commit;
create index idx_object_id on t(object_id);
set linesize 1000
set autotrace  on
select count(*) from t;
set autotrace off
delete from t where rownum<=292000;
commit;
set autotrace on
select count(*) from t;
alter table t enable row movement;
alter table t shrink space;
```

```
select count(*) from t;
执行计划
--------
Plan hash value: 2966233522
-------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     5   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    |   740 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
         15  consistent gets
          0  physical reads
          0  redo size
        424  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

奇怪，索引去哪儿了？怎么不走索引了？

```
set autotrace off
select index_name,status from user_indexes where index_name='IDX_OBJECT_ID';
INDEX_NAME                     STATUS
-----------------------------------
IDX_OBJECT_ID                  VALID
```

也没失效啊，继续试验发现，原来走索引更慢，INDEX FULL SCAN 的逻辑读为 649，远大于 TABLE ACCESS FULL 的 15，具体如下：

```
select /*+index(t)*/ count(*) from t;
执行计划
--------
| Id  | Operation         | Name          | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |     1 |   675   (1)| 00:00:09 |
|   1 |  SORT AGGREGATE   |               |     1 |            |          |
|   2 |   INDEX FULL SCAN | IDX_OBJECT_ID |   740 |   675   (1)| 00:00:09 |
-----------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        649  consistent gets
          0  physical reads
```

```
          0  redo size
        424  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-19　shrink 索引不失效也被弃用

结论，alter table t shrink 不会导致索引失效，但是索引块的高水平位无法释放。还是会产生大量的逻辑读。

5. 范围查询为何就用不到索引

这里说的是反向键索引的故事。

```
--以下语句能用到索引。
select * from t where id=28;
执行计划
--------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |    69 |   138K|   401   (0)| 00:00:05 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T         |    69 |   138K|   401   (0)| 00:00:05 |
|*  2 |   INDEX RANGE SCAN           | IDX_T_ID |   486 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

--不过奇怪的是，缘何下列语句却用不到索引，索引去哪儿了?

select * from t where id>=28 and id<=50;
执行计划
--------
Plan hash value: 1601196873
---------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |   304 |   608K|  1709   (1)| 00:00:21 |
|*  1 |  TABLE ACCESS FULL| T    |   304 |   608K|  1709   (1)| 00:00:21 |
-------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       6303  consistent gets
          0  physical reads
          0  redo size
       2263  bytes sent via SQL*Net to client
        426  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
```

```
          0  sorts (memory)
          0  sorts (disk)
         23  rows processed
```

这是为啥呢？索引肯定没有失效，否则 where id=28 也用不到索引。另外范围查询最终也不过返回 23 条记录，怎么就用不到索引了。接下来分析才发现：

```
select index_name,index_type from user_indexes where table_name='T';
INDEX_NAME                     INDEX_TYPE
------------------------------ -----------
IDX_T_ID                       NORMAL/REV
```

脚本 9-20　反向键索引与范围查询

原来这是将字节反转的反向索引，这个函数索引本身已经失去了范围的概念，当然不能用在范围查询中，不过等值查询还是没问题，反转查询再反转复原即可。

6. 回收站还原表后的索引故事

注意：关于误 drop 表后从回收站中取回表，需要记住一些细节，比如，这时其实该表的索引已经被重命名了。

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id);
set autotrace off
select index_name,status from user_indexes where table_name='T';

INDEX_NAME                     STATUS
------------------------------ --------
IDX_OBJECT_ID                  VALID

set autotrace traceonly
set linesize 1000
select /*+index(t idx_object_id)*/ * from t where object_id=19;

执行计划
--------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1 |   207 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T             |     1 |   207 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | IDX_OBJECT_ID |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
   2 - access("OBJECT_ID"=19)
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
```

```
          0  physical reads
          0  redo size
       1392  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

上述脚本没啥问题，接下来讲述一个真实的故事。某天某人不小心误 drop 了某表，就是这个 drop table t 动作，让此人惊出一身冷汗。最后他从回收站中用 flashback table t to before drop 的方式将该表取回。庆幸不已，不过后来却发现索引丢了。

```
select status from user_indexes where index_name='IDX_OBJECT_ID';
未选定行
--其实索引并没有丢，只是换了一个名字
select index_name,status from user_indexes where table_name='T';
INDEX_NAME                     STATUS
------------------------------ -------
BIN$9VPINfhiSnqqKgiI9qMgmw==$0 VALID
--另外可以重命名回去
alter index "BIN$Tqzx3kCNTryUkitDNSF9Mw==$0" rename to IDX_OBJECT_ID;
```

不过原先用 HINT 方式写上索引名的也不用担心，Oracle 还是会识别回来的，如下：

```
SQL> select /*+index(t idx_object_id)*/ * from t where object_id=19;
执行计划
--------
Plan hash value: 3597545767
---------------------------
| Id |Operation                   |Name                          |Rows|Bytes |Cost (%CPU)|
-----------------------------------------------------------------------------------------
|   0|SELECT STATEMENT            |                              |   1|  207 |    2   (0)|
|   1| TABLE ACCESS BY INDEX ROWID|T                             |   1|  207 |    2   (0)|
|*  2|  INDEX RANGE SCAN          |BIN$NyJOneU9S6K0fDw02M90Lg==$0|   1|      |    1   (0)|
-----------------------------------------------------------------------------------------
```

脚本 9-21　表 drop 再回收后索引的情况

7. 回收站恢复与约束的那点事

有一个系统由于不小心表被 drop 了，然后管理员快速从回收站中恢复了表，以为万事大吉就收工走了。后来才发现约束丢失了，由于约束丢失，导致了部分数据错乱。案例的脚本大致构造如下：

```
drop table t_p cascade constraints purge;
drop table t_c cascade constraints purge;

CREATE TABLE T_P (ID NUMBER, NAME VARCHAR2(30));
ALTER TABLE T_P ADD CONSTRAINT  T_P_ID_PK  PRIMARY KEY (ID);
CREATE TABLE T_C (ID NUMBER, FID NUMBER, NAME VARCHAR2(30));
```

```
ALTER TABLE T_C ADD CONSTRAINT FK_T_C FOREIGN KEY (FID) REFERENCES T_P (ID);
set autotrace off
INSERT INTO T_P SELECT ROWNUM, TABLE_NAME FROM ALL_TABLES;
INSERT INTO T_C SELECT ROWNUM, MOD(ROWNUM, 1000) + 1, OBJECT_NAME  FROM ALL_OBJECTS;
COMMIT;
CREATE INDEX IND_T_C_FID ON T_C (FID);
```

发现约束关系丢失了（这时候要考虑重建约束）。

```
Set linesize 1000
SELECT TABLE_NAME,
       CONSTRAINT_NAME,
       STATUS,
       CONSTRAINT_TYPE,
       R_CONSTRAINT_NAME
  FROM USER_CONSTRAINTS
 WHERE TABLE_NAME = 'T_C';
TABLE_NAME                     CONSTRAINT_NAME                STATUS   C R_CONSTRAINT_NAME
------------------------------ ------------------------------ -------- - ------------------
T_C                            FK_T_C                         ENABLED  R T_P_ID_PK
---删除表并完成回收站的恢复
DROP TABLE T_C ;
FLASHBACK TABLE T_C TO BEFORE  DROP;
--发现约束关系丢失了
SELECT TABLE_NAME,
       CONSTRAINT_NAME,
       STATUS,
       CONSTRAINT_TYPE,
       R_CONSTRAINT_NAME
  FROM USER_CONSTRAINTS
 WHERE TABLE_NAME = 'T_C';
未选定行
```

脚本 9-22　表 drop 再回收后约束丢失

8. 看看最典型的时间查询通病

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(created);
set autotrace traceonly
set linesize 1000

--以下写法在开发人员的代码中经常出现，是一个非常常见的通病。由于对列进行了运算，所以用不到索引，如下：
select * from t where trunc(created)>=TO_DATE('2013-12-14', 'YYYY-MM-DD')
and trunc(created)<=TO_DATE('2013-12-15', 'YYYY-MM-DD');

执行计划
--------
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    12 |  2484 |   296   (2)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| T    |    12 |  2484 |   296   (2)| 00:00:04 |
-------------------------------------------------------------------------------
   1 - filter(TRUNC(INTERNAL_FUNCTION("CREATED"))>=TO_DATE(' 2013-12-14
              00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              TRUNC(INTERNAL_FUNCTION("CREATED"))<=TO_DATE(' 2013-12-15 00:00:00',
              'syyyy-mm-dd hh24:mi:ss'))
统计信息
--------
          0  recursive calls
          0  db block gets
       1049  consistent gets
          0  physical reads
          0  redo size
       1390  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-23　时间查询用不上索引的低效写法

改为如下，立即就用到索引：

```
select * from t where created>=TO_DATE('2013-12-14', 'YYYY-MM-DD')
and created<TO_DATE('2013-12-15', 'YYYY-MM-DD')+1;

执行计划
--------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |   207 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T             |     1 |   207 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_OBJECT_ID |     1 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
   2 - access("CREATED">=TO_DATE(' 2013-12-14 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
             AND "CREATED"<TO_DATE(' 2013-12-16 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
统计信息
--------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
       1393  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

```
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-24　时间查询用上索引的高效写法

9. 请注意这写法是案例非笑话

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id);
VARIABLE id NUMBER;
EXECUTE :id := 8;
set linesize 1000
set autotrace traceonly
select * from t where object_id/2=:id;

执行计划
--------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |     1 |    36 |     9   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T_COL_TYPE  |     1 |    36 |     9   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
   1 - filter(TO_NUMBER("ID")=6)
统计信息
--------
          0  recursive calls
          0  db block gets
         32  consistent gets
          0  physical reads
          0  redo size
        540  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-25　对列进行运算导致索引无法使用

真见过有人写类似语句的，实际上只有如下写法才可以用到索引（因为列运算会用不到索引，除非是建函数索引）：

```
select * from t where object_id=:id*2;
执行计划
--------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |   685 |   138K|     6   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T             |   685 |   138K|     6   (0)| 00:00:01 |
```

```
|*  2 |   INDEX RANGE SCAN          | IDX_OBJECT_ID |   274 |       |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
       1407  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 9-26　消除对列进行运算后用到索引

所以，写法很重要，第一个写法不是笑话，是真有其事。

10. 组合升降序排序索引有玄机

```
drop table t purge;
create table t as select * from dba_objects where object_id is not null ;
set autotrace off
insert into t select * from t;
insert into t select * from t;
commit;
create index idx_t on t (owner,object_id);
alter table t modify owner not null;
alter table t modify object_id  not null;

set linesize 1000
set autotrace traceonly

--听说 order by 列有索引可以消除排序，测试发现，Oracle 选择不用索引，排序依然存在，索引去哪儿了?
select  * from t a order by owner desc ,object_type asc;
执行计划
-------------
| Id  | Operation           | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |   398K|    78M|       | 19133   (1)| 00:03:50 |
|   1 |  SORT ORDER BY      |      |   398K|    78M|    94M| 19133   (1)| 00:03:50 |
|   2 |   TABLE ACCESS FULL| T    |   398K|    78M|       |  1177   (1)| 00:00:15 |
-----------------------------------------------------------------------------------
统计信息
-------------
          0  recursive calls
          0  db block gets
       4209  consistent gets
```

```
         0  physical reads
         0  redo size
  13981752  bytes sent via SQL*Net to client
    215080  bytes received via SQL*Net from client
     19517  SQL*Net roundtrips to/from client
         1  sorts (memory)
         0  sorts (disk)
    292740  rows processed
```

换个思路，建如下索引(owner desc,object_type asc);，效率果然提高了，代价比未用索引导致排序的代价 19133 低，是 14687。

```
drop index idx_t;
create index idx_t on t(owner desc,object_type asc);

--哦，索引在这，效率果然提高了,代价比未用索引导致排序的代价 19133 低，是 14687。
select  * from t a order by owner desc ,object_type asc;
执行计划
----------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |   398K|    78M| 14687   (1)| 00:02:57 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T     |   398K|    78M| 14687   (1)| 00:02:57 |
|   2 |   INDEX FULL SCAN            | IDX_T |   398K|       |  1085   (1)| 00:00:14 |
----------------------------------------------------------------------------------------
统计信息
--------
         0  recursive calls
         0  db block gets
     52710  consistent gets
         0  physical reads
         0  redo size
  13821025  bytes sent via SQL*Net to client
    215080  bytes received via SQL*Net from client
     19517  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
    292740  rows processed
```

脚本 9-27　组合升降索引消除排序

9.3.3　索引的取舍

既然索引有好处也有坏处，那我们对待索引的建立就要特别谨慎，如果有些索引建立后 SQL 根本就用不上，那真是有害无益了！如何识别建了索引能用上，难道是建完再看吗？当然不是！我们来一起看看 Oracle 提供的一个有趣特性：虚拟索引。

以下脚本完成虚拟索引的创建：

```
drop table t purge;
create table t as select * from dba_objects;
--创建虚拟索引，首先要将_use_nosegment_indexes 的隐含参数设置为 true
alter session set "_use_nosegment_indexes"=true;
--虚拟索引的创建语法比较简单，实际上就是普通索引语法后面加一个 nosegment 关键字
create index ix_t_id on t(object_id) nosegment;
```

接下来用 explain plan for 的方式查看是否用到索引：

```
explain plan for select * from t where object_id=1;
select * from table(dbms_xplan.display());
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
Plan hash value: 206018885
--------------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |    19 |  3933 |     5   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T       |    19 |  3933 |     5   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IX_T_ID |   450 |       |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
   2 - access("OBJECT_ID"=1)
Note
-----
   - dynamic sampling used for this statement

已选择 18 行。
```

通过执行计划可以看出 Oracle 确实会用到这个索引，说明此时在 object_id 列建索引是有用的。不过这毕竟是虚拟索引，并没有真正建立，用 statistics_level=all 的方式我们就可以跟踪到 Oracle 真实执行计划，如下：

```
set autotrace off
--以下是真实执行计划，显然用不到索引。
alter session set statistics_level=all;
select * from t where object_id=1;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
SQL_ID  2qhwh0nzrzx2r, child number 1
-------------------------------------
select * from t where object_id=1
Plan hash value: 1601196873
--------------------------------------------------------------------------------------------
| Id  | Operation         | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
--------------------------------------------------------------------------------------------
|*  1 |  TABLE ACCESS FULL| T    |      1 |     19 |      0 |00:00:00.02 |    1723 |
--------------------------------------------------------------------------------------------
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("OBJECT_ID"=1)
Note
-----
   - dynamic sampling used for this statement
已选择 21 行。
```

脚本 9-28　用虚拟索引来判断索引的取舍

当然，从数据字典中是无法找到这个索引的。

```
SQL> select index_name,status from user_indexes where table_name='T';
未选定行
```

注意，虚拟索引的几个特点：

- 无法执行 alter index。
- 不能创建和虚拟索引同名的实际索引。
- 可以创建和虚拟索引包含相同列但不同名的实际索引。
- 在 Oracle 10g 中使用回收站特性的时候，必须显式 drop 虚拟索引，或者在 drop table 后再 purge table，才能创建同名的索引。
- 虚拟索引分析有效，但是数据字典里查不到结果，估计是 oracle 内部临时保存了分析结果。

9.4　本章习题、总结与延伸

习题 1：举例说明一下哪些场景会用不上索引（索引没失效）。

习题 2：举例说明一下哪些场景会导致索引失效或者丢失了。

习题 3：说说你对影响数据插入性能的认识。

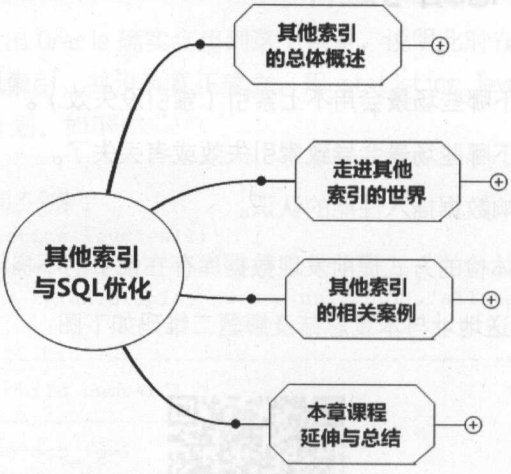
习题 4：你能否通过体检的方式提前发现数据库存在索引的不良问题?

习题及疑问的邮件发送地址与本章总结及解题二维码如下图：

第 10 章　且慢，其他索引应用让 SQL 飞

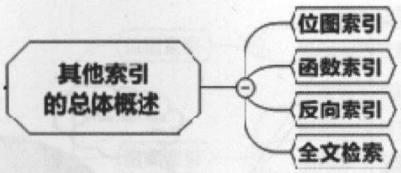
走进其他索引的精彩世界

前面一章谈到了索引的好与坏，不知道大家注意到没有，这些其实都是一些普通的 Btree 索引。其实还有一些比较特殊的索引，比如位图索引、函数索引、反向键索引和全文索引。它们的结构很特殊，应用的场景也比较特殊，不过如果我们能巧妙地将这类索引的特性和业务场景结合起来，在 SQL 优化上将起到意想不到的效果。本章还是先从其他索引的总体概述开始讲述，接着进一步通过各种试验了解特性，然后进入案例实战体会环节，最后总结。总体学习思路如下图所示：



10.1　其他索引的总体概述

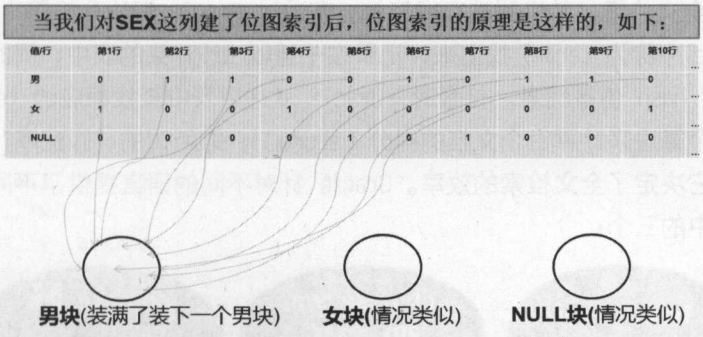
其他索引主要分成如下 4 个部分：



10.1.1　位图索引

简单地说说位图索引的结构，比如 T 表有 4 个字段，分别是 ID、NAME、SEX 和 STATUS，其中 SEX 取值仅为男或女，有时由于不知道性别，暂时为空，具体如下：

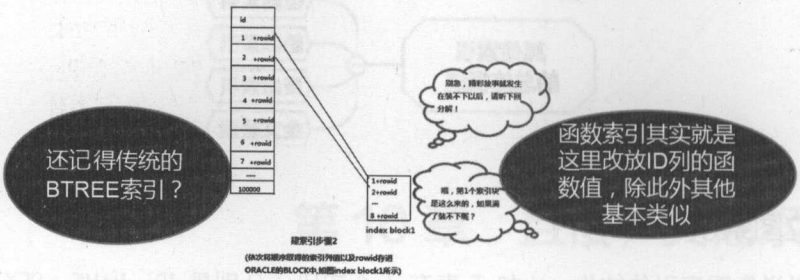
ID	NAME	SEX	STATUS
1	张一	女	毕业
2	李二	男	在读
3	王三	男	毕业
4	赵四	女	在读
5	马五		在读
6	刘六	男	毕业
7	孔七		毕业
8	叶八	男	在读
9	陈九	男	毕业
10	丁大	女	在读
……	……	……	……



简单地说，就是位图索引存储的是比特值。

10.1.2　函数索引

基于函数的索引是将一个函数计算得到的结果存储在行的列中，而不是存储列数据本身。可以把基于函数的索引看作一个虚拟列上的索引（这个列不是物理地存储在表中）。

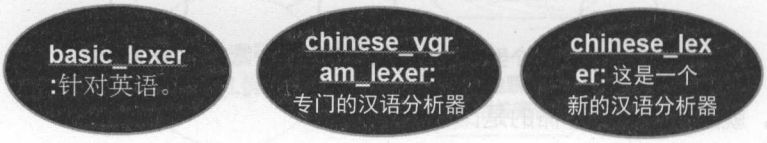


10.1.3 反向键索引

反向键索引就是普通的 B*TREE 索引，只不过键中的字节会“反转”。利用反向键索引，如果索引中填充的是递增的值，那么索引条目在索引中可以得到更均匀的分布。如 687002、687003、687004 等值是顺序的，如果是传统 B*TREE 索引，这些值就会在同一个右侧块上，加剧了块的竞争。如果是反向键索引，Oracle 会逻辑地将 687002、687003、00786 都转换为如下所示。一下子距离变得很远，于是索引的插入立即分布到多个块上去了。

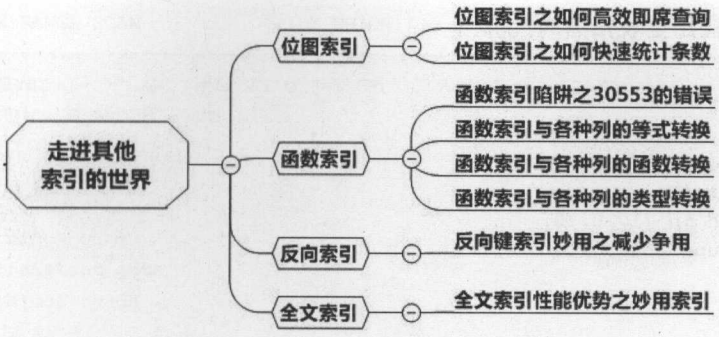
10.1.4 全文索引

Oracle 实现全文检索，其机制其实很简单。即通过 Oracle 词法分析器（lexer）将所有的表意单元（Oracle 中称为 term）找出来，记录在一组以 dr$开头的表中，同时记下该 term 出现的位置、次数、hash 值等信息。检索时，Oracle 从这组表中查找相应的 term，并计算其出现频率，根据某个算法来计算每个文档的得分（score），即所谓的“匹配率”。而 lexer 则是该机制的核心，它决定了全文检索的效率。Oracle 针对不同的语言提供了不同的 lexer，而我们通常能用到其中的三个：



10.2 走进其他索引的世界

说完前面的基本概念后，我们一起进入更具体的内部世界，深入地研究其他索引的特性，如下图所示：



10.2.1　位图索引

位图索引的结构比较特殊，其索引块存储的 0、1 的值非常适合与或非的运算，比如 0 和 1 的与就是 0，在即席查询的场景下性能比较高效。此外由于位图索引存储的是比特值而非索引列的取值，这时候用 Count(*) 统计条数会特别高效。首先我们来看即席查询的例子。

1. 位图索引之如何高效即席查询

做位图索引与即席查询试验前的准备：

```
drop table t purge;
set autotrace off
create table t
(name_id,
 gender not null,
 location not null,
 age_group not null,
 data
 )
 as
 select rownum,
        decode(ceil(dbms_random.value(0,2)),
               1,'M',
               2,'F')gender,
        ceil(dbms_random.value(1,50)) location,
        decode(ceil(dbms_random.value(0,3)),
               1,'child',
               2,'young',
               3,'middle_age',
               4,'old'),
         rpad('*',400,'*')
from dual
connect by rownum<=100000;

--注意，以下收集统计信息操作必须先执行。
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'T',estimate_percent
=> 10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;
```

脚本 10-1　位图索引试验环境准备

即席查询中应用**全表扫描**的代价：

```
set linesize 1000
set autotrace traceonly
select *
    from t
    where gender='M'
    and location in (1,10,30)
    and age_group='child';

执行计划
--------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   489 |   113K|  1674   (1)| 00:00:21 |
|*  1 |  TABLE ACCESS FULL| T    |   489 |   113K|  1674   (1)| 00:00:21 |
-------------------------------------------------------------------------------
   1 - filter("GENDER"='M' AND ("LOCATION"=1 OR "LOCATION"=10 OR
              "LOCATION"=30) AND "AGE_GROUP"='child')
统计信息
--------
          0  recursive calls
          0  db block gets
       6112  consistent gets
          0  physical reads
          0  redo size
      15885  bytes sent via SQL*Net to client
        943  bytes received via SQL*Net from client
         50  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        723  rows processed
```

脚本 10-2　即席查询中应用全表扫描的代价

以下是即席查询中，Oracle 选择**组合索引**情况的代价和逻辑读(注意，回表的代价特别大)：

```
drop index idx_union;
create index idx_union on t(gender,location,age_group);
select *
    from t
    where gender='M'
    and location in (1,10,30)
    and age_group='child';

普通联合索引执行计划
--------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |   810 |   180K|   793   (0)| 00:00:10 |
|   1 |  INLIST ITERATOR             |           |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| T         |   810 |   180K|   793   (0)| 00:00:10 |
```

```
|*  3 |    INDEX RANGE SCAN          | IDX_UNION |   810 |       |     4   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------

   3 - access("GENDER"='M' AND ("LOCATION"=1 OR "LOCATION"=10 OR "LOCATION"=30)
             AND "AGE_GROUP"='child')
统计信息
--------
          0  recursive calls
          0  db block gets
       1071  consistent gets
          0  physical reads
          0  redo size
     318987  bytes sent via SQL*Net to client
        943  bytes received via SQL*Net from client
         50  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        731  rows processed
```

脚本 10-3　即席查询中应用组合索引的代价

即席查询应用**位图索引**，性能有飞跃，Oracle 自己选择了使用位图索引：

```
create bitmap index gender_idx on t(gender);
create bitmap index location_idx on t(location);
create bitmap index age_group_idx on t(age_group);
select *
    from t
    where gender='M'
    and location in (1,10,30)
    and age_group='child';

位图索引执行计划
----------------
|Id |Operation                      |Name          |Rows  |Bytes |Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|  0|SELECT STATEMENT               |              |  810 |  180K|  236   (0)| 00:00:03|
|  1| TABLE ACCESS BY INDEX ROWID   |T             |  810 |  180K|  236   (0)| 00:00:03|
|  2|  BITMAP CONVERSION TO ROWIDS  |              |      |      |           |          |
|  3|   BITMAP AND                  |              |      |      |           |          |
|  4|    BITMAP OR                  |              |      |      |           |          |
|* 5|     BITMAP INDEX SINGLE VALUE|LOCATION_IDX  |      |      |           |          |
|* 6|     BITMAP INDEX SINGLE VALUE|LOCATION_IDX  |      |      |           |          |
|* 7|     BITMAP INDEX SINGLE VALUE|LOCATION_IDX  |      |      |           |          |
|* 8|    BITMAP INDEX SINGLE VALUE |AGE_GROUP_IDX|      |      |           |          |
|* 9|    BITMAP INDEX SINGLE VALUE |GENDER_IDX    |      |      |           |          |
-----------------------------------------------------------------------------------------

   5 - access("LOCATION"=1)
   6 - access("LOCATION"=10)
   7 - access("LOCATION"=30)
   8 - access("AGE_GROUP"='child')
   9 - access("GENDER"='M')
统计信息
```

```
--------
         0  recursive calls
         0  db block gets
       722  consistent gets
         0  physical reads
         0  redo size
    318987  bytes sent via SQL*Net to client
       943  bytes received via SQL*Net from client
        50  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
       731  rows processed
```

脚本 10-4　即席查询中应用位图索引的代价

逻辑读从 1071 降到 722，性能果然大幅度提升。

2. 位图索引之如何快速统计条数

接下来我们看看统计条数的优化，首先是环境准备，如下：

```
drop table t purge;
set autotrace off
create table t as select * from dba_objects;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
update t set object_id=rownum;
commit;
```

脚本 10-5　Count 性能试验的环境准备

场景 1，观察 COUNT(*)全表扫描的代价：

```
set autotrace on
set linesize 1000
select count(*) from t;
  COUNT(*)
----------
   4684992
执行计划
--------
Plan hash value: 2966233522
----------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 | 20420   (11)| 00:04:06 |
|   1 |  SORT AGGREGATE    |      |     1 |             |          |
|   2 |   TABLE ACCESS FULL| T    |   294M| 20420   (11)| 00:04:06 |
```

```
-------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
      66731  consistent gets
          0  physical reads
          0  redo size
        426  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 10-6　Count 应用全表扫描的代价

场景 2，观察 COUNT(*)用普通索引的代价：

```
create index idx_t_obj on t(object_id);
alter table T modify object_id not null;
set autotrace on
select count(*) from t;
  COUNT(*)
----------
  4684992
普通索引的执行计划
------------------
| Id  | Operation             | Name      | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |     1 |  3047   (2)| 00:00:37 |
|   1 |  SORT AGGREGATE       |           |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX_T_OBJ |  4620K|  3047   (2)| 00:00:37 |
-------------------------------------------------------------------------------
普通索引的统计信息
------------------
          0  recursive calls
          0  db block gets
      10998  consistent gets
          0  physical reads
          0  redo size
        426  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 10-7　Count 应用普通索引的代价

观察 COUNT(*)用位图索引的代价（注意，这里我们特意取了 status 这个重复度很高的列

做索引）：

```
create bitmap index idx_bitm_t_status on t(status);
select count(*) from t;

SQL> select count(*) from t;

  COUNT(*)
----------
  4684992

位图索引的执行计划
------------------
| Id  | Operation                     | Name              | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                   |     1 |   115   (0)| 00:00:02 |
|   1 |  SORT AGGREGATE               |                   |     1 |            |          |
|   2 |   BITMAP CONVERSION COUNT     |                   |  4620K|   115   (0)| 00:00:02 |
|   3 |    BITMAP INDEX FAST FULL SCAN| IDX_BITM_T_STATUS |       |            |          |
-----------------------------------------------------------------------------------------
位图索引的统计信息
------------------
          0  recursive calls
          0  db block gets
        125  consistent gets
          0  physical reads
          0  redo size
        426  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
13  rows processed
```

脚本 10-8　Count 应用位图索引的代价

10.2.2　函数索引

前面已经介绍了函数索引的原理，现在我们来看看两个经典的例子：一个是只对表的部分记录建索引；另一个是关于函数索引如何减少递归调用。

1. 函数索引妙用之部分记录建索引

首先看一个例子，普通索引的情况，如下：

```
drop table t purge;
set autotrace off
create table t (id int ,status varchar2(2));
--建立普通索引
```

```
create index id_normal on t(status);
insert into t select rownum ,'Y' from dual connect by rownum<=1000000;
insert into t select 1 ,'N' from dual;
commit;
analyze table t compute statistics for table for all indexes for all indexed columns;

set linesize 1000
set autotrace traceonly
select * from t where status='N';
执行计划
----------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |     1 |    10 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T         |     1 |    10 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | ID_NORMAL |     1 |       |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------------------
   2 - access("STATUS"='N')
统计信息
--------
          0  recursive calls
          0  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
        483  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

--看索引情况
set autotrace off
analyze index id_normal validate structure;
select name,btree_space,lf_rows,height from index_stats;
set autotrace off
analyze index id_normal validate structure;
select name,btree_space,lf_rows,height from index_stats;
NAME                           BTREE_SPACE    LF_ROWS     HEIGHT
------------------------------ ----------- ---------- ----------
ID_NORMAL                         22960352    1000001          3
```

脚本 10-9　未对部分数据建索引的普通索引方式

接下来看看建函数索引的情况，这是一个非常特殊的例子，由于绝大部分的记录都是 Y，只有极少记录是 N，所以我们想只对记录是 N 的情况建索引，可行吗？不信，请往下看：

```
drop index id_normal;
create index id_status on  t (Case when status= 'N' then 'N' end);
```

```
analyze table t compute statistics for table for all indexes for all indexed columns;
/*以下这个 select * from t where (case when status='N' then 'N' end)='N'

写法不能变,如果是 select * from t where status='N'将无效!笔者见过有些人设置了选择性索引，却这样调用的，结果根本起不到任何效果!
*/

set autotrace traceonly
select * from t where (case when status='N' then 'N' end)='N';
执行计划
--------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |     1 |    10 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T          |     1 |    10 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | ID_STATUS |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
   2 - access(CASE "STATUS" WHEN 'N' THEN 'N' END ='N')
统计信息
--------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
        479  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

--接着观察 id_status（即函数索引）索引的情况
set autotrace off
analyze index id_status validate structure;
select name,btree_space,lf_rows,height from index_stats;

NAME                           BTREE_SPACE    LF_ROWS     HEIGHT
------------------------------ ----------- ---------- ----------
ID_STATUS                            8000          1          1
```

脚本 10-10　对部分数据建索引的函数索引方式

这里看出，脚本 10-9 的逻辑读（consistent gets）为 5，索引叶子数（LF_ROWS）为 1000001，高度（HEIGHT）为 3，而脚本 10-10 的逻辑读为 2，索引叶子数为 1，高度为 1，性能差异明显。

2. 函数索引妙用之减少递归调用

首先构造自定义函数的环境，如下所示：

```
drop table t1 purge;
drop table t2 purge;

create table t1 (first_name varchar2(200),last_name varchar2(200),id number);
create table t2 as select * from dba_objects where rownum<=1000;
insert into t1 (first_name,last_name,id) select object_name,object_type,rownum from
dba_objects where rownum<=1000;
commit;

create or replace function get_obj_name(p_id t2.object_id%type) return
t2.object_name%type DETERMINISTIC is
v_name t2.object_name%type;
begin
select object_name
into v_name
from t2
where object_id=p_id;
return v_name;
end;
/
```

脚本 10-11　自定义函数的环境构造

接下来看看未建函数索引的情况下，函数调用的写法和对应的性能，如下所示：

```
set linesize 1000
set autotrace traceonly
select *   from t1 where get_obj_name(id)='TEST'  ;
执行计划
----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    10 |  2170 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    10 |  2170 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------
统计信息
--------
       1057  recursive calls
          0  db block gets
      16007  consistent gets
          0  physical reads
          0  redo size
        410  bytes sent via SQL*Net to client
        404  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

脚本 10-12　未建函数索引的函数调用性能

接下来建自定义函数 get_obj_name 的函数索引，如下所示：

```
create index idx_func_id on t1(get_obj_name(id));
select *   from t1 where get_obj_name(id)='TEST'  ;
执行计划
--------
Plan hash value: 4083325411
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    10 | 22190 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T1          |    10 | 22190 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_FUNC_ID |     4 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
        410  bytes sent via SQL*Net to client
        404  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

脚本 10-13　建函数索引后的性能情况

可以看出前后性能差异显著，未建函数索引时 recursive calls 为 1057，consistent gets 为 16007，而建了函数索引后，recursive calls 为 0，consistent gets 为 2，性能差异简直是天壤之别！

10.2.3　反向键索引

反向键索引能减少争用，道理非常简单。Oracle 的最小访问单位是 Block，那么值为 10023 和 10024 两条记录很容易在一起。而由于最新的数据又最容易被人查询，所以很容易这两个数字同时被查到，那如果反向一下值就变成了 32001 和 42001，就把它们隔离到很远的位置，自然就不在一个块中了，从而就能减少热块竞争。

10.2.4　全文索引

1. 全文索引性能优势之妙用索引

环境准备：

```
sqlplus ljb/ljb
drop table test purge;
create table test as select * from dba_objects;
update test set object_name='高兴' where rownum<=2;
create index idx_object_name on test(object_name);
set autotrace traceonly explain
select * from test where object_name like '%高兴%';
exit;
sqlplus "/ as sysdba"
grant ctxapp to ljb;
alter user ctxsys  account unlock;
alter user ctxsys identified  by ctxsys;
connect ctxsys/ctxsys;
grant execute on ctx_ddl to ljb;
connect ljb/ljb

--第一次执行无须注释掉头两条
Begin
ctx_ddl.drop_preference('club_lexer');
ctx_ddl.drop_preference('mywordlist');
ctx_ddl.create_preference('club_lexer','CHINESE_LEXER');
ctx_ddl.create_preference('mywordlist', 'BASIC_WORDLIST');
ctx_ddl.set_attribute('mywordlist','PREFIX_INDEX','TRUE');
ctx_ddl.set_attribute('mywordlist','PREFIX_MIN_LENGTH',1);
ctx_ddl.set_attribute('mywordlist','PREFIX_MAX_LENGTH', 5);
ctx_ddl.set_attribute('mywordlist','SUBSTRING_INDEX', 'YES');
end;
/

create index  id_cont_test on TEST (object_name) indextype is ctxsys.context
parameters (
'DATASTORE CTXSYS.DIRECT_DATASTORE FILTER
CTXSYS.NULL_FILTER LEXER club_lexer WORDLIST mywordlist');
```

脚本 10-14　全文检索的环境搭建

接下来执行普通的查询语句，果然用不到索引：

```
exec ctx_ddl.sync_index('id_cont_TEST', '20M');
set autotrace traceonly
set linesize 1000
select * from test where OBJECT_NAME like '%高兴%';
执行计划
--------
Plan hash value: 1357081020
----------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    34 |  7038 |   292   (1)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| TEST |    34 |  7038 |   292   (1)| 00:00:04 |
```

```
---------------------------------------------------------------------------
   1 - filter("OBJECT_NAME" IS NOT NULL AND "OBJECT_NAME" LIKE '%高兴%')
统计信息
--------
          0  recursive calls
          0  db block gets
       1049  consistent gets
          0  physical reads
          0  redo size
       1498  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

脚本 10-15　普通索引的模糊查找用不到索引

接下来用全文检索方法，果然用到索引：

```
select * from test where contains(OBJECT_NAME,'高兴')>0;
执行计划
--------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |    49 | 10731 |    14   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| TEST         |    49 | 10731 |    14   (0)| 00:00:01 |
|*  2 |   DOMAIN INDEX               | ID_CONT_TEST |       |       |     4   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
   2 - access("CTXSYS"."CONTAINS"("OBJECT_NAME",'高兴')>0)
统计信息
--------
         11  recursive calls
          0  db block gets
         21  consistent gets
          0  physical reads
          0  redo size
       1504  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

脚本 10-16　应用全文检索完成快速检索

脚本 10-15 的 consistent gets 为 1049，而脚本 10-16　为 21，性能差异巨大，这背后的原理是什么呢？请看下面解释：

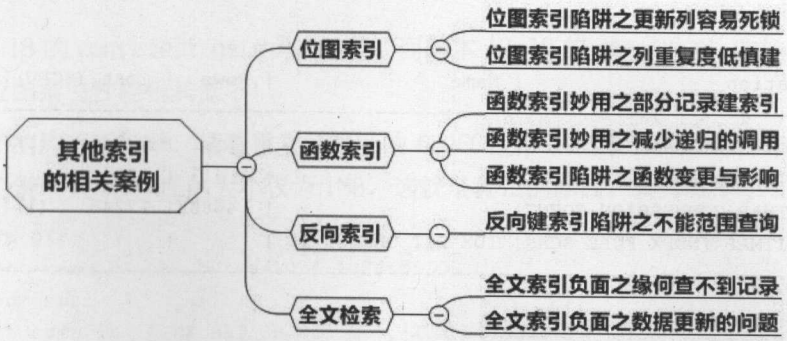
Oracle 实现全文检索，其机制其实很简单。即通过 Oracle 获得专利的词法分析器(lexer)，将文章中所有的表意单元（Oracle 称为 term，此处我理解为单词或者一些有意义的

词语）找出来，记录在一组以 dr$开头的表中，同时记下该 term 出现的位置、次数、hash 值等信息。检索时，Oracle 从这组表中查找相应的 term，并计算其出现频率，根据某个算法来计算每个文档的得分（score），即所谓的“匹配率”。而 lexer 则是该机制的核心，它决定了全文检索的效率。Oracle 针对不同的语言提供了不同的 lexer，而我们通常能用到其中的三个：

- basic_lexer。针对英语。它能根据空格和标点将英语单词从句子中分离，还能自动将一些出现频率过高且已经失去检索意义的单词作为“垃圾”处理，如 if 、is 等，具有较高的处理效率。但该 lexer 应用于汉语则有很多问题，由于它只认空格和标点，而汉语的一句话中通常不会有空格，因此，它会把整句话作为一个 term，这事实上失去了检索能力。以“中国人民站起来了”这句话为例，basic_lexer 分析的结果只有一个 term ,就是“中国人民站起来了”。此时若检索“中国”，将检索不到内容。
- chinese_vgram_lexer。专门的汉语分析器，支持所有汉字字符集。该分析器按字为单元来分析汉语句子。“中国人民站起来了”这句话，会被它分析成如下几个 term：“中”、“中国”、“国人”、“人民”、“民站”、“站起”、“起来”、“来了”、“了”。可以看出，这种分析方法，实现算法很简单，并且能实现“一网打尽”，但效率则是差强人意。
- chinese_lexer。这是一个新的汉语分析器，只支持 utf8 字符集。上面已经提到，chinese vgram lexer 这个分析器由于不认识常用的汉语词汇，因此分析的单元非常机械，像上面的“民站”、“站起”在汉语中根本不会单独出现，因此这种 term 是没有意义的，反而影响效率。chinese_lexer 的最大改进就是能认识大部分常用汉语词汇，因此能更有效率地分析句子，像以上两个愚蠢的单元将不会再出现，极大提高了效率。但是它只支持 utf8，如果你的数据库是 zhs16gbk 字符集，则只能使用 Chinese vgram lexer。

10.3　其他索引的相关案例

此次案例集中在其他索引的缺陷上，具体如下图所示：



10.3.1 位图索引

1. 位图索引陷阱之列重复度低慎建

有这么一个案例，有人在系统中误对重复度很低的字段建了位图索引，还强制在该字段上走索引，结果系统性能大幅度下降。我们构造一个环境来举例说明，首先是环境准备，如下：

```
--测试位图索引重复度前准备工作
drop table t purge;
set autotrace off
create table t as select * from dba_objects;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
update t set object_id=rownum;
commit;
```

脚本 10-17　位图索引案例的重复列环境构造

COUNT(*)在列重复度低时一般不会考虑使用位图索引：

```
create bitmap index idx_bit_object_id on t(object_id);
create bitmap index idx_bit_status on t(status);
--注意，以下收集统计信息的操作必须先执行。
exec dbms_stats.gather_table_stats(ownname => 'LJB',tabname => 'T',estimate_percent =>
10,method_opt=> 'for all indexed columns',cascade=>TRUE) ;

set linesize 1000
set autotrace traceonly

/*同学们可能还记得前面 count(*)性能大比拼中位图索引一马当先的事，现在我们看看，不在 status 列建位图索
引，在 object_id 列建位图索引后，是啥情况
*/
--create bitmap index idx_bit_object_id on t(object_id);
select /*+index(t,idx_bit_object_id)*/  count(*) from t;
执行计划
--------
| Id  | Operation                | Name               | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                    |     1 | 17245   (1)| 00:03:27 |
|   1 |  SORT AGGREGATE          |                    |     1 |            |          |
|   2 |   BITMAP CONVERSION COUNT|                    |  4688K| 17245   (1)| 00:03:27 |
|   3 |    BITMAP INDEX FULL SCAN| IDX_BIT_OBJECT_ID  |       |            |          |
-----------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
```

```
          0  db block gets
      16837  consistent gets
          0  physical reads
          0  redo size
        426  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 10-18　在重复度极低的列建位图索引

接下来看看查询运用针对 Status 列建函数索引的情况：

```
select /*+index(t,index idx_bit_status)*/  count(*) from t;
执行计划
--------
| Id  | Operation                     | Name           | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                |     1 |   105   (0)| 00:00:02 |
|   1 |  SORT AGGREGATE               |                |     1 |            |          |
|   2 |   BITMAP CONVERSION COUNT     |                |  4688K|   105   (0)| 00:00:02 |
|   3 |    BITMAP INDEX FAST FULL SCAN| IDX_BIT_STATUS |       |            |          |
-----------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        125  consistent gets
          0  physical reads
          0  redo size
        426  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 10-19　在重复度极高的列建位图索引

脚本 10-18 的 consistent gets 为 16837，而脚本 10-19 的 consistent gets 为 125，重复度低的列建位图索引性能极其低下！

性能低下的原因很简单，请看重复度很低的 OBJECT_ID 列的索引的大小，达到 136MB，而重复度很高的索引列 STATUS 的大小仅为 1MB，这就是真正的原因，如脚本所示：

```
set autotrace off
select segment_name,blocks,bytes/1024/1024 "SIZE(M)"
from user_segments
where segment_name in( 'IDX_BIT_OBJECT_ID','IDX_BIT_STATUS');
```

```
SEGMENT_NAME                   BLOCKS     SIZE(M)
------------------------------ ---------- ----------
IDX_BIT_OBJECT_ID               17408        136
IDX_BIT_STATUS                    128          1
```

脚本 10-20　在重复度低的列建位图索引性能低下原因

2. 位图索引陷阱之更新列容易死锁

前面描述了位图索引只适合用在重复度很高的列上，不过真的是只要重复度很高就可以吗？其实并非如此，如果该列会被频繁更新，也是不适合建位图索引的，因为更新会产生死锁。

比如某列是传输标志位，0 表示未处理，1 表示已处理。取值只有两个，可以说重复度非常高了，但是由于程序经常要进行更新，当处理完毕后会把 0 更新为 1。根据前面位图索引的原理特性我们不难知道，这个更新动作会导致该表死锁。

那什么列合适建位图索引呢？比如性别字段，一般来说性别只有男女两个，而且性别一般不容易被替换，这种列重复度高又极少更新的，比较适合建位图索引。

10.3.2　函数索引

1. 函数索引陷阱之 30553 错误

平时建的函数索引一般都是对 ORACLE 的自带函数做函数索引，如 upper()等，但是如果要基于自定义函数建索引，则必须使用 DETERMINISTIC 关键字，否则会报 ORA-30553 错误，这点要引起大家的注意。请看下面例子。

建完自定义函数后我们试着建立函数索引，发现建立失败：

```
sqlplus ljb/ljb
drop table test;
create table test as select * from user_objects ;
create or replace function f_minus1(i int)
   return int
   is
   begin
   return(i-1);
   end;
/
---建完函数后我们试着建立函数索引，发现建立失败
create index idx_ljb_test on test (f_minus1(object_id));
将会出现如下错误：
ORA-30553: 函数不能确定
```

加上 DETERMINISTIC 关键字后的自定义函数可以建立函数索引：

```
将函数加上 DETERMINISTIC 关键字重建
create or replace function f_minus1(i int)
   return int DETERMINISTIC
```

```
    is
    begin
    return(i-1);
   end;
/
--现在发现加上 DETERMINISTIC 关键字后的自定义函数可以建立函数索引了!
create index idx_test on test (f_minus1(object_id));
explain plan for select * from test where f_minus1(object_id)=23;
set linesize 1000
SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
------------------
Plan hash value: 2473784974
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |     8 |  1624 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| TEST     |     8 |  1624 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_TEST |     3 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
PLAN_TABLE_OUTPUT
-----------------
   2 - access("LJB"."F_MINUS1"("OBJECT_ID")=23)
Note
----
   - dynamic sampling used for this statement
已选择 18 行。
```

脚本 10-21　函数索引陷阱之 30553 错误

2. 函数索引与各种列的运算

测试函数索引前的准备：

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id);
create index idx_object_name on t(object_name);
create index idx_created on t(created);
```

比较 where object_id-10<=30 和 where object_id<=30+10 写法的性能，先看写法 1：

```
set autotrace traceonly
set linesize 1000
select * from t where object_id-10<=30;

执行计划
--------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT  |      |    12 |  2484 |   293   (1)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| T    |    12 |  2484 |   293   (1)| 00:00:04 |
---------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1051  consistent gets
          0  physical reads
          0  redo size
       2898  bytes sent via SQL*Net to client
        437  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         39  rows processed
```

脚本 10-22　对列进行运算的性能

写法 2，其实你应该这么写代码，才可以让 oracle 用上索引：

```
select * from t where object_id<=30+10;

已选择 39 行。
执行计划
--------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |    39 |  8073 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T             |    39 |  8073 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | IDX_OBJECT_ID |    39 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          9  consistent gets
          0  physical reads
          0  redo size
       4781  bytes sent via SQL*Net to client
        437  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         39  rows processed
```

脚本 10-23　改变写法，消除列运算后的性能

当然，你也可以这样建索引，create index idx_object_id_2 on t(object_id-10); 确实走索引了，建这样的索引，你真是够有勇气了!这虽然可以走索引，但是设计思路显然是不对

的。

```
select * from t where object_id-10<=30;
执行计划
--------
| Id  | Operation                   | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                 |  3873 |   832K|    14   (0)| 00:00:01|
|   1 |  TABLE ACCESS BY INDEX ROWID| T               |  3873 |   832K|    14   (0)| 00:00:01|
|*  2 |   INDEX RANGE SCAN          | IDX_OBJECT_ID_2 |   697 |       |     3   (0)| 00:00:01|
-------------------------------------------------------------------------------------------
统计信息
--------
          1  recursive calls
          0  db block gets
          9  consistent gets
          0  physical reads
          0  redo size
       2761  bytes sent via SQL*Net to client
        437  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         39  rows processed
```

脚本 10-24　列运算可用函数索引来优化，但是要慎用

3. 函数索引与各种列的函数转换

环境准备：

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id);
create index idx_object_name on t(object_name);
create index idx_created on t(created);
```

对列做 UPPER 操作，无法用到索引：

```
set autotrace traceonly
set linesize 1000
---以下语句由于列运算，所以走的是全表扫描
select * from t  where upper(object_name)='T' ;
执行计划
--------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    12 |  2484 |   293   (1)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| T    |    12 |  2484 |   293   (1)| 00:00:04 |
-----------------------------------------------------------------------------
统计信息
--------
```

```
          0  recursive calls
          0  db block gets
       1049  consistent gets
          0  physical reads
          0  redo size
       1500  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

脚本 10-25　对列做 UPPER 操作，无法用到索引

去掉列的 upper 操作后立即用索引：

```
select * from t  where  object_name='T' ;
执行计划
--------
Plan hash value: 1138138579
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |     2 |   414 |     4   (0)| 00:00:01|
|   1 |  TABLE ACCESS BY INDEX ROWID| T               |     2 |   414 |     4   (0)| 00:00:01|
|*  2 |   INDEX RANGE SCAN           | IDX_OBJECT_NAME |     2 |       |     3   (0)| 00:00:01|
-------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
       1506  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

脚本 10-26　去掉列 UPPER 操作，能用到索引

如果必须用 upper 的条件，那你想用到索引，就得去建函数索引：

```
create index idx_func_ojbnam on t(upper(object_name));
--继续执行，终于走索引了。
select * from t  where upper(object_name)='T' ;
执行计划
--------
| Id  | Operation                    | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT            |                 |   775 |   206K|   152   (0)| 00:00:02|
|   1 |  TABLE ACCESS BY INDEX ROWID| T               |   775 |   206K|   152   (0)| 00:00:02|
|*  2 |   INDEX RANGE SCAN          | IDX_FUNC_OJBNAM |   310 |       |     3   (0)| 00:00:01|
-------------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
       1500  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

脚本 10-27　建函数索引则能用到索引

4. 函数索引与各种列的类型转换

```
--举例说明:
drop table t_col_type purge;
create table t_col_type(id varchar2(20),col2 varchar2(20),col3 varchar2(20));
insert into t_col_type select rownum,'abc','efg' from dual connect by level<=10000;
commit;
create index idx_id on t_col_type(id);
set linesize 1000
set autotrace traceonly

select * from t_col_type where id=6;
执行计划
--------
| Id  | Operation         | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |            |     1 |    36 |     9   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T_COL_TYPE |     1 |    36 |     9   (0)| 00:00:01 |
-------------------------------------------------------------------------------
   1 - filter(TO_NUMBER("ID")=6)
统计信息
--------
          0  recursive calls
          0  db block gets
         32  consistent gets
          0  physical reads
          0  redo size
        540  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

```
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 10-28　列的类型转化导致用不到索引

实际上只有如下写法才可以用到索引，这个很不应该，是什么类型的取值就设置什么样的字段：

```
select * from t_col_type where id='6';
执行计划
--------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     1 |    36 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_COL_TYPE  |     1 |    36 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_ID      |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
   2 - access("ID"='6')
统计信息
--------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        544  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 10-29　消除列的类型转化能用到索引

再有就是函数索引的写法：

```
create index idx_func_tonumber_id on t_col_type(to_number(id));
select * from t_col_type where to_number(id)=6;

执行计划
--------
| Id  |Operation                    |Name                |Rows | Bytes |Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT             |                    |  100|  4900 |     2   (0)| 00:00:01 |
|   1 | TABLE ACCESS BY INDEX ROWID|T_COL_TYPE          |  100|  4900 |     2   (0)| 00:00:01 |
|*  2 |  INDEX RANGE SCAN           |IDX_FUNC_TONUMBER_ID|   40|       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
2 - access(TO_NUMBER("ID")=6)
统计信息
--------
```

```
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        540  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 10-30　列类型转换中函数索引的思路

这里无论是脚本 10-28、脚本 10-29 还是脚本 10-30 都不是最好的实现思路，最好的实现思路就是什么类型的字段就放什么类型的值。

结论：

什么类型就放什么值，比如存放字符的字段就设为 varchar2 类型，存放数值的字段就设置为 number 类型，存放日期的字段就设置为 date 类型，否则会发生类型转换，导致性能问题！

5. 函数索引陷阱之函数变更与影响

在应用函数索引时，如果要用到自定义函数，一定要注意，这里是有陷阱的。在自定义函数代码更新时，这个对应的函数索引也要重建，否则数据无法随着自定义函数代码的变化而变化。具体看下列构造的例子，首先是环境搭建，如下：

```
drop table t purge;
create table t ( x number, y varchar2(30));
set autotrace off
insert into t SELECT rownum, rownum||'a' FROM dual connect by rownum < 1000;
create or replace
package pkg_f is
function f(p_value varchar2) return varchar2 deterministic;
end;
/

create or replace
package body pkg_f is
function f(p_value varchar2) return varchar2
deterministic is
begin
return p_value;
end;
end;
/
```

```
create index idx_pkg_f_y on t ( pkg_f.f(y));
analyze table t compute statistics for table for all indexes for all indexed columns;
set autotrace on explain
SELECT * FROM t WHERE pkg_f.f(y)= '8a';
SQL> SELECT * FROM t WHERE pkg_f.f(y)= '8a';

         X Y
---------- ------------------------------
         8 8a
执行计划
--------
Plan hash value: 3110004532

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     1 |    12 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T           |     1 |    12 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_PKG_F_Y |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("PKG_F"."F"("Y")='8a')
```

脚本 10-31　函数索引陷阱试验的环境搭建

将包的代码修改后，我们惊奇地发现查询结果有误：

```
--将包的代码修改如下：
create or replace
package body pkg_f is
function f(p_value varchar2) return varchar2
deterministic is
begin
return p_value||'b';
end;
end;
/

惊奇地发现查询出错误的值：
SELECT * FROM t WHERE pkg_f.f(y)= '8a';

执行计划
--------
Plan hash value: 3110004532
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     1 |    12 |     2   (0)| 00:00:01 |
```

```
|   1 |  TABLE ACCESS BY INDEX ROWID| T           |     1 |    12 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | IDX_PKG_F_Y |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("PKG_F"."F"("Y")='8a')
```

脚本 10-32　函数代码更改后，查询数据有误

在重建索引后查询没有记录，这才是正确的结果：

```
SQL> drop index idx_pkg_f_y;
索引已删除。
SQL> create index idx_pkg_f_y on t ( pkg_f.f(y));
索引已创建。
SQL> SELECT * FROM t WHERE pkg_f.f(y)= '8a';
未选定行
执行计划
--------
Plan hash value: 3110004532
----------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    10 |   120 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T           |    10 |   120 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_PKG_F_Y |     4 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
13 - access("PKG_F"."F"("Y")='8a')
```

脚本 10-33　函数索引重建后数据查询正常

10.3.3　反向键索引

1. 反向键索引陷阱之不能范围查询

脚本环境准备：

```
drop table t purge;
create table t as select * from dba_objects;
update t set CREATED=sysdate-rownum ;
create index idx_rev_objn on t(object_name) reverse ;
create index idx_rev_created on t(created) reverse ;
```

反向键索引可以用在等值查询：

```
set autotrace traceonly
set linesize 1000
select * from t where created=sysdate-1;
执行计划
```

```
-----------------------------------------------------------------------------------------------
| Id| Operation                    | Name            | Rows | Bytes |Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT             |                 |    12|  2484 |  290   (0)| 00:00:04 |
|  1|  TABLE ACCESS BY INDEX ROWID| T               |    12|  2484 |  290   (0)| 00:00:04 |
|* 2|   INDEX RANGE SCAN           | IDX_REV_CREATED |   336|       |    1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
-
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
       1184  bytes sent via SQL*Net to client
        405  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

脚本 10-34　反向键索引可用于等值查询

但是反向键索引不能用在范围查询：

```
select * from t where created>=sysdate-10 and created<=sysdate-1;
执行计划
-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   153 | 31671 |   296   (2)| 00:00:04 |
|*  1 |  FILTER            |      |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| T    |   153 | 31671 |   296   (2)| 00:00:04 |
-----------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       1049  consistent gets
          0  physical reads
          0  redo size
       1763  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          9  rows processed
```

脚本 10-35　反向键索引不可用于范围查询

10.3.4　全文索引

1. 全文索引负面之谨防数据更新

全文检索一定要注意一点，数据更新要求 ctx_ddl.sync_index 实时同步，否则就会出现数据丢失的情况，具体如下：

```
create index  id_cont_test on TEST (object_name) indextype is ctxsys.context
parameters (
'DATASTORE CTXSYS.DIRECT_DATASTORE FILTER
CTXSYS.NULL_FILTER LEXER club_lexer WORDLIST mywordlist');

exec ctx_ddl.sync_index('id_cont_TEST', '20M');
set autotrace on explain
set linesize 1000

select count(*) from test where contains(OBJECT_NAME,'高兴')>0;
  COUNT(*)
----------
     2

执行计划
----------------------------------------------------------------------------------
| Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)| Time|
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |     1 |    78 |     4   (0)| 00:00:01|
|   1 |  SORT AGGREGATE   |               |     1 |    78 |            |         |
|*  2 |   DOMAIN INDEX    | ID_CONT_TEST  |    35 |  2730 |     4   (0)| 00:00:01|
----------------------------------------------------------------------------------

update test set object_name='高兴' where object_id>=3 and object_id<=5;
commit;

--发现由于又修改了 3 条记录，查询结果本应该由 2 条变更为 5 条记录，但是发现再查，依然是 2 条!
select count(*) from test where contains(OBJECT_NAME,'高兴')>0;
  COUNT(*)
----------
     2

---继续执行同步命令后
exec ctx_ddl.sync_index('id_cont_test', '20M');

---再次查询后，终于发现这下是 5 条记录了。
SQL> select count(*) from test where contains(OBJECT_NAME,'高兴')>0;
  COUNT(*)
----------
     5
```

脚本 10-36　数据更新与全文检索

10.4 本章习题、总结与延伸

位图索引分析				
原理	优势和陷阱		举例	文中章节
存储键值为 0 和 1 的比特值，占用空间极少	优势	高效即席查询	分析 consistent gets 的情况：全表扫描是 6112；组合索引是 1071；位图索引是 722	10.2.1
		快速统计条数	分析 consistent gets 的情况：全表扫描是 66731；索引快速全扫描是 10998；位图索引是 125	
	陷阱	列重复度低慎建	分析 consistent gets 的情况：对唯一性很高的列建位图索引是 16837；对高度重复的列建位图索引是 125	10.3.1
		更新列容易死锁	比如只有 0，1 两个取值，当 0 更新为 1 时，全表锁。如果有第 3 个取值比如 2，那记录为 2 的列没有锁住	

函数索引分析				
原理	优势和陷阱		举例	文中章节
存储 rowid 和列的函数值	优势	让索引变得短小精悍	create index id_status on t (Case when status= 'N' then 'N' end)；命令执行后。consistent gets 从 5 缩减为 3，索引的高度从 3 缩减为 1	10.2.2
		减少递归调用	create index idx_func_id on t1(get_obj_name(id))；命令执行后，recursive calls 从 1057 缩减到 0，consistent gets 从 16007 缩减到 2	
	陷阱	30553 的错误	函数索引无法创建，提示“ORA-30553：函数不能确定”错误，后续在函数中增加 DETERMINISTIC 关键字后解决	10.3.2
		各种列的运算形式	select * from t where object_id-10<=30；应该修正 select * from t where object_id<=30+10;的写法	
		各种列的函数转换	要想让 select * from t where upper(object_name)='T' ;走索引，必须要 create index idx_func_ojbnam on t(upper(object_name));建索引	
		各种列的类型转换	select * from t_col_type where id=6;如果这个 id 的字段是 varchar2 型，将用不到索引	
		函数变更与影响	函数及包中的代码改变后，对应的函数索引要重建，否则值会不对	

反向键索引分析				
原理	优势和陷阱		举例	文中章节
反转的键值加上 rowid	优势	减少热块竞争	Oracle 的最小访问单位是 Block，值为 10023 和 10024 的两条记录很容易在一起。如果反向一下值就变成 32001 和 42001，就隔离到很远的位置，这样它们就不在一个块中。自然就能减少热块竞争	10.2.3
	陷阱	不能范围查询	create index idx_rev_created on t(created) reverse ;后，如下语句无法走索引 select * from t where created>=sysdate-10 and created<=sysdate-1;	10.3.3

全文索引分析				
原理	优势和陷阱		举例	文中章节
通过 Oracle 词法分析器分析并记录在 dr$打头的表中	优势	能查询模糊匹配	select * from test where contains(OBJECT_NAME,'高兴')>0; 能用到索引，access("CTXSYS"."CONTAINS"("OBJECT_NAME",'高兴')>0)	10.2.4
	陷阱	缘何查不到记录	语句 1:select count(*) from test where contains(OBJECT_NAME,'高')>0; 语句 2: select count(*) from test where contains(OBJECT_NAME,'兴')>0; 语句 1 能搜索到而语句 2 搜索不到	10.3.4
		谨防数据更新	数据更新需要执行同步命令 exec ctx_ddl.sync_index('id_cont_test', '20M');否则更新的记录看不到	

请认真学习“其他索引”相关知识后完成如下习题。并将答题与相关疑问以邮件提交给笔者。

- 习题 1：说说位图索引的优势和不适合使用的场景。
- 习题 2：简要说说函数索引的原理及好处。
- 习题 3：说说使用反向键索引、全文索引的应用场景。
- 习题 4：如何通过监控的方式跟踪这些特殊索引？
  - 系统有哪些函数索引？
  - 系统有哪些位图索引？
  - 不该建位图索引的列。
  - 哪些 SQL 存在列运算？
  - 系统有无反向键索引？
  - 系统有哪些全文索引？

习题解说及本章总结的二维码如下图所示：

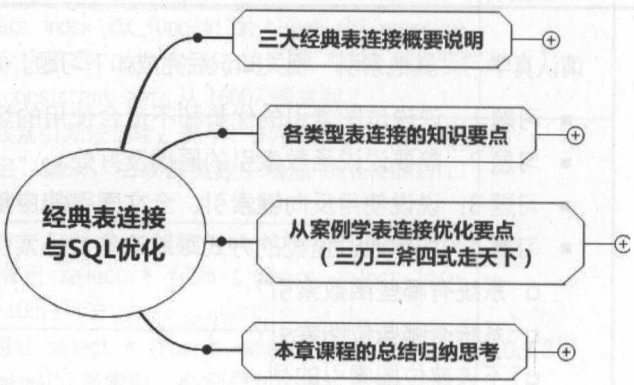
第 11 章　且慢，表连接的秘密让 SQL 飞

原来表连接原理如此实用

SQL 中最常见的就是多表关联的写法，这也是关系型数据库最大的优势之一。表连接的类型可以分成 Nested Loops Join、Hash Join、Merge Sort Join 三类。那我们选择哪一类会让 SQL 跑得更快呢？答案是：每个连接类型都有自己适用的场景，SQL 的执行计划会根据代价去判断该使用哪种表连接类型，不用我们去关心。

哇，这是什么答案，那本章我们还有必要学吗？

恩，很有必要。本章我们将了解这三大表连接类型及表连接的知识要点。还将通过研究具体的优化案例来加深印象，最后是思考回顾。本章总体学习思路如右图所示：



11.1　三大经典表连接概要说明

我来说一个故事吧。有两间房间，分别住着男孩和女孩。准备安排跳舞。

方法 1：选出男孩，比如小明去女孩房间寻找高度匹配的女孩，然后接着选小军去女孩房间选择匹配的女孩……

方法 2：男生排队，从高到矮。同时女生房间也排队，从高到矮。然后高对高矮对矮进行男女匹配，一起跳舞。

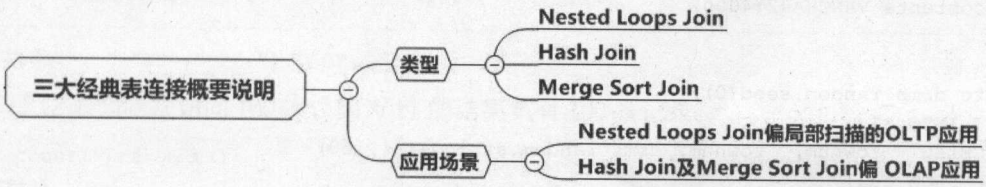
方法 3：女孩按班级在房间里面排成一列，从高到矮，不同班级排到不同的队伍。这时再让男孩根据他们所属的班级到各个队伍去找跟自己匹配的女孩。

到底哪一种模式更好呢？

有人认为第 2 种方法更好，也有人认为第 3 种方法更好，但是却几乎没有人认为第 1 种方法可行。

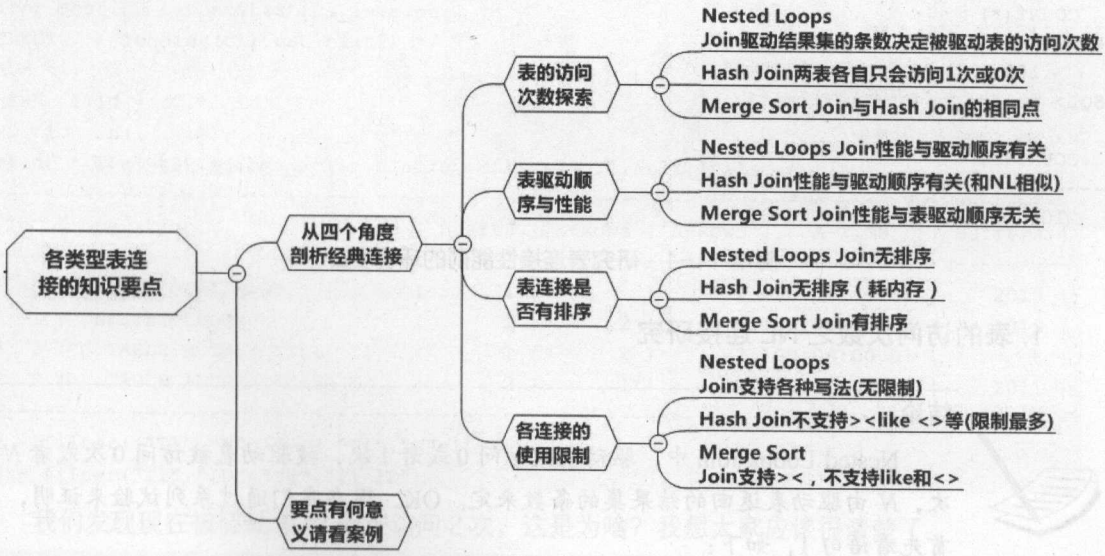
其实我要告诉大家的是，方法 1 就是 Nested Loops Join 连接，方法 2 是 Merge Sort Join 连接，方法 3 是 Hash Join 连接。现实中绝大部分的查询都是 NL 连接，其次是 Hash join，最后才是 Merge Sort Join。为什么？因为很多人以为这个故事中，所有男生都要参与跳舞，实际上我并没有这么说，可能小军选完就结束了，这时候无论用第 2 种还是第 3 种方式，最终为了选出 2 个人来，付出的代价显然更大。

而现实中我们在访问数据库时，一般都是查指定的用户的信息，比如自己查自己的话费等，这才是最广泛的应用。这就是典型的 OLTP，也就是第一种跳舞选择方式，NL 连接。



11.2　各类型表连接的知识要点

关于表连接的三大类型，我们从表的访问次数、表驱动顺序、是否排序、使用限制这 4 个维度来理解，如下图所示：



11.2.1 从表的访问次数探索

表访问次数在表连接的探讨中是非常重要的，从执行计划中是看 STARTS 对应的取值。下面我们展开一系列试验，首先是环境准备。

构造 t1、t2 表（注：本章所有的脚本都在这个构造的脚本基础上进行试验）：

```
DROP TABLE t1 CASCADE CONSTRAINTS PURGE;
DROP TABLE t2 CASCADE CONSTRAINTS PURGE;
CREATE TABLE t1 (
     id NUMBER NOT NULL,
     n NUMBER,
     contents VARCHAR2(4000)
   )
   ;
CREATE TABLE t2 (
     id NUMBER NOT NULL,
     t1_id NUMBER NOT NULL,
     n NUMBER,
     contents VARCHAR2(4000)
   )
   ;
execute dbms_random.seed(0);
INSERT INTO t1
     SELECT  rownum,  rownum, dbms_random.string('a', 50)
       FROM dual
     CONNECT BY level <= 100
      ORDER BY dbms_random.random;
INSERT INTO t2 SELECT rownum, rownum, rownum, dbms_random.string('b', 50) FROM dual
CONNECT BY level <= 100000
    ORDER BY dbms_random.random;
COMMIT;
SQL> select count(*) from t1;

  COUNT(*)
----------
     100
SQL> select count(*) from t2;

  COUNT(*)
----------
   100000
```

脚本 11-1 研究表连接性能前的环境准备

1. 表的访问次数之 NL 连接研究

结论：

Nested Loops Join 中，驱动表被访问 0 或者 1 次，被驱动表被访问 0 次或者 N 次，N 由驱动表返回的结果集的条数来定。OK，现在我们通过系列试验来证明，首先看语句 1，如下：

```
--我们用设置 statistics_level=all 的方式来观察如下表连接语句的执行计划：
--t2 表被访问 100 次(驱动表被访问 1 次，被驱动表被访问 100 次)
--这个 set linesize 1000 对 dbms_xplan.display_cursor 还是有影响的，如果没有设置，默认情况下的输出，将会少很多列，如 BUFFERS 等
Set linesize 1000
alter session set statistics_level=all ;
SELECT /*+ leading(t1) use_nl(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id;
--略去记录结果
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |    100 |00:00:00.94 |     100K|
|   1 |  NESTED LOOPS      |      |      1 |    100 |    100 |00:00:00.94 |     100K|
|   2 |   TABLE ACCESS FULL| T1   |      1 |    100 |    100 |00:00:00.01 |      14 |
|*  3 |   TABLE ACCESS FULL| T2   |    100 |      1 |    100 |00:00:00.94 |     100K|
-----------------------------------------------------------------------------------------
3 - filter("T1"."ID"="T2"."T1_ID")
```

解释 t2 表为啥被访问 100 次，因为 t1 的结果集有 100 条记录：

```
select count(*) from t1;
  COUNT(*)
-------------
    100
```

脚本 11-2　NL 研究，T2 表被访问 100 次及其原因

接下来继续试验，看语句 2，如下：

```
Set linesize 1000
alter session set statistics_level=all ;
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n in(17, 19);
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |      2 |00:00:00.02 |    2019 |
|   1 |  NESTED LOOPS      |      |      1 |      2 |      2 |00:00:00.02 |    2019 |
|*  2 |   TABLE ACCESS FULL| T1   |      1 |      2 |      2 |00:00:00.01 |       8 |
|*  3 |   TABLE ACCESS FULL| T2   |      2 |      1 |      2 |00:00:00.02 |    2011 |
-----------------------------------------------------------------------------------------
 2 - filter(("T1"."N"=17 OR "T1"."N"=19))
 3 - filter("T1"."ID"="T2"."T1_ID")
```

我们发现现在被驱动表 t2 表被访问 2 次，这是为啥？我想大家应该很清楚了：

```
---解释 t2 表为啥被访问 2 次
```

```
select count(*) from t1 where t1.n in (17,19);
  COUNT(*)
----------
         2
```

脚本 11-3　NL 研究，T2 表被访问 2 次及其原因

接下来，再做两个夸张点的试验，首先是语句 3，如下：

```
--接下来，t2 表居然被访问 0 次(驱动表被访问 1 次，被驱动表被访问 0 次)
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 999999999;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |      0 |00:00:00.01 |       7 |
|   1 |  NESTED LOOPS      |      |      1 |      1 |      0 |00:00:00.01 |       7 |
|*  2 |   TABLE ACCESS FULL| T1   |      1 |      1 |      0 |00:00:00.01 |       7 |
|*  3 |   TABLE ACCESS FULL| T2   |      0 |      1 |      0 |00:00:00.01 |       0 |
-------------------------------------------------------------------------------------------
2 - filter("T1"."N"=999999999)
3 - filter("T1"."ID"="T2"."T1_ID")
```

脚本 11-4　NL 研究，T2 表被访问 0 次

接下来看语句 4，如下：

```
---到最后，不只是 t2 表被访问 0 次，连 t1 表也访问 0 次
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND 1=2;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |      0 |00:00:00.01 |
|*  1 |  FILTER            |      |      1 |        |      0 |00:00:00.01 |
|   2 |   NESTED LOOPS     |      |      0 |    100 |      0 |00:00:00.01 |
|   3 |    TABLE ACCESS FULL| T1   |      0 |    100 |      0 |00:00:00.01 |
|*  4 |    TABLE ACCESS FULL| T2   |      0 |      1 |      0 |00:00:00.01 |
-----------------------------------------------------------------------------
1 - filter(NULL IS NOT NULL)
4 - filter("T1"."ID"="T2"."T1_ID")
```

脚本 11-5　NL 研究，T2 表和 T1 表皆无访问

大家惊奇地发现，语句 3 中 t2 表被访问 0 次，试验 2 更神奇，除了 t2 表被访问 0 次，t1 表也没被访问过，也是 0 次。其中语句 3 好理解，select count(*) from t1 where t1.n =

999999999 返回的是 0 条记录，当然 t2 表不被访问。而语句 4 则可以被理解为 AND 1=2 这个条件根本就不会成立，所以 t1 表根本无须访问，直接通过访问数据字典，获取到两表的结构就好了。

2. 表的访问次数之 HASH 连接研究

结论：

Hash Join 中，驱动表被访问 0 或者 1 次，被驱动表也是被访问 0 次或者 1 次，绝大部分场景下是驱动表和被驱动表各被访问 1 次。

关于 Hash Join 访问次数的问题，这里就不再过多讨论，请认真看下面一组试验，来分别证明这些结论：

```
--Hash Join 中 t2 表只会被访问 1 次或 0 次(驱动表被访问 1 次，被驱动表被访问 1 次)
set linesize 1000
SELECT /*+ leading(t1) use_hash(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------
|Id|Operation          |Name|Starts|E-Rows|A-Rows|   A-Time   |Buffers|OMem|1Mem|Used-Mem|
-----------------------------------------------------------------------------------------------
| 0|SELECT STATEMENT   |    |     1|      |  100 |00:00:00.07|   1019|    |    |        |
|*1| HASH JOIN         |    |     1|  100 |  100 |00:00:00.07|   1019|742K|742K|1178K (0|
| 2|  TABLE ACCESS FULL|T1  |     1|  100 |  100 |00:00:00.01|      7|    |    |        |
| 3|  TABLE ACCESS FULL|T2  |     1|  111K|  100K|00:00:00.02|   1012|    |    |        |
-----------------------------------------------------------------------------------------------
1 - access("T1"."ID"="T2"."T1_ID")


--Hash Join 中 t2 表被访问 0 次的情况
SELECT /*+ leading(t1) use_hash(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=999999999;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------
| Id|Operation          |Name|Starts|E-Rows|A-Rows|   A-Time   |Buffers|OMem|1Mem|Used-Mem|
-----------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT   |    |     1|      |     0|00:00:00.01 |      7|    |    |        |
|* 1| HASH JOIN         |    |     1|     1|     0|00:00:00.01 |      7|676K|676K|205K (0)|
|* 2|  TABLE ACCESS FULL| T1 |     1|     1|     0|00:00:00.01 |      7|    |    |        |
|  3|  TABLE ACCESS FULL| T2 |     0|  111K|     0|00:00:00.01 |      0|    |    |        |
-----------------------------------------------------------------------------------------------
1 - access("T1"."ID"="T2"."T1_ID")
2 - filter("T1"."N"=999999999)

--Hash Join 中 t1 和 t2 表都被访问 0 次的情况
```

```
SELECT /*+ leading(t1) use_hash(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and 1=2;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------------
| Id |Operation           |Name|Starts | E-Rows |A-Rows|   A-Time   |OMem |1Mem |Used-Mem|
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT    |    |     1 |        |     0|00:00:00.01|     |     |        |
|*  1| FILTER             |    |     1 |        |     0|00:00:00.01|     |     |        |
|*  2|  HASH JOIN         |    |     0 |    100 |     0|00:00:00.01| 732K| 732K|        |
|   3|   TABLE ACCESS FULL| T1 |     0 |    100 |     0|00:00:00.01|     |     |        |
|   4|   TABLE ACCESS FULL| T2 |     0 |    111K|     0|00:00:00.01|     |     |        |
-------------------------------------------------------------------------------------------
 1 - filter(NULL IS NOT NULL)
 2 - access("T1"."ID"="T2"."T1_ID")
```

脚本 11-6　表的访问次数之 HASH 连接研究

3. 表的访问次数之排序合并连接研究

关于 Merge Sort Join 的表访问次数，和 Hash Join 是一样的，具体的试验这里就不做了，请你根据随书脚本自行试验来证明。

11.2.2　表驱动顺序与性能

表的驱动顺序在表连接中对性能到底有没有影响呢？我们分别做试验研究。

1. 表驱动顺序与性能之 Nested Loops Join

首先是 NL 连接，一起试验如下：

```
set linesize 1000
alter session set statistics_level=all;
SELECT /*+ leading(t1) use_nl(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |      1 |00:00:00.01 |    1014 |
|   1 |  NESTED LOOPS      |      |      1 |      1 |      1 |00:00:00.01 |    1014 |
|*  2 |   TABLE ACCESS FULL| T1   |      1 |      1 |      1 |00:00:00.01 |       8 |
|*  3 |   TABLE ACCESS FULL| T2   |      1 |      1 |      1 |00:00:00.01 |    1006 |
-------------------------------------------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"="T2"."T1_ID")
```

脚本 11-7　Nested Loops Join 的 T1 表先访问的情况

```
--Nested Loops Join 的 t2 表先被访问的情况
alter session set statistics_level=all;
SELECT /*+ leading(t2) use_nl(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |      1 |00:00:01.46 |     701K|
|   1 |  NESTED LOOPS      |      |      1 |      1 |      1 |00:00:01.46 |     701K|
|   2 |   TABLE ACCESS FULL| T2   |      1 |    100K|    100K|00:00:00.02 |    1006 |
|*  3 |   TABLE ACCESS FULL| T1   |    100K|      1 |      1 |00:00:01.40 |     700K|
-------------------------------------------------------------------------------------------
   3 - filter(("T1"."N"=19 AND "T1"."ID"="T2"."T1_ID"))
```

脚本 11-8　Nested Loops Join 的 T2 表先访问的情况

通过观察发现，脚本 11-7 的 Buffers 为 1014，而脚本 11-8 的 Buffers 为 701，看来，在 Nested Loops Join 中，表的驱动顺序对性能有着巨大的影响。

2. 表驱动顺序与性能之 Hash Join

接下来研究 Hash 连接与表驱动顺序的情况，首先是 T1 表先驱动的情况，试验如下：

```
set linesize 1000
alter session set statistics_level=all;
SELECT /*+ leading(t1) use_hash(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------------------
| Id|Operation          |Name|Starts|E-Rows|A-Rows |   A-Time   |Buffers|OMem|1Mem|Used-Mem|
-------------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT   |    |    1 |      |     1 |00:00:00.06|  1013 |    |    |        |
|* 1| HASH JOIN         |    |    1 |     1|     1 |00:00:00.06|  1013 | 42K|742K|335K (0)|
|* 2|  TABLE ACCESS FULL|T1  |    1 |     1|     1 |00:00:00.01|     7 |    |    |        |
|  3|  TABLE ACCESS FULL|T2  |    1 |  100K|   100K|00:00:00.02|  1006 |    |    |        |
-------------------------------------------------------------------------------------------------
   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T1"."N"=19)
```

脚本 11-9　Hash Join 的 T1 表先访问情况

接下来看 T2 表先驱动的情况

```
---Hash Join 的 t2 表先被访问的情况
set linesize 1000
SELECT /*+ leading(t2) use_hash(t1)*/ *
```

```
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------------
| Id |Operation          |Name|Starts|E-Rows |A-Rows|   A-Time  |Buffers|OMem |1Mem |Used-Mem|
-----------------------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT   |    |    1 |       |    1 |00:00:00.11|  1013 |     |     |        |
|* 1 | HASH JOIN         |    |    1 |     1 |    1 |00:00:00.11|  1013 |9472K|1956K| 10M (0)|
|  2 |  TABLE ACCESS FULL| T2 |    1 |   100K|  100K|00:00:00.02|  1005 |     |     |        |
|* 3 |  TABLE ACCESS FULL| T1 |    1 |     1 |    1 |00:00:00.01|     8 |     |     |        |
-----------------------------------------------------------------------------------------------------
   1 - access("T1"."ID"="T2"."T1_ID")
   3 - filter("T1"."N"=19)
```

脚本 11-10　Hash Join 的 T2 表先访问情况

通过观察发现，脚本 11-7 的 OMem 开销为 3439KB，而脚本 11-8 为 9472KB，说明小的结果集驱动时，内存消耗要小得多。看来，在 Hash Join 中，表的驱动顺序对性能也是有影响的。

3. 表驱动顺序与性能之 Merge Sort Join

最后分析排序合并连接与驱动顺序的关系，首先是 T1 表先驱动的情况，试验如下：

```
set linesize 1000
alter session set statistics_level=all;
SELECT /*+ leading(t1) use_merge(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------------
| Id|Operation          |Name|Starts|E-Rows|A-Rows|   A-Time  |Buffers |OMem |1Mem |Used-Mem|
-----------------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT   |    |    1 |      |    1 |00:00:00.08|   1012 |     |     |        |
|  1| MERGE JOIN        |    |    1 |    1 |    1 |00:00:00.08|   1012 |     |     |        |
|  2|  SORT JOIN        |    |    1 |    1 |    1 |00:00:00.01|      7 |2048 |2048 |2048  (0)|
|* 3|   TABLE ACCESS FULL|T1  |    1 |    1 |    1 |00:00:00.01|      7 |     |     |        |
|* 4|  SORT JOIN        |    |    1 |  100K|    1 |00:00:00.08|   1005 |9266K|1184K|8236K (0)|
|  5|   TABLE ACCESS FULL|T2  |    1 |  100K|  100K|00:00:00.02|   1005 |     |     |        |
-----------------------------------------------------------------------------------------------------
   3 - filter("T1"."N"=19)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
```

脚本 11-11　Merge Sort Join 的 T1 表先访问情况

接下来看 T2 表先驱动的情况：

```
--Merge Sort Join 的 t2 表先被访问的情况
set linesize 1000
```

```
SELECT /*+ leading(t2) use_merge(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------
| Id|Operation           |Name|Starts|E-Rows|A-Rows|   A-Time   |Buffers|OMem |1Mem |Used-Mem |
-------------------------------------------------------------------------------
|  0|SELECT STATEMENT    |    |    1 |      |    1 |00:00:00.11|  1012 |     |     |         |
|  1| MERGE JOIN         |    |    1 |    1 |    1 |00:00:00.11|  1012 |     |     |         |
|  2|  SORT JOIN         |    |    1 |  100K|   20 |00:00:00.11|  1005 |9266K|1184K|8236K (0)|
|  3|   TABLE ACCESS FULL| T2 |    1 |  100K|  100K|00:00:00.02|  1005 |     |     |         |
|* 4|  SORT JOIN         |    |   20 |    1 |    1 |00:00:00.01|     7 |2048 |2048 |2048  (0)|
|* 5|   TABLE ACCESS FULL| T1 |    1 |    1 |    1 |00:00:00.01|     7 |     |     |         |
-------------------------------------------------------------------------------
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
   5 - filter("T1"."N"=19)
```

脚本 11-12　Merge Sort Join 的 T2 表先访问情况

通过观察发现，脚本 11-11 和脚本 11-12 的 OMem 开销是一样的，无非是 9266K 和 2048 的顺序颠倒一下而已，加起来是一样的，说明在 Merge Sort Join 中，表的驱动顺序对性也是毫无影响的。

11.2.3　表连接是否有排序

排序是影响数据库性能的非常重要的要素，我们探讨一下这三种连接方式中，哪些是有排序的。

1. 表连接是否有排序之 Nested Loops Join

我们先看看 NL 连接与排序，用 set autotrace on 的模式进行跟踪，查看统计信息的 sorts 模块即可得知有无排序，具体试验如下：

```
set linesize 1000
set autotrace traceonly
SELECT /*+ leading(t1) use_nl(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;

-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   123 |   276   (1)| 00:00:04 |
|   1 |  NESTED LOOPS      |      |     1 |   123 |   276   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
```

```
|*  3 |   TABLE ACCESS FULL| T2   |     1 |    66 |   273   (1)| 00:00:04 |
---------------------------------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"="T2"."T1_ID")

统计信息
--------
          0  recursive calls
          0  db block gets
       1014  consistent gets
          0  physical reads
          0  redo size
        880  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 11-13　表连接是否有排序之 Nested Loops Join

这里的 sorts (memory)和 sorts (disk)都是 0，说明没有排序。接下来我们看看 Hash 连接的情况。

2. 表连接是否有排序之 Hash Join

继续试验 Hash 连接的场景，用 use_hash 的 hint 来确保使用 Hash 连接，观察 sorts (memory)的取值，具体试验如下：

```
set linesize 1000
set autotrace traceonly
SELECT /*+ leading(t1) use_hash(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
--------------
| Id - | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   123 |   277   (1)| 00:00:04 |
|*  1 |  HASH JOIN         |      |     1 |   123 |   277   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |   100K|  6445K|   273   (1)| 00:00:04 |
-------------------------------------------------------------------------------
   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T1"."N"=19)
统计信息
--------
          0  recursive calls
          0  db block gets
       1013  consistent gets
```

```
          0  physical reads
          0  redo size
        880  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 11-14　表连接是否有排序之 Hash Join

试验结果发现 sorts (memory)的取值为 0，没有排序。

3. 表连接是否有排序之 Merge Sort Join

最后试验排序合并连接的场景，用 use_merge 的 hint 来确保使用排序合并连接，观察 sorts (memory)的取值，具体试验如下：

```
set linesize 1000
set autotrace traceonly
SELECT /*+ leading(t1) use_merge(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;

执行计划
--------
| Id  | Operation           | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     1 |   123 |       |  1852   (1)| 00:00:23 |
|   1 |  MERGE JOIN         |      |     1 |   123 |       |  1852   (1)| 00:00:23 |
|   2 |   SORT JOIN         |      |     1 |    57 |       |     4  (25)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL| T1   |     1 |    57 |       |     3   (0)| 00:00:01 |
|*  4 |   SORT JOIN         |      |  100K|  6445K|    15M|  1848   (1)| 00:00:23 |
|   5 |    TABLE ACCESS FULL| T2   |  100K|  6445K|       |   273   (1)| 00:00:04 |
-------------------------------------------------------------------------------------
   3 - filter("T1"."N"=19)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
统计信息
--------
          0  recursive calls
          0  db block gets
       1012  consistent gets
          0  physical reads
          0  redo size
        880  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
```

```
          0  sorts (disk)
          1  rows processed
```

脚本 11-15　表连接是否有排序之 Merge Sort Join

试验结果发现 sorts (memory)的取值为 2，说明有排序，而且是 2 次排序。

11.2.4　各连接的使用限制

1. 各连接的使用限制之 Hash Join

Hash Join 不支持连接条件是大于、小于、不等于和 like 的场景，因为此语句 1、语句 2 和语句 3 都不能按照 Hint 的方式走 Hash Join，请看下列试验结果。

语句 1（连接条件为“>”，结果无法根据 Hint 走 Hash Join）：

```
set linesize 1000
set autotrace traceonly explain
SELECT /*+ leading(t1) use_hash(t2)*/ *
  FROM t1, t2
  WHERE t1.id > t2.t1_id
  AND t1.n = 19;
------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    50 |  6150 |   276   (1)| 00:00:04 |
|   1 |  NESTED LOOPS       |      |    50 |  6150 |   276   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T2   |    50 |  3300 |   273   (1)| 00:00:04 |
------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID">"T2"."T1_ID")
```

语句 2（连接条件为“<”，结果无法根据 Hint 走 Hash Join）：

```
SELECT /*+ leading(t1) use_hash(t2)*/ *
  FROM t1, t2
  WHERE t1.id < t2.t1_id
  AND t1.n = 19;
-------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      | 99950 |    11M|   276   (1)| 00:00:04 |
|   1 |  NESTED LOOPS       |      | 99950 |    11M|   276   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T2   | 99950 |  6442K|   273   (1)| 00:00:04 |
------------------------------------------------------------------------------
Predicate Information (identified by operation id):
```

```
------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"<"T2"."T1_ID")
```

脚本 11-16　Hash Join 不支持大于或者小于的连接条件

语句 3（连接条件为“<>”，结果无法根据 Hint 走 Hash Join）：

```
SELECT /*+ leading(t1) use_hash(t2)*/ *
  FROM t1, t2
  WHERE t1.id <> t2.t1_id
  AND t1.n = 19;
--------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 99999 |    11M|   276   (1)| 00:00:04 |
|   1 |  NESTED LOOPS      |      | 99999 |    11M|   276   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T2   | 99999 |  6445K|   273   (1)| 00:00:04 |
-------------------------------------------------------------------------
Predicate Information (identified by operation id):
------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"<>"T2"."T1_ID")
```

脚本 11-17　Hash Join 不支持不等值连接条件

注：Hash Join 不支持 Like 的等值条件，因为 Hash Join 是一种经典的等值算法，所以类似 WHERE t1.id <> t2.t1_id 的试验就不再举例了。

2. 各连接的使用限制之排序合并连接

Merge Sort Join 支持连接条件是大于、小于的场景，但是不支持不等于和 like 的场景，因此语句 1 和语句 2 可以按照 Hint 的方式走 Hash 连接，而语句 3 和语句 4 都不能按照 Hint 的方式走 Hash 连接，请看下列试验结果。

语句 1（连接条件为“>”，结果是 Merge Sort Join 支持这个算法，可以根据 Hint 走 Merge Sort Join）：

```
set linesize 1000
set autotrace traceonly explain
SELECT /*+ leading(t1) use_merge(t2)*/ *
  FROM t1, t2
  WHERE t1.id > t2.t1_id
  AND t1.n = 19;
---------------
| Id  | Operation          | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    50 |  6150 |       |  1852   (1)| 00:00:23 |
|   1 |  MERGE JOIN        |      |    50 |  6150 |       |  1852   (1)| 00:00:23 |
```

```
|   2 |   SORT JOIN          |      |     1 |    57 |       |     4  (25)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL| T1   |     1 |    57 |       |     3   (0)| 00:00:01 |
|*  4 |   SORT JOIN          |      |   100K|  6445K|    15M|  1848   (1)| 00:00:23 |
|   5 |    TABLE ACCESS FULL| T2   |   100K|  6445K|       |   273   (1)| 00:00:04 |
-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("T1"."N"=19)
   4 - access(INTERNAL_FUNCTION("T1"."ID")>INTERNAL_FUNCTION("T2"."T1_ID"))
       filter(INTERNAL_FUNCTION("T1"."ID")>INTERNAL_FUNCTION("T2"."T1_ID"))
```

脚本 11-18　Merge Sort Join 支持大于或者小于的连接条件

语句 2（连接条件为“<”，结果是 Merge Sort Join 支持这个算法，可以根据 Hint 走 Merge Sort Join，代码略去）；语句 3（连接条件为“<>”，结果是 Merge Sort Join 不支持这个算法，无法根据 Hint 走 Merge Sort Join）：

```
SELECT /*+ leading(t1) use_merge(t2)*/ *
  FROM t1, t2
  WHERE t1.id <> t2.t1_id
  AND t1.n = 19;
--------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      | 99999 |    11M|   276   (1)| 00:00:04 |
|   1 |  NESTED LOOPS       |      | 99999 |    11M|   276   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T2   | 99999 |  6445K|   273   (1)| 00:00:04 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"<>"T2"."T1_ID")
```

脚本 11-19　Merge Sort Join 不支持不等于的连接条件

语句 4（连接条件为“like”，结果是 Merge Sort Join 不支持这个算法，无法根据 Hint 走 Merge Sort Join）：

```
SELECT /*+ leading(t1) use_merge(t2)*/ *
  FROM t1, t2
  WHERE t1.id like t2.t1_id
  AND t1.n = 19;
--------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |  5000 |   600K|   276   (1)| 00:00:04 |
|   1 |  NESTED LOOPS       |      |  5000 |   600K|   276   (1)| 00:00:04 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    57 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T2   |  5000 |   322K|   273   (1)| 00:00:04 |
-------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter(TO_CHAR("T1"."ID") LIKE TO_CHAR("T2"."T1_ID"))
```

脚本 11-20　Merge Sort Join 不支持 LIKE 的连接条件

3. 各连接的使用限制之 NL 连接

通过前面的试验可以看出，Hash Join 限制最多，不支持大于、小于、不等于和 LIKE 等连接条件。而 Merge Sort Join 不支持不等于和 LIKE 等连接条件，却支持大于、小于的连接条件。那 Nested Loops Join 是什么情况呢?

其实已经不需要做试验了，因为大家应该可以注意到，只要是前面不支持的情况，全部都走成了 Nested Loops Join，所以 Nested Loops Join 是没有限制的。

11.2.5　三大表连接的特性总结

三大表连接方式的各个特性

连接类型	应用场景	表访问次数	表驱动顺序与性能	是否排序	各连接的使用限制
Nested Loops Join	偏局部扫描的 OLTP 应用	驱动表被访问 0 或者 1 次，被驱动表被访问 0 次或者 *N* 次，*N* 由驱动表返回的结果集的条数来定	小结果集先驱动，性能更好	无排序	无限制
Hash Join	偏全扫描的 OLAP 应用	驱动表被访问 0 或者 1 次，被驱动表也是被访问 0 次或者 1 次	小结果集先驱动，性能更好	无排序，但耗内存	不支持连接条件是“>、<、<>、like”的连接方式
Merge Sort Join		同 Hash Join	驱动顺序和性能无关	有排序	不支持连接条件是“<>、like”的连接方式，支持“>、<”连接方式

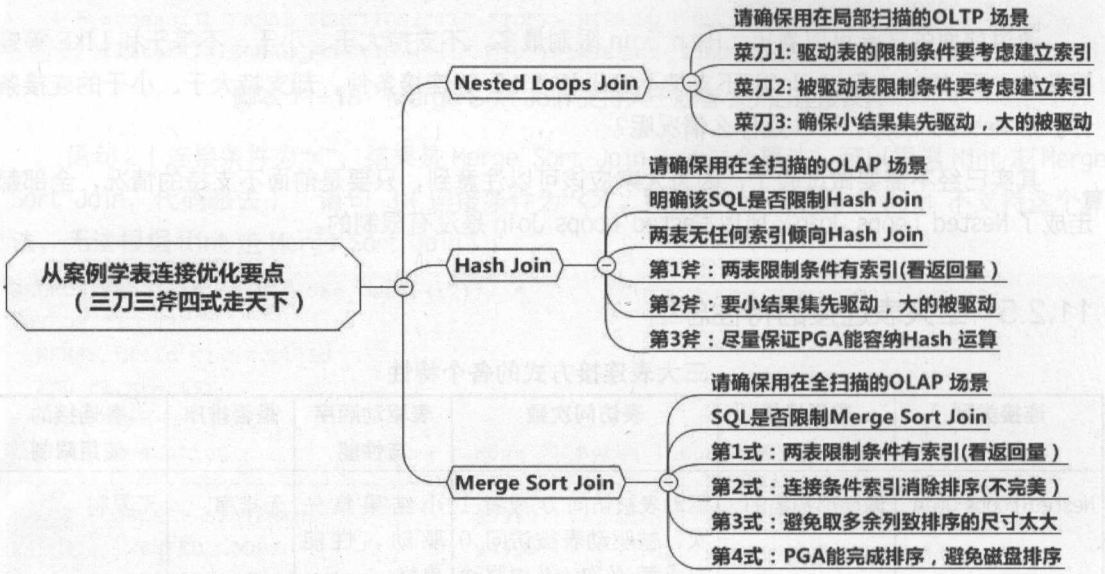
11.3　从案例学表连接优化要点（三刀三斧四式走天下）

细心的读者应该注意到，笔者在讲述每种表连接的时候都用了 hint，但是我们观察 11.2.2 节的例子可以发现，这个语句很显然是符合 NL 连接特性的，那为啥一定要加 hint，不是自然而然就能走 NL 连接吗? 因为驱动表才返回一条记录，被驱动表仅被访问一次，这显然是嵌套

循环连接的高效应用场景。

实际情况并不是如此，读者自行试验可以发现去掉 hint 后该语句走的是 Hash 连接，而非 NL 连接。

为啥呢？因为我们没有给它配置强大的武器，导致 NL 跑不快，那如何配置呢，这里我为三个表连接分别准备了三刀三斧四式，具体思路如下图所示：



11.3.1 一次 Nested Loops Join 的优化全过程

1. 请确保用在局部扫描的 OLTP 场景

说明一下，NL 连接是只适合用在 OLTP 场景的。通俗地说就是应用在有大量访问，且每个访问最终返回的记录很少的场景。

2. 菜刀 1：驱动表的限制条件要考虑建立索引

Nested Loops Join 两表无索引试验：

```
 set linesize 1000
set autotrace off
alter session set statistics_level=all ;
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT    |      |      1 |        |      1 |00:00:00.01 |    1014 |
|   1 |  NESTED LOOPS       |      |      1 |      1 |      1 |00:00:00.01 |    1014 |
|*  2 |   TABLE ACCESS FULL| T1   |      1 |      1 |      1 |00:00:00.01 |       8 |
|*  3 |   TABLE ACCESS FULL| T2   |      1 |      1 |      1 |00:00:00.01 |    1006 |
-------------------------------------------------------------------------------------------

   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"="T2"."T1_ID")
```

脚本 11-21　Nested Loops Join 两表无索引试验

第 1 把菜刀，对驱动表（t1 表）的限制条件建索引：

```
CREATE INDEX t1_n ON t1 (n);
---有了限制条件的索引，Nested Loops Join 性能略有提升
set linesize 1000
alter session set statistics_level=all ;
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------
|Id | Operation                    | Name|Starts|E-Rows|A-Rows|   A-Time   |Buffers|Reads |
-----------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT             |     |     1|      |     1|00:00:00.06|   1009|  1007|
|  1|  NESTED LOOPS                |     |     1|     1|     1|00:00:00.06|   1009|  1007|
|  2|   TABLE ACCESS BY INDEX ROWID| T1  |     1|     1|     1|00:00:00.01|      3|     6|
|* 3|    INDEX RANGE SCAN          | T1_N|     1|     1|     1|00:00:00.01|      2|     1|
|* 4|   TABLE ACCESS FULL          | T2  |     1|     1|     1|00:00:00.05|   1006|  1001|
-----------------------------------------------------------------------------------------------

   3 - access("T1"."N"=19)
   4 - filter("T1"."ID"="T2"."T1_ID")
```

脚本 11-22　Nested Loops Join，对驱动表限制条件建索引

3. 菜刀 2：被驱动表连接条件要考虑建立索引

继续给 Nested Loops Join 使用第 2 把菜刀（对被驱动表 t2 的连接条件建索引），如下：

```
CREATE INDEX t2_t1_id ON t2(t1_id);
----表连接性能有了大幅度提升
alter session set statistics_level=all ;
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------
| Id |Operation                    |Name    |Starts|E-Rows|A-Rows|   A-Time   |Buffers|Reads|
-----------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT             |        |     1|      |     1|00:00:00.01|      7|    4|
```

```
|   1| NESTED LOOPS                  |         |     1|      |     1|00:00:00.01|      7|   4 |
|   2|  NESTED LOOPS                 |         |     1|     1|     1|00:00:00.01|      6|   4 |
|   3|   TABLE ACCESS BY INDEX ROWID|T1       |     1|     1|     1|00:00:00.01|      3|   0 |
|*  4|    INDEX RANGE SCAN           |T1_N     |     1|     1|     1|00:00:00.01|      2|   0 |
|*  5|   INDEX RANGE SCAN            |T2_T1_ID|     1|     1|     1|00:00:00.01|      3|   4 |
|   6|  TABLE ACCESS BY INDEX ROWID |T2       |     1|     1|     1|00:00:00.01|      1|   0 |
-----------------------------------------------------------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

脚本 11-23　Nested Loops Join，对被驱动表连接条件建索引

性能有了大幅度提升，天啊，BUFFERS 居然只有 7！

增加了索引后 Oracle 不用 HINT，终于自己去选择 Nested Loops Join 了：

```
alter session set statistics_level=all ;
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------
|Id | Operation                     |Name     |Starts|E-Rows |A-Rows|   A-Time   |Buffers|
-----------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT              |         |     1|       |     1|00:00:00.01 |      7|
|  1|  NESTED LOOPS                 |         |     1|       |     1|00:00:00.01 |      7|
|  2|   NESTED LOOPS                |         |     1|     1 |     1|00:00:00.01 |      6|
|  3|    TABLE ACCESS BY INDEX ROWID|T1       |     1|     1 |     1|00:00:00.01 |      3|
|* 4|     INDEX RANGE SCAN          |T1_N     |     1|     1 |     1|00:00:00.01 |      2|
|* 5|    INDEX RANGE SCAN           |T2_T1_ID|     1|     1 |     1|00:00:00.01 |      3|
|  6|   TABLE ACCESS BY INDEX ROWID |T2       |     1|     1 |     1|00:00:00.01 |      1|
-----------------------------------------------------------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

脚本 11-24　条件完善自动走 Nested Loops Join

4. 菜刀 3：确保小结果集先驱动，大的被驱动

这部分内容在 11.2.2 节中描述过，这里就略去不做试验了。

11.3.2　一次 Hash Join 的优化全过程

1. 请确保用在全扫描的 OLAP 场景

一般来说 Hash Join 连接更适合用在 OLAP 场景。通俗地说就是最终返回的记录比较多。

2. 明确该 SQL 是否限制 Hash Join

说明 Hash 连接有很多限制，比如连接条件必须是等值，像>、<、<>、like 等都不可作为

连接条件。

3. 两表无任何索引倾向 Hash Join

有时候为了考虑平衡，表中某些列不能建索引，在两表无任何索引的时候，一般 SQL 会倾向于使用 Hash 连接，这点还请大家注意一下。

4. 第 1 斧：两表限制条件有索引(看返回量)

首先测试 Hash Join 两表的限制条件皆无索引的情况：

```
alter session set statistics_level=all ;
set linesize 1000
SELECT /*+ leading(t2) use_hash(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19
and t2.n=12;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------
|Id |peration          | Name|Starts|E-Rows|A-Rows|   A-Time  |Buffers|
-------------------------------------------------------------------------
|  0|ELECT STATEMENT   |     |     1|      |     0|00:00:00.01|   1104|
|* 1|HASH JOIN         |     |     1|     1|     0|00:00:00.01|   1104|
|* 2| TABLE ACCESS FULL| T2  |     1|    11|     1|00:00:00.01|   1005|
|* 3| TABLE ACCESS FULL| T1  |     1|     1|     1|00:00:00.01|     99|
-------------------------------------------------------------------------
   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T2"."N"=12)
   3 - filter("T1"."N"=19)
```

脚本 11-25　Hash Join 两表限制条件皆无索引

接下来试验，在两表的限制条件建索引后，发现索引都可以用到，性能大幅度提升。首先是对 t1 表的限制条件建索引的情况，测试发现性能果然有提升：

```
create index idx_t1_n on t1(n);
SELECT /*+ leading(t2) use_hash(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19
and t2.n=12;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------
|Id |Operation                    |Name    |Starts|E-Rows|A-Rows|   A-Time  |Buffers|
-----------------------------------------------------------------------------------
|  0|SELECT STATEMENT             |        |     1|      |     0|00:00:00.01|   1008|
|* 1| HASH JOIN                   |        |     1|     1|     0|00:00:00.01|   1008|
|* 2|  TABLE ACCESS FULL          |T2      |     1|    11|     1|00:00:00.01|   1005|
|  3|  TABLE ACCESS BY INDEX ROWID|T1      |     1|     1|     1|00:00:00.01|      3|
```

```
|* 4|   INDEX RANGE SCAN          |IDX_T1_N|      1|      1|      1|00:00:00.01|       2|
-------------------------------------------------------------------------------------------
   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T2"."N"=12)
```

脚本 11-26　Hash Join 驱动表限制条件有索引

接下来对 t2 表的限制条件再建索引，又更快了！

```
create index idx_t2_n on t2(n);

SELECT /*+ leading(t2) use_hash(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19
and t2.n=12;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------------
|Id |Operation                    |Name    |Starts|E-Rows|A-Rows|   A-Time  |Buffers |
-------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT             |        |     1|      |     0|00:00:00.01|       6 |
|* 1| HASH JOIN                   |        |     1|     1|     0|00:00:00.01|       6 |
|  2|  TABLE ACCESS BY INDEX ROWID|T2      |     1|     1|     1|00:00:00.01|       3 |
|* 3|   INDEX RANGE SCAN          |IDX_T2_N|     1|     1|     1|00:00:00.01|       2 |
|  4|  TABLE ACCESS BY INDEX ROWID|T1      |     1|     1|     1|00:00:00.01|       3 |
|* 5|   INDEX RANGE SCAN          |IDX_T1_N|     1|     1|     1|00:00:00.01|       2 |
-------------------------------------------------------------------------------------------
   1 - access("T1"."ID"="T2"."T1_ID")
   3 - access("T2"."N"=12)
   5 - access("T1"."N"=19)
```

脚本 11-27　Hash Join 被驱动表限制条件有索引

5. 第 2 斧：要小结果集先驱动，大的被驱动

这个在 12.2.2 节中描述并试验过，这里就不再重复做试验了，不过为了让读者加深印象，可以用令 set_table_stats 欺骗 Oracle 的方式进行表顺序的颠倒。

在无索引且是全扫描的情况下，一般走 HASH 连接，看下面性能：

```
set linesize 1000
alter session set statistics_level=all;
SELECT  *
FROM t1, t2
WHERE t1.id = t2.t1_id;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------------
|Id|Operation          |Name|Starts|E-Rows|A-Rows|   A-Time  |Buffers|OMem|1Mem|Used-Mem |
-------------------------------------------------------------------------------------------
| 0|SELECT STATEMENT   |    |     1 |      |  100 |00:00:00.07|   1019|    |    |         |
|*1| HASH JOIN         |    |     1 |   100|  100 |00:00:00.07|   1019|742K|742K|1202K (0)|
```

```
| 2|  TABLE ACCESS FULL|T1  |    1 |   100|  100 |00:00:00.01|      7|     |     |          |
| 3|  TABLE ACCESS FULL|T2  |    1 | 89127|  100K|00:00:00.02|   1012|     |     |          |
-----------------------------------------------------------------------------------------------

   1 - access("T1"."ID"="T2"."T1_ID")
```

脚本 11-28　无索引且全扫描，一般走 Hash Join

以下是经常出现的案例，由于统计信息的错误而导致执行计划的错误，我们用 set_table_stats 构造 t1 表是小表而 t2 表是大表，如下：

```
EXEC  dbms_stats.set_table_stats(user, 'T1', numrows => 20000000  ,numblks => 1000000);
EXEC  dbms_stats.set_table_stats(user, 'T2', numrows => 1  ,numblks => 1);
```

继续做试验如下，发现表驱动的顺序变了，结果内存的使用激增，如下：

```
set linesize 1000
alter session set statistics_level=all;
SELECT  *
FROM t1, t2
WHERE t1.id = t2.t1_id;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------
| Id |Operation          |Name|Starts|E-Rows|A-Rows|   A-Time   |Buffers|OMem |1Mem |Used-Mem|
-----------------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT   |    |    1 |      |  100 |00:00:00.10|  1019 |     |     |        |
|* 1 | HASH JOIN         |    |    1 |   20M|  100 |00:00:00.10|  1019 |9472K|1956K|  9M (0)|
|  2 |  TABLE ACCESS FULL| T2 |    1 |    1 |  100K|00:00:00.02|  1005 |     |     |        |
|  3 |  TABLE ACCESS FULL| T1 |    1 |   20M|  100 |00:00:00.01|    14 |     |     |        |
-----------------------------------------------------------------------------------------------

   1 - access("T1"."ID"="T2"."T1_ID")
```

脚本 11-29　感受 set_table_stats 骗过 Oracle 的方式

6. 第 3 斧：尽量保证 PGA 能容纳 Hash 运算

这类场景一般是 hash 连接占用 HASH AREA 内存区过多，这时候我们可以考虑增大 PGA。如果是 Oracle 11g，则默认是直接增大 memory_target，也可以选择手工管理。

Hash Join 算法：

step 1　判定小表是否能够全部存放在 hash area 内存中，如果可以，则做内存 hash join。如果不行，转第二步。

step 2　决定 fan-out 数。(Number of Partitions) * C<= Favm *M，其中 C 为 Cluster size，其值为 DB_BLOCK_SIZE*HASH_MULTIBLOCK_IO_COUNT；Favm 为 hash area 内存可以使用的百分比，一般为 0.8 左右；M 为 Hash_area_size 的大小。

step 3　读取部分小表 S，采用内部 hash 函数(这里称为 hash_fun_1)，将连接键值映射至某个分区，同时采用 hash_fun_2 函数对连接键值产生另外一个 hash 值，这个 hash 值用于创建 hash table 用，并且与连接键值存放在一起。

step 4　对 build input 建立位图向量。

step 5 如果内存中没有空间了，则将分区写至磁盘上。

step 6 读取小表 S 的剩余部分，重复第三步，直至小表 S 全部读完。

step 7 将分区按大小排序，选取几个分区建立 hash table(这里选取分区的原则是使选取的数量最多)。

step 8 根据前面用 hash_fun_2 函数计算好的 hash 值，建立 hash table。

step 9 读取表 B，采用位图向量进行位图向量过滤。

step 10 对经过过滤的数据采用 hash_fun_1 函数将数据映射到相应的分区中去，并计算 hash_fun_2 的 hash 值。

step 11 如果所落的分区在内存中，则将前面通过 hash_fun_2 函数计算所得的 hash 值与内存中已存在的 hash table 做连接，将结果写到磁盘上。如果所落的分区不在内存中，则将相应的值与表 S 相应的分区放在一起。

step 12 继续读取表 B，重复第 9 步，直至表 B 读取完毕。

step 13 读取相应的(Si,Bi)做 hash 连接。在这里会发生动态角色互换。

step 14 如果分过区后，最小的分区也比内存大，则发生 nested- loop hash join。

11.3.3 一次 Merge Sort Join 的优化全过程

1. 请确保用在全扫描的 OLAP 场景

一般来说排序合并连接倾向于吞吐量比较大的操作，也就是更适合用在 OLAP 场景。通俗地说就是最终返回的记录比较多。

2. SQL 是否限制 Merge Sort Join

排序合并连接也有很多限制，除了>、<可以外，<>和 like 皆不可用，都不可作为连接条件，所以也需要提前注意要优化的 SQL 语句是什么样的。

3. 第 1 式：两表限制条件有索引(看返回量)

首先是，两表限制条件皆无索引的情况，如下：

```
alter session set statistics_level=all ;
set linesize 1000
SELECT /*+ leading(t2) use_merge(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19
and t2.n=12;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------
| Id|Operation          |Name |Starts|E-Rows|A-Rows |   A-Time   |Buffers |OMem |1Mem|Used-Mem |
```

```
-----------------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT     |      |      1|       |      0 |00:00:00.01 |   1104 |      |     |          |
|  1| MERGE JOIN          |      |      1|      1|      0 |00:00:00.01 |   1104 |      |     |          |
|  2|  SORT JOIN          |      |      1|     11|      1 |00:00:00.01 |   1005 |2048  |2048|2048   (0)|
|* 3|   TABLE ACCESS FULL| T2   |      1|     11|      1 |00:00:00.01 |   1005 |      |     |          |
|* 4|  SORT JOIN          |      |      1|      1|      0 |00:00:00.01 |     99 |2048  |2048|2048   (0)|
|* 5|   TABLE ACCESS FULL| T1   |      1|      1|      1 |00:00:00.01 |     99 |      |     |          |
-----------------------------------------------------------------------------------------------------
   3 - filter("T2"."N"=12)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
   5 - filter("T1"."N"=19)
```

脚本 11-30　Merge Sort Join 两表限制条件皆无索引

在对两表的限制条件建索引后，发现索引都可以用到，性能大幅度提升。首先对 t1 表的限制条件建索引，发现如下 Merge Sort Join 快了：

```
create index idx_t1_n on t1(n);
SELECT /*+ leading(t2) use_merge(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19
and t2.n=12;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------
| Id|Operation                      |Name     |Starts|E-Rows|A-Rows|   A-Time   |Buffers|OMem|1Mem|Used-Mem |
-----------------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT               |         |     1|      |     0|00:00:00.01|   1008|    |    |          |
|  1| MERGE JOIN                    |         |     1|    1 |     0|00:00:00.01|   1008|    |    |          |
|  2|  SORT JOIN                    |         |     1|   11 |     1|00:00:00.01|   1005|2048|2048|2048   (0)|
|* 3|   TABLE ACCESS FULL           |T2       |     1|   11 |     1|00:00:00.01|   1005|    |    |          |
|* 4|  SORT JOIN                    |         |     1|    1 |     0|00:00:00.01|      3|2048|2048|2048   (0)|
|  5|   TABLE ACCESS BY INDEX ROWID|T1       |     1|    1 |     1|00:00:00.01|      3|    |    |          |
|* 6|    INDEX RANGE SCAN           |IDX_T1_N|     1|    1 |     1|00:00:00.01|      2|    |    |          |
-----------------------------------------------------------------------------------------------------
   3 - filter("T2"."N"=12)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
   6 - access("T1"."N"=19)
```

脚本 11-31　Merge Sort Join 中 T1 表限制条件有索引

接下来对 t2 表的限制条件建索引，发现如下 Merge Sort Join 更快了：

```
create index idx_t2_n on t2(n);
SELECT /*+ leading(t2) use_merge(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19
and t2.n=12;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------
```

```
|Id |Operation                    |Name    |Starts|E-Rows|A-Rows|   A-Time  |Buffers|OMem|1Mem| Used-Mem|
-------------------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT             |        |     1|      |     0|00:00:00.01|      6|    |    |         |
|  1| MERGE JOIN                  |        |     1|     1|     0|00:00:00.01|      6|    |    |         |
|  2|  SORT JOIN                  |        |     1|     1|     1|00:00:00.01|      3|2048|2048|2048  (0)|
|  3|   TABLE ACCESS BY INDEX ROWID|T2      |     1|     1|     1|00:00:00.01|      3|    |    |         |
|* 4|    INDEX RANGE SCAN         |IDX_T2_N|     1|     1|     1|00:00:00.01|      2|    |    |         |
|* 5|  SORT JOIN                  |        |     1|     1|     0|00:00:00.01|      3|2048|2048|2048  (0)|
|  6|   TABLE ACCESS BY INDEX ROWID|T1      |     1|     1|     1|00:00:00.01|      3|    |    |         |
|* 7|    INDEX RANGE SCAN         |IDX_T1_N|     1|     1|     1|00:00:00.01|      2|    |    |         |
-------------------------------------------------------------------------------------------------------

   4 - access("T2"."N"=12)
   5 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
   7 - access("T1"."N"=19)
```

脚本 11-32　Merge Sort Join 中 T2 表限制条件有索引

4. 第 2 式：连接条件索引消除排序(不完美)

首先看两表的连接条件都无索引的情况，如下，有两次排序：

```
set linesize 1000
set autotrace traceonly
SELECT /*+ leading(t1) use_merge(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id;

执行计划
--------
| Id  | Operation           | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     1 |   123 |       |  1852   (1)| 00:00:23 |
|   1 |  MERGE JOIN         |      |     1 |   123 |       |  1852   (1)| 00:00:23 |
|   2 |   SORT JOIN         |      |     1 |    57 |       |     4  (25)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL| T1   |     1 |    57 |       |     3   (0)| 00:00:01 |
|*  4 |   SORT JOIN         |      |   100K|  6445K|    15M|  1848   (1)| 00:00:23 |
|   5 |    TABLE ACCESS FULL| T2   |   100K|  6445K|       |   273   (1)| 00:00:04 |
-----------------------------------------------------------------------------------

   3 - filter("T1"."N"=19)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")

统计信息
--------
          0  recursive calls
          0  db block gets
       1012  consistent gets
          0  physical reads
          0  redo size
        880  bytes sent via SQL*Net to client
```

```
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

接下来在 t1 表建索引，发现排序消除了一个：

```
CREATE INDEX idx_t1_id ON t1(id);
set linesize 1000
set autotrace traceonly
SELECT /*+ leading(t1) use_merge(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id;
-----------------------
| Id  | Operation                    | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |   100 |   397K|       | 47930   (1)| 00:09:36 |
|   1 |  MERGE JOIN                  |           |   100 |   397K|       | 47930   (1)| 00:09:36 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1        |   100 |   198K|       |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | IDX_T1_ID |   100 |       |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |           |   111K|   217M|   582M| 47928   (1)| 00:09:36 |
|   5 |    TABLE ACCESS FULL         | T2        |   111K|   217M|       |   274   (1)| 00:00:04 |
-----------------------------------------------------------------------------------------------------
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")

统计信息
----------
          0  recursive calls
          0  db block gets
       1021  consistent gets
          0  physical reads
          0  redo size
      13432  bytes sent via SQL*Net to client
        481  bytes received via SQL*Net from client
          8  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
        100  rows processed
```

不过遗憾的是，接下来在 t2 表的连接条件建索引，发现排序依然有一个，无法消除：

```
CREATE INDEX idx_t2_t1_id ON t2(t1_id);
set linesize 1000
set autotrace traceonly
SELECT /*+ leading(t1) use_merge(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id;
-----------------------
| Id  | Operation                    | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
```

```
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |   100 |   397K|       | 38263   (1)| 00:07:40 |
|   1 |  MERGE JOIN                  |           |   100 |   397K|       | 38263   (1)| 00:07:40 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1        |   100 |   198K|       |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | IDX_T1_ID |   100 |       |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |           | 89127 |   173M|   464M| 38261   (1)| 00:07:40 |
|   5 |    TABLE ACCESS FULL         | T2        | 89127 |   173M|       |   273   (1)| 00:00:04 |
-----------------------------------------------------------------------------------------------------

   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
统计信息
--------
          0  recursive calls
          0  db block gets
       1021  consistent gets
          0  physical reads
          0  redo size
      13432  bytes sent via SQL*Net to client
        482  bytes received via SQL*Net from client
          8  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
        100  rows processed
```

脚本 11-33　Merge Sort Join 中消除排序的思路

结论：连接字段有索引，争取利用索引来消除排序（可惜的是，由于 Oracle 算法的限制，只能避免一次排序）。

5. 第 3 式：避免取多余列致排序的尺寸太大

Merge Sort Join 取所有字段的情况：

```
alter session set statistics_level=all ;
set linesize 1000
SELECT /*+ leading(t2) use_merge(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------------
|Id |Operation           |Name|Starts|E-Rows|A-Rows|   A-Time  |Buffers|OMem |1Mem |Used-Mem |
-----------------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT    |    |     1|      |     1 |00:00:00.14|   1012|     |     |         |
|  1| MERGE JOIN         |    |     1|     1|     1 |00:00:00.14|   1012|     |     |         |
|  2|  SORT JOIN         |    |     1|89127 |    20 |00:00:00.13|   1005|9266K|1184K|8236K (0)|
|  3|   TABLE ACCESS FULL| T2 |     1|89127 |  100K|00:00:00.03|   1005|     |     |         |
|* 4|  SORT JOIN         |    |    20|     1|     1 |00:00:00.01|      7|2048 |2048 |2048  (0)|
|* 5|   TABLE ACCESS FULL| T1 |     1|     1|     1 |00:00:00.01|      7|     |     |         |
```

```
-------------------------------------------------------------------------------------------------------
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
   5 - filter("T1"."N"=19)
```

脚本 11-34　Merge Sort Join 中取所有字段的情况

Merge Sort Join 取部分字段的情况：

```
SELECT /*+ leading(t2) use_merge(t1)*/ t1.id
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-------------------------------------------------------------------------------------------------------
|Id |Operation           |Name|Starts|E-Rows|A-Rows|   A-Time   |Buffers|OMem |1Mem|Used-Mem |
-------------------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT    |    |    1 |      |    1 |00:00:00.11|  1012 |     |    |         |
|  1| MERGE JOIN         |    |    1 |    1 |    1 |00:00:00.11|  1012 |     |    |         |
|  2|  SORT JOIN         |    |    1 |89127 |   20 |00:00:00.11|  1005 |1895K|658K|1684K (0)|
|  3|   TABLE ACCESS FULL| T2 |    1 |89127 |  100K|00:00:00.03|  1005 |     |    |         |
|* 4|  SORT JOIN         |    |   20 |    1 |    1 |00:00:00.01|     7 |2048 |2048|2048  (0)|
|* 5|   TABLE ACCESS FULL| T1 |    1 |    1 |    1 |00:00:00.01|     7 |     |    |         |
-------------------------------------------------------------------------------------------------------
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
   5 - filter("T1"."N"=19)
```

脚本 11-35　Merge Sort Join 中取部分字段的情况

从结果可以看出，取所有字段的 Used-Mem 是 8236KB，而取部分字段的 Used-Mem 是 1684KB，差异非常明显，前者内存消耗非常大！

6. 第 4 式：PGA 能完成排序，避免磁盘排序

应用的场景是参与排序合并连接的尺寸过大，这时候我们可以考虑增大 PGA。如果是 Oracle 11g，则默认是直接增大 memory_target，也可以选择手工管理。

11.3.4　一次统计信息收集不准确引发的 NL 性能瓶颈

在电信行业，80%的表连接都是 NL 连接，因为这是由 oltp 系统的特性决定的：虽然数据量很大，查询更新很频繁，但是最终需要查询返回及更新的数据量却很少。

但是若 NL 连接应用不当，会导致致命的问题，因为 NL 最大的特点是驱动表返回多少条，被驱动表被访问多少次，所以 NL 连接在驱动表返回极少的时候，性能不低。但是当驱动表返回很多的时候，性能却很低下，这时万万不可使用 NL 连接！举例如下：

```
--略去
from (select a.alarm_title,
```

```
            --略去其他字段
          from ne_alarm_list a
         where  a.alarm_type = 20
               and a.alarm_Level in ('1', '2')
               and a.alarm_state in (0)
               and  a.last_send_time  >=  TO_DATE('&P_DATE_BEGIN'  ||  '  00:00:00',
'YYYY-MM-DD hh24:mi:ss')
               and a.last_send_time <= TO_DATE('&P_DATE_END' || ' 23:59:59', 'YYYY-
MM-DD hh24:mi:ss')
        ) a1,
      manage_region b,
      net_element d,
      (select *
         from tp_domain_listvalues
        where domain_code LIKE 'DOMAIN_ALARM_STATE%') f,
      kpi_code_list g,
      ne_alarm_msg_source_rela h,
      net_element k,
      manage_region l
 where a1.alarm_region_origin = b.region_id(+)
   and a1.ne_id = d.ne_id
   --and instr(d.path, :P_NE_ID, 1, 1) > 0
   and a1.alarm_state = f.list_value
   and a1.kpi_id = g.kpi_id
   and a1.ne_alarm_list_id = h.ne_alarm_msg_id
   and h.source_type = '19'
   and h.source_id = k.ne_id
   and k.region_id = l.region_id
   and l.region_id in ('&PSOURCE_REGION')
   order by a1.alarm_state, a1.create_time desc
```

截取部分执行计划分析，我们惊奇地发现，Oracle 预估 NE_ALARM_MSG_SOURCE_ RELA 表在 source_type='19'条件下返回 1 条记录，实际返回了 477K 条，预估严重不准确！怪不得会使用 NL 连接。结果 NL 连接导致后续的被驱动表被访问了 477K 次，真是惊人的数字！

```
| Id  | Operation                      | Name                      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |  OMem
-------------------------------------------------------------------------------------------------------------------------------
|  10 |    NESTED LOOPS                |                           |      1 |        |    118 |00:00:18.92 |    3355K|
|  11 |     NESTED LOOPS               |                           |      1 |      1 |    118 |00:00:18.92 |    3355K|
|* 12 |      HASH JOIN                 |                           |      1 |      1 |    118 |00:00:18.92 |    3355K|
|  13 |       NESTED LOOPS             |                           |      1 |      1 |    118 |00:00:18.92 |    3354K|
|  14 |        NESTED LOOPS            |                           |      1 |      1 |    477K|00:00:11.46 |    1917K|
|  15 |         NESTED LOOPS           |                           |      1 |      1 |    477K|00:00:07.17 |    1439K|
|* 16 |          TABLE ACCESS FULL     | NE_ALARM_MSG_SOURCE_RELA  |      1 |      1 |    477K|00:00:00.48 |    6683 |
|  17 |          TABLE ACCESS BY INDEX ROWID| NET_ELEMENT          |    477K|      1 |    477K|00:00:05.97 |    1432K|
|* 18 |           INDEX UNIQUE SCAN    | PK_NET_ELEMENT            |    477K|      1 |    477K|00:00:03.61 |     955K|
|  19 |         TABLE ACCESS BY INDEX ROWID | MANAGE_REGION        |    477K|      1 |    477K|00:00:03.57 |     477K|
|* 20 |          INDEX UNIQUE SCAN     | PK_MANAGE_REGION          |    477K|      1 |    477K|00:00:01.41 |       2 |
|* 21 |        TABLE ACCESS BY INDEX ROWID | NE_ALARM_LIST         |    477K|      1 |    118 |00:00:07.03 |    1437K|
|* 22 |         INDEX UNIQUE SCAN      | PK_NE_ALARM_LIST          |    477K|      1 |    477K|00:00:03.44 |     955K|
|* 23 |       TABLE ACCESS FULL        | TP_DOMAIN_LISTVALUES      |      1 |    246 |      2 |00:00:00.01 |      62 |
|  24 |      TABLE ACCESS BY INDEX ROWID | KPI_CODE_LIST           |    118 |      1 |    118 |00:00:00.01 |     238 |
|* 25 |       INDEX UNIQUE SCAN        | PK_KPI_CODE_LIST          |    118 |      1 |    118 |00:00:00.01 |     120 |
|  26 |     TABLE ACCESS BY INDEX ROWID | NET_ELEMENT              |    118 |      1 |    118 |00:00:00.01 |     356 |
|* 27 |      INDEX UNIQUE SCAN         | PK_NET_ELEMENT            |    118 |      1 |    118 |00:00:00.01 |     238 |
|  28 |    TABLE ACCESS BY INDEX ROWID | MANAGE_REGION             |    118 |      1 |    118 |00:00:00.01 |     120 |
|* 29 |     INDEX UNIQUE SCAN          | PK_MANAGE_REGION          |    118 |      1 |    118 |00:00:00.01 |       2 |

                                                    --  -------  --  -  -------  -  -----    --  -  ---
                                                   16 - filter("H"."SOURCE_TYPE"='19')
```

那该如何优化呢？没有定式，有的时候要改写 SQL，有的时候需要增加索引，而有的时候，却只要重新收集一下表的统计信息或者直方图即可。而此时，我们非常清楚地判断出来，Oracle 对 NE_ALARM_MSG_SOURCE_RELA 表的 SOURCE_TYPE 列的数据分析严重错误了。

实际这张表有快 200 万条记录，而这个表的 SOURCE_TYPE 列只有 3 个取值，多是返回结果。因此我们要做的事情非常简单，重新收集这个表的列的直方图即可，如下：

```
exec dbms_stats.gather_table_stats(ownname => 'BOSSWG',tabname => 'NE_ALARM_MSG_SOURCE_
RELA',estimate_percent => 10,method_opt=>'for all columns size 254',cascade=>TRUE) ;
```

新的执行计划如下，即从 NL 更新为部分 HASH 了。

```
|* 12 |     HASH JOIN                        |                                  |      1 |     66 |    946 |00:00:00.16 |   15165 |
|  13 |      NESTED LOOPS                     |                                  |      1 |     65 |    946 |00:00:00.15 |   15039 |
|* 14 |       HASH JOIN                       |                                  |      1 |     65 |    946 |00:00:00.14 |   12199 |
|  15 |        NESTED LOOPS                   |                                  |      1 |     65 |    946 |00:00:00.14 |   12161 |
|  16 |         NESTED LOOPS                  |                                  |      1 |    172 |    946 |00:00:00.13 |    9321 |
|* 17 |          TABLE ACCESS BY INDEX ROWID| NE_ALARM_LIST                    |      1 |    140 |    946 |00:00:00.11 |    6481 |
|* 18 |           INDEX RANGE SCAN            | IDX_LAST_SEND_TIME               |      1 |   6448 |   8304 |00:00:00.06 |      73 |
|  19 |          TABLE ACCESS BY INDEX ROWID| NE_ALARM_MSG_SOURCE_RELA         |    946 |      1 |    946 |00:00:00.02 |    2840 |
|* 20 |           INDEX UNIQUE SCAN           | PK_NE_ALARM_MSG_SOURCE_RELA      |    946 |      1 |    946 |00:00:00.01 |    1894 |
|  21 |         TABLE ACCESS BY INDEX ROWID | NET_ELEMENT                      |    946 |      1 |    946 |00:00:00.01 |    2840 |
|* 22 |          INDEX UNIQUE SCAN            | PK_NET_ELEMENT                   |    946 |      1 |    946 |00:00:00.01 |    1894 |
|  23 |        TABLE ACCESS FULL              | MANAGE_REGION                    |      1 |     91 |     91 |00:00:00.01 |      38 |
|  24 |       TABLE ACCESS BY INDEX ROWID     | NET_ELEMENT                      |    946 |      1 |    946 |00:00:00.01 |    2840 |
|* 25 |        INDEX UNIQUE SCAN              | PK_NET_ELEMENT                   |    946 |      1 |    946 |00:00:00.01 |    1894 |
|  26 |      TABLE ACCESS FULL                | KPI_CODE_LIST                    |      1 |   6284 |   6288 |00:00:00.01 |     126 |
|* 27 |     TABLE ACCESS FULL                 | TP_DOMAIN_LISTVALUES             |      1 |    257 |      2 |00:00:00.01 |      62 |
```

执行时间从原先的 20s 缩减为 0.2s。

让表连接提速的方法

连接类型	优化要点	举　例	文中章节
Nested Loops Join（确保用在 OLTP 场景）	驱动表的限制条件要考虑建立索引	对驱动表（t1 表）的限制条件建索引：CREATE INDEX t1_n ON t1 (n)后，Nested Loops Join 性能略有提升	11.3.1
	被驱动表的限制条件要考虑建立索引	对被驱动表 t2 表的连接条件建索引：CREATE INDEX t2_t1_id ON t2(t1_id)，Nested Loops Join 性能有大幅度提升	11.3.1
	确保小结果集先驱动，大的被驱动	倒过来性能差异巨大	11.3.1
Hash Join（确保用在 OLTP 场景）	两表限制条件有索引	create index idx_t1_n on t1(n); create index idx_t2_n on t2(n);	11.3.2
	要小结果集先驱动，大的被驱动	倒过来性能差异巨大	11.3.2
	尽量保证 PGA 能容纳 Hash 运算	磁盘排序的开销将非常大	11.3.2
Merge Sort Join（确保用在 OLTP 场景）	两表限制条件有索引	create index idx_t1_n on t1(n); create index idx_t2_n on t2(n);	11.3.3
	连接条件索引消除排序（不完美）	连接字段有索引，争取利用索引来消除排序（可惜的是，Oracle 算法的限制，只能避免一次排序）	11.3.3

续表

连接类型	优化要点	举　例	文中章节
Merge Sort Join（确保用在 OLTP 场景）	避免取多余列致排序的尺寸太大	这是由于只取部分字段时，参与排序的尺寸会小很多	11.3.3
	PGA 能完成排序，避免磁盘排序	磁盘排序的开销将非常大	11.3.3

11.4　本章习题、总结与延伸

习题 1：请分析如下语句的执行计划，并回答问题。

```
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |      1 |00:00:00.01 |    1014 |
|   1 |  NESTED LOOPS      |      |      1 |      1 |      1 |00:00:00.01 |    1014 |
|*  2 |   TABLE ACCESS FULL| T1   |      1 |      1 |      1 |00:00:00.01 |       8 |
|*  3 |   TABLE ACCESS FULL| T2   |      1 |      1 |      1 |00:00:00.01 |    1006 |
-------------------------------------------------------------------------------------------
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"="T2"."T1_ID")
```

提问：t1 表有记录 100 条，t2 表有记录 10 万条，请问这个执行计划高效吗，为什么？你有办法提升这个语句的性能吗？

习题 2：请分析如下语句的执行计划，并回答问题。

```
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | E-Rows | A-Rows |   A-Time   |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |        |   8000 |00:00:00.06 |
|   1 |  NESTED LOOPS                |      |      1 |      1 |   8000 |00:00:00.06 |
| * 2 |   TABLE ACCESS FULL          | T1   |      1 |   8000 |      1 |00:00:00.01 |
|*  4 |   TABLE ACCESS FULL          | T2   |   8000 |     18 |   7982 |00:00:00.05 |
-----------------------------------------------------------------------------------------
   3 - access("T1"."N"=19)
   4 - filter("T1"."ID" like "T2"."T1_ID")
```

提问：t1 表有记录 1 万条，t2 表有记录 10 万条，请问这个执行计划高效吗，为什么？你有办法提升这个语句的性能吗？

习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

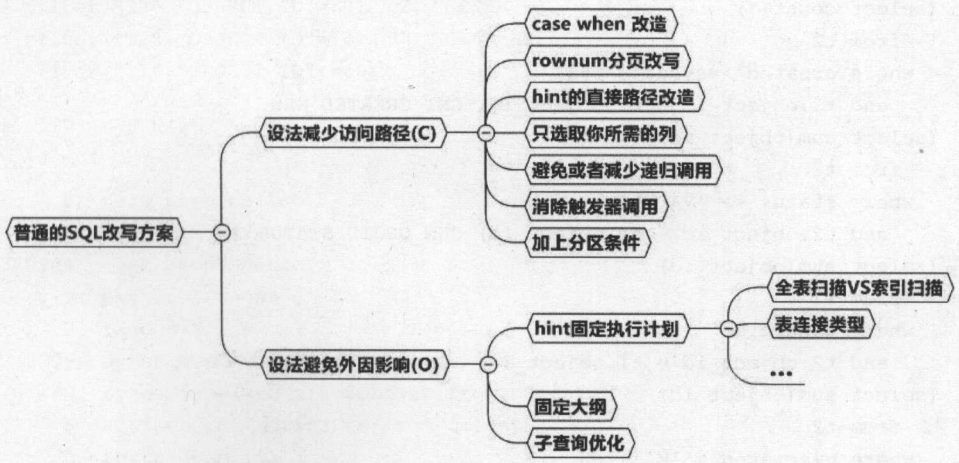
第 12 章　动手，经典等价改写让 SQL 飞

其实等价与否总在一念之间

SQL 优化的本质就是减少访问路径，前面的章节中我们已经学到了很多减少访问路径的思路，比如增加索引从全表扫描转换成索引范围扫描，比如把表改造成分区表从而从全表扫描转化成局部分区扫描，这些都属于不需要改写 SQL 就能完成的减少访问路径的思路。当然，在很多场景下，我们必须要完成一些等价改写，比如 case when 改造、rownum 分页改写，等等。

除了减少访问路径外，还要注意避免外因的影响，比如，一些执行计划不稳定，所在环境的资源不足，等等，这些也是我们需要注意的。

总体思路如下图：



12.1　设法减少访问路径

接下来我们从 Case When、Rownum、Hint 直接路径改造、只取所需列、避免递归调用、

rowid 优化这 6 个案例展开，向读者展现不同写法前后的优化效果。

12.1.1 Case When 改造

以下 SQL 是一个非常经典的案例，其中 CNT_TEMPORARY_Y、CNT_CREATED_NEW、SUM_OBJID_STATUS_V、SUM_OBJID_GENERATED_、SUM_OBJID_GENERATED_M、SUM_OBJID_GENERATED_Q 这 6 处是构造的列，作为结果集要展现出来。实战写法中 t 表被访问了多次，具体如下：

```
drop table t1 purge;
drop table t2 purge;
create table t1 as select * from dba_objects ;
create table t2 as select * from dba_objects ;
update t2 set status='INVALID' WHERE ROWNUM<=10000;
update t2 set generated='Y' WHERE ROWNUM<=10000;
update t2 set temporary='Y' WHERE ROWNUM<=10000;
update t2 set temporary='M' WHERE temporary<>'Y';
update t2 set temporary='Q' WHERE temporary<>'Y' or temporary<>'M';
COMMIT;

set autotrace traceonly
set linesize 1000

select t1.object_name,
       t1.object_id,
       (select count(*)
          from t2
         where temporary = 'Y'
           and t2.object_id = t1.object_id) CNT_TEMPORARY_Y,
       (select count(*)
          from t2
         where created >=sysdate-365
           and t2.object_id = t1.object_id) CNT_CREATED_NEW,
       (select sum(object_id)
          from t2
         where status <> 'VALUD'
           and t2.object_id = t1.object_id) SUM_OBJID_STATUS_V,
       (select sum(object_id)
          from t2
         where generated = 'Y'
           and t2.object_id = t1.object_id)  SUM_OBJID_GENERATED_Y,
       (select sum(object_id)
          from t2
         where generated = 'M'
           and t2.object_id = t1.object_id)  SUM_OBJID_GENERATED_M,
       (select sum(object_id)
          from t2
         where generated = 'Q'
           and t2.object_id = t1.object_id)  SUM_OBJID_GENERATED_Q
```

```
  from t1
 where t1.object_id <= 50;
执行计划
--------
Plan hash value: 2340670826
---------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    47 |  1410 |   285   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE     |      |     1 |     7 |            |          |
|*  2 |   TABLE ACCESS FULL| T2   |     1 |     7 |   286   (1)| 00:00:04 |
|   3 |  SORT AGGREGATE     |      |     1 |    13 |            |          |
|*  4 |   TABLE ACCESS FULL| T2   |     1 |    13 |   285   (1)| 00:00:04 |
|   5 |  SORT AGGREGATE     |      |     1 |    12 |            |          |
|*  6 |   TABLE ACCESS FULL| T2   |     1 |    12 |   286   (1)| 00:00:04 |
|   7 |  SORT AGGREGATE     |      |     1 |     7 |            |          |
|*  8 |   TABLE ACCESS FULL| T2   |     1 |     7 |   286   (1)| 00:00:04 |
|   9 |  SORT AGGREGATE     |      |     1 |     7 |            |          |
|* 10 |   TABLE ACCESS FULL| T2   |     1 |     7 |   286   (1)| 00:00:04 |
|  11 |  SORT AGGREGATE     |      |     1 |     7 |            |          |
|* 12 |   TABLE ACCESS FULL| T2   |     1 |     7 |   286   (1)| 00:00:04 |
|* 13 |  TABLE ACCESS FULL  | T1   |    47 |  1410 |   285   (1)| 00:00:04 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("T2"."OBJECT_ID"=:B1 AND "TEMPORARY"='Y')
   4 - filter("T2"."OBJECT_ID"=:B1 AND "CREATED">=SYSDATE@!-365)
   6 - filter("T2"."OBJECT_ID"=:B1 AND "STATUS"<>'VALUD')
   8 - filter("T2"."OBJECT_ID"=:B1 AND "GENERATED"='Y')
  10 - filter("T2"."OBJECT_ID"=:B1 AND "GENERATED"='M')
  12 - filter("T2"."OBJECT_ID"=:B1 AND "GENERATED"='Q')
  13 - filter("T1"."OBJECT_ID"<=50)

统计信息
--------
          0  recursive calls
          0  db block gets
     300904  consistent gets
          0  physical reads
          0  redo size
       2379  bytes sent via SQL*Net to client
        449  bytes received via SQL*Net from client
          5  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         49  rows processed
```

脚本 12-1　让表被访问多次的低效写法

这里通过执行计划，可以看出 t2 表被访问了 6 次，该 SQL 语句的逻辑读高达 300904！

来，看看 case when 改造的写法 2，如下：

```
with w_t2 as
(select
       t2.object_id,
       count(case when t2.temporary='Y' then 1 end ) CNT_TEMPORARY_Y,
       count(case when created >=sysdate-365  then 1 end ) CNT_CREATED_NEW,
       sum(case when t2.status<>'VALID' then t2.object_id end ) SUM_OBJID_STATUS_V,
       sum(case when t2.generated = 'Y' then t2.object_id end ) SUM_OBJID_GENERATED_Y,
       sum(case when t2.generated = 'M' then t2.object_id end ) SUM_OBJID_GENERATED_M,
       sum(case when t2.generated = 'Q' then t2.object_id end ) SUM_OBJID_GENERATED_Q
 from  t2
 group by t2.object_id)
select t1.object_name,t1.object_id,w_t2.* from t1,w_t2
where t1.object_id=w_t2.object_id
and t1.object_id<=50;

执行计划
---------
Plan hash value: 3226881135
-------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |    47 |  4512 |   572   (1)| 00:00:07 |
|   1 |  HASH GROUP BY       |      |    47 |  4512 |   572   (1)| 00:00:07 |
|*  2 |   HASH JOIN          |      |    47 |  4512 |   571   (1)| 00:00:07 |
|*  3 |    TABLE ACCESS FULL| T1   |    47 |  3384 |   285   (1)| 00:00:04 |
|*  4 |    TABLE ACCESS FULL| T2   |    47 |  1128 |   285   (1)| 00:00:04 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("T1"."OBJECT_ID"="T2"."OBJECT_ID")
   3 - filter("T1"."OBJECT_ID"<=50)
   4 - filter("T2"."OBJECT_ID"<=50)
统计信息
--------
          0  recursive calls
          0  db block gets
       2040  consistent gets
          0  physical reads
          0  redo size
       2633  bytes sent via SQL*Net to client
        449  bytes received via SQL*Net from client
          5  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         49  rows processed
```

脚本 12-2　CASE WHEN 改造后的高效写法

观察写法 2 的执行计划，我们发现 t2 表的访问次数仅 1 次。而且逻辑读为 2040！差异如此巨大的本质原因就是写法 2 通过 CASE WHEN 的经典写法，将这些查询进行了合并，减少了表访问次数，自然就大幅度提升了性能！

12.1.2　Rownum 分页改写

写法 1：

```
drop table test_rownum purge;
create table test_rownum as select * from dba_objects;
select count(*) from test_rownum;
   COUNT(*)
 ----------
    111052
alter session set statistics_level=all ;
 set linesize 1000
 set pagesize 500
 select * from (select t.*,rownum as rn   from test_rownum t) a where a.rn>=1 and
a.rn<=10;

 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

 SQL_ID  b3z5zu2apz8zu, child number 0
 -------------------------------------
 select * from (select t.*,rownum as rn   from test_rownum t) a where a.rn>=1 and
a.rn<=10

 Plan hash value: 1157459331
 ------------------------------------------------------------------------------------------
 | Id  |Operation           | Name       |Starts |E-Rows |A-Rows |   A-Time  |Buffers |
 ------------------------------------------------------------------------------------------
 |*  1 | VIEW               |            |     1 |   101K|    10 |00:00:00.16|   1723 |
 |   2 |  COUNT             |            |     1 |       |   111K|00:00:00.01|   1723 |
 |   3 |   TABLE ACCESS FULL| TEST_ROWNUM|     1 |   101K|   111K|00:00:00.01|   1723 |
 ------------------------------------------------------------------------------------------
0  sorts (disk)
        11  rows processed
```

脚本 12-3　Rownum 分页的普通写法

写法 2：

```
select * from (select t.*,rownum as rn  from test_rownum t where rownum<=10) a where
a.rn>=1;
 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
 PLAN_TABLE_OUTPUT
 ------------------------------------------------------------------------------------------
--------
 SQL_ID  5jvt2xsjrqc33, child number 0
```

```
------------------------------------
select * from (select t.*,rownum as rn  from test_rownum t where rownum<=10) a where
a.rn>=1

Plan hash value: 2232708344
-------------------------------------------------------------------------------------
| Id |Operation           |Name        |Starts |E-Rows |A-Rows |   A-Time   |Buffers |
-------------------------------------------------------------------------------------
|*  1| VIEW               |            |     1 |    10 |    10 |00:00:00.01|       5 |
|*  2|  COUNT STOPKEY     |            |     1 |       |    10 |00:00:00.01|       5 |
|   3|   TABLE ACCESS FULL|TEST_ROWNUM|     1 |   101K|    10 |00:00:00.01|       5 |
-------------------------------------------------------------------------------------
```

脚本 12-4　Rownum 分页的高效写法

写法 1 的 Buffers 是 1723，而写法 2 是 5，为啥会有如此巨大的差异！认真看执行计划就能明白其中的原委。写法 1 的执行计划 Id=2 处的 Operation 对应的关键字是 COUNT，而写法 1 的执行计划 Id=2 处的 Operation 对应的关键字是 COUNT STOPKEY。写法 2 对应的 A-ROWS 是 10，这意味着该 SQL 仅扫描 10 条记录，显然是局部扫描。而写法 1 是 111K，这意味着全表每条记录都扫描了。

12.1.3　Hint 直接路径改造

构造环境：

```
SET AUTOTRACE OFF
DROP TABLE T1 PURGE ;
DROP TABLE T2 PURGE ;
DROP TABLE T3 PURGE ;
DROP TABLE T4 PURGE ;
DROP TABLE T PURGE ;

CREATE TABLE T1 AS SELECT * FROM DBA_OBJECTS where 1=2;
CREATE TABLE T2 AS SELECT * FROM DBA_OBJECTS where 1=2;
CREATE TABLE T3 AS SELECT * FROM DBA_OBJECTS where 1=2;
CREATE TABLE T4 AS SELECT * FROM DBA_OBJECTS where 1=2;
CREATE TABLE T AS SELECT * FROM DBA_OBJECTS ;
INSERT INTO T SELECT * FROM T;
INSERT INTO T SELECT * FROM T;
INSERT INTO T SELECT * FROM T;
INSERT INTO T SELECT * FROM T;
INSERT INTO T SELECT * FROM T;
COMMIT;

alter table T4 nologging;
alter session enable parallel dml;
set linesize 1000
set timing on
```

脚本 12-5　Hint 直接路径改造的环境构造

比较两表插入的速度：

```
SQL> insert  into T1 select * from T;
已创建 2207936 行。

已用时间:  00: 00: 15.22
SQL> insert  /*+append*/ into T2 select * from T;

已创建 2207936 行。
已用时间:  00: 00: 05.22
```

脚本 12-6　普通插入与直接路径写的性能差异

插入 t1 表和 t2 表的速度差异如此之大，说明绕过 SGA 的直接路径访问性能提升非常明显，值得在特定的场合下使用。

12.1.4　只取你所需的列

1. 只取所需列，访问视图变快

环境准备，建一个视图 v_t1_join_t2：

```
drop table t1 cascade constraints purge;
drop table t2 cascade constraints  purge;
create table t1 as select * from dba_objects;
create table t2 as select * from dba_objects where rownum<=10000;
update t1 set object_id=rownum ;
update t2 set object_id=rownum ;
commit;
create or replace view v_t1_join_t2
as select t2.object_id,t2.object_name,
t1.object_type,t1.owner from t1,t2
where t1.object_id=t2.object_id;
alter table T1 add constraint
pk_object_id primary key (OBJECT_ID);
alter table T2 add constraint
fk_objecdt_id foreign key (OBJECT_ID) references t1 (OBJECT_ID);
```

执行 select * from v_t1_join_t2 语句，必须要访问 t2 和 t1 表，如下：

```
SQL> set autotrace traceonly
SQL> set linesize 1000
Predicate Information (identified by operation id):
---------------------------------------------------
SQL> select * from v_t1_join_t2;
已选择 10000 行。
执行计划
--------
Plan hash value: 2959412835
---------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT    |       |  7996 |   937K|   335   (1)| 00:00:05 |
|*  1 |  HASH JOIN          |       |  7996 |   937K|   335   (1)| 00:00:05 |
|   2 |   TABLE ACCESS FULL| T2    |  7996 |   616K|    27   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T1    |   110K|  4406K|   307   (1)| 00:00:04 |
-------------------------------------------------------------------------------
```

脚本 12-7　普通访问视图的写法

但是有时会出现这样的情况：开发人员实际只需要取 object_id、object_name 两个列，但是他们为了简单，直接使用 select * from 先把所有列取回本地，再过滤 object_id、object_name 这两列，这时其实 select * from v_t1_join_t2 语句是等同于 select object_id,object_name from v_t1_join_t2 的。那我们看看这个语句执行后是啥情况，如下：

```
SQL> select object_id,object_name from v_t1_join_t2;
已选择 10000 行。
执行计划
--------
Plan hash value: 1513984157
---------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |  7996 |   616K|    27   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T2   |  7996 |   616K|    27   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

脚本 12-8　只取部分列后，访问视图有变化

我们惊奇地发现，这个语句只访问了 t2 表。这是为啥呢？因为 object_id、object_name 这两个列全部来自 t2 表，而且这个视图中的 t1 和 t2 表有主外键关联，确保了只取 t2 表记录不会取错。这样 Oracle 既保障了高效又确保了记录的准确。

2. 只取所需列，索引无须回表

场景 1，一个普通的利用索引查询的 SQL 语句，如下：

```
drop table t purge;
create table t as select * from dba_objects;
set linesize 1000
set autotrace traceonly
drop index idx_obj_idtp;
create index idx_obj_idtp on t(object_id,object_type);
SQL> select * from t where object_id=28;
执行计划
--------
Plan hash value: 2615995481
---------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |     1 |   207 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T            |     1 |   207 |     3   (0)| 00:00:01 |
```

```
|*  2 |   INDEX RANGE SCAN          | IDX_OBJ_IDTP |     1 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
--略去
```

脚本 12-9　普通的利用索引的查询

该语句通过 IDX_OBJ_ID 定位到了 rowid，然后通过 TABLE ACCESS BY INDEX ROWID 回到表中，获取到了 object_id 列为*的信息，然后将结果展现。不过假如这里的情况和上述的情况类似，开发人员只是通过 select * 把所有列都取回来，再取 object_id、object_type 两列，实际上这和 select object_id,object_type from t where object_id=28 等价。但是这么写性能可不一样了，由于 object_type 列的信息在索引中已经有了，所以就无须回表了。如下：

```
SQL> select object_id,object_type from t where object_id=28;
执行计划
--------
Plan hash value: 4115116200
---------------------------
| Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |     1 |    24 |     2   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| IDX_OBJ_IDTP |     1 |    24 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
```

脚本 12-10　利用组合索引消除回表的索引查询

12.1.5　避免或者减少递归调用

避免或者减少函数调用，这一个是非常重要的优化要点，也是一个非常常用的优化手段，现实中类似的案例场景也非常常见。一般来说，能避免函数调用就避免，一般是将函数调用改写成表连接的模式。但是有的时候函数调用不可避免，比如一些非常复杂的逻辑封装在函数中，一般人员实现起来比较困难，且使用面又广，那就只有使用函数调用，不过遇到这个场景，我们依然可以优化，那就是通过写法的优化，将函数调用的次数降低。

首先我们看看函数调用避免的手法和对应案例。

1. 避免 SQL 函数调用有啥好处

构造一个这样的案例，people 表中有性别字段，这个性别字段的取值（不是男就是女）来源于一张性别表 sex。现在有一个简单的需求，要查出 people 表所有字段的详细记录，具体环境准备如下：

```
drop table people purge;
drop table sex purge;

create table people (first_name varchar2(200),last_name varchar2(200),sex_id number);
create table sex (name varchar2(20), sex_id number);
insert into people (first_name,last_name,sex_id) select object_name,object_type,1
from dba_objects;
```

```
insert into sex (name,sex_id) values ('男',1);
insert into sex (name,sex_id) values ('女',2);
insert into sex (name,sex_id) values ('不详',3);
commit;

create or replace function get_sex_name(p_id sex.sex_id%type) return sex.name%type is
v_name sex.name%type;
begin
select name
into v_name
from sex
where sex_id=p_id;
return v_name;
end;
/
```

脚本 12-11　函数调用性能研究的环境准备

以下两种写法是等价的，都是为了查询 people 表信息，同时通过 sex 表，获取人员的性别信息。

```
写法 1
select sex_id,
first_name||' '||last_name full_name,
get_sex_name(sex_id) gender
from people;

写法 2
select p.sex_id,
p.first_name||' '||p.last_name full_name,
sex.name
from people p, sex
where sex.sex_id=p.sex_id;
```

脚本 12-12　函数调用和两表关联两个等价写法

通过 autotrace 比较观察，发现两种写法在性能上存在巨大差异，首先跟踪写法 1：

```
set autotrace traceonly  statistics
select sex_id,
first_name||' '||last_name full_name,
get_sex_name(sex_id) gender
from people;
已选择 111120 行。
统计信息
--------
     111120  recursive calls
          0  db block gets
     785917  consistent gets
          0  physical reads
          0  redo size
    3974937  bytes sent via SQL*Net to client
```

```
      81893  bytes received via SQL*Net from client
       7409  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
     111120  rows processed
```

脚本 12-13　函数调用写法的性能

接下来跟踪写法 2：

```
select p.sex_id,
p.first_name||' '||p.last_name full_name,
sex.name
from people p, sex
where sex.sex_id=p.sex_id;

已选择 111120 行。

统计信息
--------
          0  recursive calls
          0  db block gets
       8084  consistent gets
          0  physical reads
          0  redo size
    3974935  bytes sent via SQL*Net to client
      81893  bytes received via SQL*Net from client
       7409  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
     111120  rows processed
```

脚本 12-14　两表关联写法的性能

结论：

这里的性能差异非常明显，究其本质原因，就是写法 1 进行了过多的函数调用。

2. 减少 SQL 函数调用有何思路

前面说过了，很多时候函数调用不可避免的时候只有想办法降低函数调用次数，降低函数调用次数有两个主要思路：

- 尽量将函数写在聚合汇集结果集后而不是写在之前，显然聚合后调用会降低调用次数。
- 当函数在取值条件的位置时，可以考虑函数索引来减少递归调用。

（1）函数写法的位置

首先我们研究第一种场景，构造两个函数 f_dea11 和 f_dea12，这两个函数其实是一模一样的，构造这个仅仅是为了说明函数索引可以减少递归调用，具体试验如下：

```
drop table t1 purge;
drop table t2 purge;
create table t1 as select * from dba_objects;
create table t2 as select * from dba_objects;
update t2 set object_id=rownum;
commit;

create or replace function f_deal1(p_name in varchar2)
 return  varchar2 deterministic
 is
v_name varchar2(200);
begin
  -- select substr(upper(p_name),1,4)   into v_name from dual;
   v_name:=substr(upper(p_name),1,4);
   return v_name;
end;
/
create or replace function f_deal2(p_name in varchar2)
 return  varchar2 deterministic
 is
v_name varchar2(200);
begin
  select substr(upper(p_name),1,4)   into v_name from dual;
  --  v_name:=substr(upper(p_name),1,4);
   return v_name;
end;
/
```

脚本 12-15　构造环境，建函数

我们观察如下两个等价语句的性能差异，首先看写法 1：

```
set autotrace traceonly statistics
set linesize 1000
select name from (select rownum rn ,f_deal1(t1.object_name) name from t1) where
rn<=12;
统计信息
--------
    111113  recursive calls
         0  db block gets
      1724  consistent gets
         0  physical reads
         0  redo size
       558  bytes sent via SQL*Net to client
       416  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
        12  rows processed
```

脚本 12-16　函数调用在汇集结果集之前

接下来看写法 2：

```
select f_deal1(t1.object_name) name from t1 where rownum<=12;
统计信息
--------
         12  recursive calls
          0  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
        558  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
rows processed
```

脚本 12-17　函数调用在汇集结果集之后

（2）用函数索引优化

当函数在 where 条件下调用，且在没有建函数索引时，结果如下：

```
select * from t1 where f_deal2(t1.object_name)='AAAA';
统计信息
--------
     111111  recursive calls
          0  db block gets
       1723  consistent gets
          0  physical reads
          0  redo size
       1124  bytes sent via SQL*Net to client
        405  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

脚本 12-18　未建函数索引的函数调用

当函数在 where 条件下调用，且建了函数索引，结果如下：

```
统计信息
--------
          0  recursive calls
          0  db block gets
       1723  consistent gets
          0  physical reads
          0  redo size
       1124  bytes sent via SQL*Net to client
        405  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
```

```
         0  sorts (memory)
         0  sorts (disk)
         0  rows processed
```

脚本 12-19　建函数索引的函数调用

当没有建函数索引时产生 111111 次递归调用，而建了后递归调用为 0，性能得到大幅度提升。

3. 避免使用触发器

建表触发器：

```
set pagesize 3000;
set linesize 300;
set trim on

--creating table
drop table t purge;
create table t(msg_id number(15), user_id number(9), msg varchar2(4000));

--creating trigger
create trigger tri_t
  before insert on t
  for each row
declare
  v_cnt number(1);
begin
  select count(*)
    into v_cnt
    from t
   where msg_id = :new.msg_id
     and rownum = 1;
end tri_t;
/
```

脚本 12-20　建表触发器

写法 1，t 表有触发器时，插入耗时 46s：

```
SQL> set timing on
SQL> --using trigger
SQL> alter trigger tri_t enable;
触发器已更改
已用时间:  00: 00: 00.03
SQL> declare
  2    v_msgid number(15);
  3    v_usrid number(9);
  4  begin
  5    dbms_random.seed(1000000000);
  6    for i in 1 .. 30000 loop
```

```
  7      v_usrid := dbms_random.value;
  8      v_msgid := i;
  9           execute immediate 'insert into t (msg_id, user_id, msg) values
(:p1, :p2,:p3)'
 10        using v_msgid, v_usrid, 'msg_' || i;
 11    end loop;
 12  end;
 13  /
PL/SQL 过程已成功完成。
已用时间:  00: 00: 46.62
```

脚本 12-21　触发器生效时的插入性能

写法 2，触发器失效的情况，插入耗时仅 1s 多：

```
SQL> truncate table t;
表被截断。
已用时间:  00: 00: 00.35
SQL> --not using trigger
SQL> alter trigger tri_t disable;
触发器已更改
已用时间:  00: 00: 00.00
SQL> declare
  2    v_msgid number(15);
  3    v_usrid number(9);
  4  begin
  5    dbms_random.seed(1000000000);
  6    for i in 1 .. 30000 loop
  7      v_usrid := dbms_random.value;
  8      v_msgid := i;
  9      execute immediate 'insert into t (msg_id, user_id, msg) values (:p1, :p2, :p3)'
 10        using v_msgid, v_usrid, 'msg_' || i;
 11    end loop;
 12  end;
 13  /
PL/SQL 过程已成功完成。
已用时间:  00: 00: 01.81
```

脚本 12-22　触发器失效时的插入性能

性能为何差异如此之大，很明显就在于：写法 1 每插入 t1 表一条记录，就调用触发器完成一次统计查询，从而产生了大量的递归调用；而写法 2 则是消除了触发器。这是现实中的一个案例，后续和开发人员确认该统计可以去掉，去掉后性能得以大幅度提升。

类似地，如果触发器调用的是更新语句，那么可以将其修改成程序批量完成，而非每条触发，性能也能大幅度提升。

12.1.6　ROWID 优化应用

ROWID 是一个伪列，即是一个非用户定义的列，而又实际存储于数据库之中。每一个表都

有一个 ROWID 列，一个 ROWID 值用于唯一确定数据库表中的一条记录。因此通过 ROWID 方式来访问数据也是 Oracle 数据库访问数据的实现方式之一。由于 Oracle ROWID 能够直接定位一条记录，因此使用 ROWID 方式来访问数据，访问数据的效率非常高！

环境准备：

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id);
```

首先我们来看看最普通的访问方式，全表扫描访问：

```
set linesize 1000
set autotrace on

select /*+full(t) */ object_name from t where object_id=2;

OBJECT_NAME
------------
C_OBJ#

执行计划
--------
Plan hash value: 1601196873
----------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    79 |   307   (1)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| T    |     1 |    79 |   307   (1)| 00:00:04 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1724  consistent gets
```

脚本 12-23　全表扫描

接下来我们来看看索引扫描方式：

```
select  object_name from t where object_id=2;

OBJECT_NAME
----------
C_OBJ#

执行计划
--------
Plan hash value: 2041828949
---------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time  |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1 |    79 |     2   (0)| 00:00:01 |
```

```
|   1 |  TABLE ACCESS BY INDEX ROWID| T             |     1 |    79 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | IDX_OBJECT_ID |     1 |       |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------
统计信息
---------
          0  recursive calls
          0  db block gets
          4  consistent gets
```

脚本 12-24　索引扫描

最后看看 rowid 扫描方式：

```
select object_name from t where object_id=2 and rowid='AAAdewAAEAAAAUMAAw';

OBJECT_NAME
-------
C_OBJ#

执行计划
--------
Plan hash value: 3207308387
----------------------------
| Id  | Operation                   | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |     1 |    91 |     1   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY USER ROWID| T    |     1 |    91 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
          1  consistent gets
```

脚本 12-25　rowid 扫描

全表扫描 consistent gets 为 1724，索引访问 consistent gets 为 4，而 rowid 访问的 consistent gets 为 1，三者的性能差别巨大。TABLE ACCESS BY USER ROWID 确实是最高效的访问方式，性能最高！

这里要特别注意使用 rowid 访问的应用场景。举个例子，我们通过索引找到了某条记录，然后要进行更新，如果不用 rowid 一般是 update tab set ...where 索引条件，如果刚才在查询的过程中顺带获取到 rowid，语句则改变成 update tab set ...where 索引条件和 rowid=xxx 条件，有了这个 rowid=xxx 条件，更新性能将进一步提升。

不过值得一提的是，一般情况下 rowid 是不会变化的。当然，如果做了 alter table move 的动作，rowid 显然就改变了。此外分区表的分区条件内的数据发生了分区转移（比如以地区为分区，厦门的数据转移到厦门来），rowid 也会产生变化。

12.2 设法避免外因影响

12.2.1 Hint 改写确保执行计划正确

我们在前面的章节中描述过关于 Hint 改写的内容，主要从 Hint、固定 Outline、重新收集统计信息三个方面来左右执行计划，从而让不正确的执行计划变得正确。这里的 Hint 部分也算是对 SQL 进行特定的等价改写。

具体的写法不再累述，略去。

12.2.2 避免子查询的错误执行计划

```
select ta.tch_id, ta.flow_id
          from tache pr,
               (select TCH_ID, c.flow_id
                  from event_q a, staff_event b, tache c
                 where (a.event_type = '2' OR a.event_type = '1')
                   and a.event_id = b.event_id
                   and (a.flag is null or a.flag = '1')
                   and b.staff_id = 1
                   and a.content_id = c.tch_id) ta
         where ta.flow_id = pr.sub_flow_id(+)
           and pr.flow_id is null
```

```
101 |  31 |          NESTED LOOPS                  |                    |     1 |    53 |    63   (2)|
102 |  32 |           MERGE JOIN CARTESIAN         |                    |     1 |    34 |    62   (2)|
103 |* 33 |            FILTER                      |                    |       |       |            |
104 |* 34 |             HASH JOIN RIGHT OUTER      |                    |     1 |    23 |    57   (2)|
105 |* 35 |              TABLE ACCESS FULL         | TACHE              |    10 |    90 |    28   (0)|
106 |  36 |              TABLE ACCESS FULL         | TACHE              |  7582 |  103K|    28   (0)|
107 |  37 |            BUFFER SORT                 |                    |    95 |  1045 |    34   (3)|
108 |* 38 |             TABLE ACCESS FULL          | STAFF_EVENT        |    95 |  1045 |     5   (0)|
109 |* 39 |           TABLE ACCESS BY INDEX ROWID  | EVENT_Q            |     1 |    19 |     1   (0)|
110 |* 40 |            INDEX UNIQUE SCAN           | IDX_EVENT_Q_EVENTID |     1 |       |     0   (0)|
111 ---------------------------------------------------------------------------------------------------
```

原因看出来了，因为结果集 ta 里的 tache 表和外围的 tache 表先关联，但是之间却没有关联条件，从而导致出现了笛卡尔乘积，所以性能大幅度下降。

因此我们要避免结果集 ta 的 tache 表和外围的 tache 表先关联，就是说让结果集 ta 内部的三个表先完成关联，再往外关联。那如何实现呢？

改写如下：

```
select ta.tch_id, ta.flow_id
          from tache pr,
               (select TCH_ID, c.flow_id,rownum
                  from event_q a, staff_event b, tache c
                 where (a.event_type = '2' OR a.event_type = '1')
                   and a.event_id = b.event_id
                   and (a.flag is null or a.flag = '1')
                   and b.staff_id = 1
```

```
            and a.content_id = c.tch_id) ta
    where ta.flow_id = pr.sub_flow_id(+)
      and pr.flow_id is null  ;
```

原理：为保证 rownum 值的准确性，保证该子查询不会移到外面去拿部分表和 tache 表先关联。

12.2.3　所在环境的资源不足等问题

这个问题主要表现为三种可能：1. 主要是体现在机器的配置比较低，内存 CPU 及 IO 的资源不够；2.外部某应用程序耗尽了主机的资源，导致主机资源不足；3. 内部程序大量使用并行操作，资源争用导致所在的环境资源不足。

12.3　本章习题、总结与延伸

习题 1：说说减少访问路径的普通 SQL 改写思路有哪些。

习题 2：你能再举出一个书中没有提及的减少访问路径的方法吗?

习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

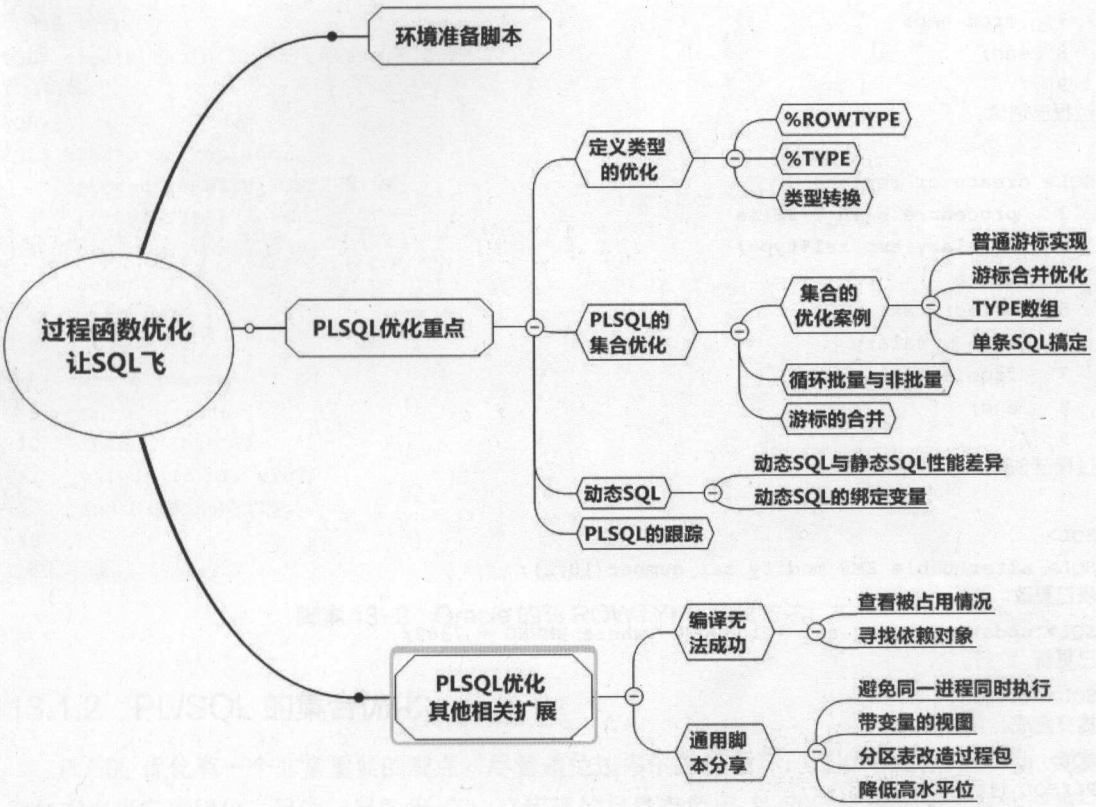
第 13 章　动手，过程函数优化让 SQL 飞

批量与否效果不言而喻

先抛出一个搬砖的故事来提问：建筑工人 A 将砖头一块一块地从某地运输到目的地。而建筑工人 B 则用板车装上砖头来运输，请问谁的效率高？答案是：当砖头特别多的时候，肯定是后者效率高。

搬砖其实就反映了 PL/SQL 代码的优化思路。在 PL/SQL 中一般都是基于游标循环的各种写法，就是建筑工人 A 在干活。而接下来我要告诉大家一个集合的概念，如果能用集合的写法来完成 PL/SQL 的优化改造，优化效果会好得多！这就是建筑工人 B 在干活。

集合的优化是本章的重点内容。同时还将介绍定义类型的优化、动态 SQL 的优化和一些其他相关扩展案例。本章总体学习思路如下图所示：



13.1　PL/SQL 优化重点

13.1.1　定义类型的优化

1. %TYPE

为了使某变量的数据类型与另一个已定义变量（尤其是针对某表某列的类型）的数据类型一致，Oracle 提供了%TYPE 的定义方法。这个新定义的变量的数据类型会自动跟随已定义的变量类型变化而变化，从而减少代码的修改。如下：

```
SQL> drop table   emp cascade constraints purge;
表已删除。
SQL> create table emp as select * from scott.emp;
表已创建。
SQL> create or replace
  2   procedure WITHOUT_TYPE is
  3   v_salary number(7,2);
  4   begin
  5   select max(sal)
  6   into v_salary
```

```
  7   from emp;
  8  end;
  9  /
过程已创建。

SQL> create or replace
  2   procedure WITH_TYPE is
  3   v_salary emp.sal%type;
  4   begin
  5   select max(sal)
  6   into v_salary
  7   from emp;
  8   end;
  9  /
过程已创建。

SQL>
SQL> alter table EMP modify sal number(10,2);
表已更改。
SQL> update EMP set sal = 1000000  where EMPNO = 7369;
已更新 1 行。
SQL> commit;
提交完成。
SQL> exec WITH_TYPE;
PL/SQL 过程已成功完成。
SQL> exec WITHOUT_TYPE;
BEGIN WITHOUT_TYPE; END;
*
第 1 行出现错误:
ORA-06502: PL/SQL: 数字或值错误 :  数值精度太高
ORA-06512: 在 "LJB.WITHOUT_TYPE", line 4
ORA-06512: 在 line 1
```

脚本 13-1　Oracle 的%TYPE 定义方法

2. %ROWTYPE

现实中我们还可以使用%ROWTYPE 来定义一个表示表中一行记录的变量，这比分别使用%TYPE 来定义表中各个列的变量要简洁得多，并且不容易遗漏、出错。这样也能增加程序的可维护性。具体如下试验所示：

```
SQL> drop table T purge;
表已删除。
SQL> drop table T1 purge;
表已删除。
SQL> create table T (
  2   c1 number,
  3   c2 number );
表已创建。
```

```
SQL> insert into T values (1,2);
已创建 1 行。
SQL> create table T1 as select * from T;
表已创建。
SQL>
SQL> create or replace
  2   procedure WITH_ROWTYPE is
  3   r T%ROWTYPE;
  4   begin
  5   select *
  6   into r
  7   from T
  8   where rownum = 1;
  9
 10   insert into T1
 11   values (r.c1, r.c2);
 12   end WITH_ROWTYPE;
 13  /
过程已创建。
```

脚本 13-2　Oracle 的% ROWTYPE 定义方法

13.1.2　PL/SQL 的集合优化

PL/SQL 优化有一个非常重要的观点：尽量避免过多的游标循环，可以考虑合并游标，用集合的观点来完成优化，另外就是能用 SQL 来实现的尽量避免用 PL/SQL 代码实现。具体请看一个需求：某领导想了解自己公司的各部门员工的收入情况，看看收入高于或者低于平均收入20%以上的人有哪些，同时也想看看谁是部门收入最低的员工。该如何实现呢?

这个例子听起来似乎不是很复杂，但却由此打造出一个优化经典案例。首先我们进行环境搭建准备，具体如下：

```
ALTER table EMP  add constraint EMP_PK primary key (EMPNO);
ALTER table DEP  add constraint DEP_PK primary key (DEPNO);

insert into EMP
 select rownum,
 'Name'||rownum,
 sysdate+rownum/100,
 dbms_random.value(1000,10000),--这里范围差异太小误差 20%就不容易统计出来
 dbms_random.value(1,50) ---此处和 DEP 的 depno 匹配
 from dual
 connect by level <= 5000;

insert into DEP
 select rownum, 'DEP'||rownum
 from dual
 connect by level <= 50;
 ----此处和部门号范围匹配
```

```
create table EMP_STAT (
 ENAME VARCHAR2(20),
 HIREDATE DATE,
 SAL NUMBER(7,2),
 DNAME VARCHAR2(20),
 MIN_SAL VARCHAR2(1) );
 MIN_SAL VARCHAR2(1) );
```

脚本 13-3　PL/SQL 集合优化的环境构造

1. 不断改进算法

（1）初始写法__游标循环每次取值

初始写法是最糟糕的写法，游标里每选一个员工就分别做一次平均值和最小值的判断，性能低下。具体如下：

```
create or replace procedure proc1 is
  v_avg_dep_sal emp.sal%type;
  v_min_dep_sal emp.sal%type;
  v_dname       dep.dname%type;
begin
  --确定部门员工的平均工资
  for i in (select empno, ename, depno, sal, hiredate from emp) loop
    select avg(sal) into v_avg_dep_sal from emp where depno = i.depno;
    --将平均工资与 emp 表中的员工进行比较，找到差异在 20%以上的记录的部门号，同时找出最低工资
    if abs(i.sal - v_avg_dep_sal) / v_avg_dep_sal > 0.20 then
      select dep.dname, min(emp.sal)
        into v_dname, v_min_dep_sal
        from dep, emp
       where dep.depno = i.depno
         and emp.depno = dep.depno
       group by dname;
      --假如最低工资找到了，就插入 EMP_STAT 表中， min_sal 表示为 Y
      if v_min_dep_sal = i.sal then
        insert into emp_stat
        values
          (i.ename, i.hiredate, i.sal, v_dname, 'Y');
        --差异在 20%的不是最低工资的，也插入 EMP_STAT 表中，只是 min_sal 表示为 N
      else
        insert into emp_stat
        values
          (i.ename, i.hiredate, i.sal, v_dname, 'N');
      end if;
    end if;
  end loop;
  commit;
end proc1;
/
```

脚本 13-4　初始写法__游标循环每次取值

（2）首次优化__游标循环合并取值

此次优化效果会比初始写法好一点，游标里每选一个员工就做一次平均值和最小值的判断，此次是合并在一起将平均值和最小值一并取了，虽然性能还是低下，但略有提升。具体代码如下：

```
create or replace procedure proc2 is
  v_avg_dep_sal emp.sal%type;
  v_min_dep_sal emp.sal%type;
  v_dname       dep.dname%type;
begin
  for i in (select empno, ename, depno, sal, hiredate from emp) loop
    --在获取到平均工资时还同时读取部门名和最低工资，这里合并做事，比之前的性能要高一些。
    select avg(emp.sal), min(emp.sal), dep.dname
      into v_avg_dep_sal, v_min_dep_sal, v_dname
      from dep, emp
     where dep.depno = i.depno
       and emp.depno = dep.depno
     group by dname;
    if abs(i.sal - v_avg_dep_sal) / v_avg_dep_sal > 0.20 then
      if v_min_dep_sal = i.sal then
        insert into emp_stat
        values
          (i.ename, i.hiredate, i.sal, v_dname, 'Y');
      else
        insert into emp_stat
        values
          (i.ename, i.hiredate, i.sal, v_dname, 'N');
      end if;
    end if;
  end loop;
  commit;
end proc2;
/
/
```

脚本 13-5　首次优化__游标循环合并取值

（3）二次优化__在内存中多次调用

前面由于员工的平均工资被一直反复计算，很没必要，因为员工的平均工资和最小值在这个场景或者是特定的时段中通常是不变的。这时新的优化方法就横空出世了，其目标是把这个值放到内存中去（类似 C 语言的结构体，定义成数组），避免反复运算，这样就提升了性能。前面性能低下的主要原因就是因为反复计算。具体看优化方法如下：

```
create or replace procedure proc3 is
  --将平均工资、最低工资、部门名用 type 来定义成数组保存在内存表（PL/SQL 表）中，避免被反复调用
  type dep_sal_details is record(
```

```
    avg_dep_sal emp.sal%type,
    min_dep_sal emp.sal%type,
    max_dep_sal emp.sal%type,
    dname       dep.dname%type);
  type dep_sals is table of dep_sal_details index by binary_integer;
  v_dep_sal dep_sals;

begin
  --将部门信息加入内存表（PL/SQL 表）中
  for i in (select avg(sal) asal, min(sal) min_sal, dname, dep.depno
              from dep, emp
             where emp.depno = dep.depno
             group by dname, dep.depno) loop
    v_dep_sal(i.depno).avg_dep_sal := i.asal;
    v_dep_sal(i.depno).min_dep_sal := i.min_sal;
    v_dep_sal(i.depno).dname := i.dname;
  end loop;
  ---接下来无须访问 dep 表，从 PL/SQL 表 v_dep_sal 中得到数据，如下：
  for each_emp in (select empno, ename, depno, sal, hiredate from emp) loop
    if abs(each_emp.sal - v_dep_sal(each_emp.depno).avg_dep_sal) / v_dep_sal
(each_emp.depno)
      .avg_dep_sal > 0.20 then
      if v_dep_sal(each_emp.depno).min_dep_sal = each_emp.sal then
        insert into emp_stat
        values
          (each_emp.ename,
           each_emp.hiredate,
           each_emp.sal,
           v_dep_sal(each_emp.depno).dname,
           'Y');
      else
        insert into emp_stat
        values
          (each_emp.ename,
           each_emp.hiredate,
           each_emp.sal,
           v_dep_sal(each_emp.depno).dname,
           'N');
      end if;
    end if;
  end loop;
  commit;
end proc3;
/
```

脚本 13-6　二次优化__在内存中多次调用

（4）三次优化__分析函数单条搞定

到此为止是不是已经优化得很好了，其实最好的方法是尝试不用 PL/SQL 方案实现代码，尽量用 SQL 来完成。经过仔细分析，我们发现，其实这个需求是可以直接用 SQL 来实现的。接下来我们看看神奇的分析函数，用一条 SQL 语句就能轻松搞定，写法如下：

```
create or replace procedure proc4 is
begin
  insert into emp_stat
    select e.empno,
           e.hiredate,
           e.sal,
           dep.dname,
           case when sal > min_sal then'Y' else 'N' end case
      from (select empno,
                   hiredate,
                   sal,
                   depno,
                   avg(sal) over(partition by depno) as avg_sal,
                   min(sal) over(partition by depno) as min_sal
              from emp) e,
           dep
     where e.depno = dep.depno
       and abs(e.sal - e.avg_sal) / e.avg_sal > 0.20;
  commit;
end proc4;
/
```

脚本 13-7　三次优化__分析函数单条搞定

2. 算法改进效果

接下来我们测试一下代码不断改进后的运行效果，分别执行 proc1～proc4，对比一下在有 50 个部门 5000 名员工的情况下，性能差异如何，具体如下：

```
SQL> set timing on
SQL> exec proc1;

PL/SQL 过程已成功完成。

已用时间:  00: 00: 02.24
SQL> exec proc2;

PL/SQL 过程已成功完成。

已用时间:  00: 00: 01.79
SQL> exec proc3;
```

```
PL/SQL 过程已成功完成。

已用时间:  00: 00: 00.12
SQL> exec proc4;

PL/SQL 过程已成功完成。

已用时间:  00: 00: 00.04
```

脚本 13-8　第 1 次测试（50 部门，5000 名员工）

执行时间分别是 2.24s、1.79s、0.12s 和 0.04s，差异非常明显。如果我们增加部门数和员工数，差异将会更大，我们将 dbms_random.value(1,50)改为 (1,500)，将第一处的 connect by level <= 5000 改为 level <= 50000；将第二处的 connect by level <= 50 改为 level <= 500，完成部门 50～500 个和员工 5000～50000 名的数据改变，具体如下：

```
insert into EMP
 select rownum,
 'Name'||rownum,
 sysdate+rownum/100,
 dbms_random.value(1000,10000),--这里范围差异太小 20%之差就不容易统计出来
 dbms_random.value(1,500) ---此处和 DEP 的 depno 匹配
 from dual
 connect by level <= 50000;

insert into DEP
 select rownum, 'DEP'||rownum
 from dual
 connect by level <= 500;
 ----此处和部门号范围匹配
```

脚本 13-9　第 2 次测试（500 部门，50000 名员工）

继续执行 proc1～proc4，此次测试的时间分别是：2 分 38 秒、1 分 37 秒、1.25 秒和 0.24 秒。

接下来将数据增加到 5000 个部门 50 万名员工，测试的时间分别是 1 小时 10 分钟、50 分钟、5 秒和 1 秒。

限于篇幅，具体的测试过程这里就不再列出，通过这些数据很容易看出，随着数据量的增加，proc1 和 proc2 将越跑越慢，而 proc3 和 proc4 则影响很小，其中 proc4 则更为强劲。这就是代码设计中的可扩展性。请看下表：

PL/SQL 的一次经典优化						
需　求	优化过程	手　段	经典代码部分截取	性能情况（50 个部门 5000 名员工）	性能情况（500 个部门 50000 名员工）	性能情况（5000 个部门 500000 名员工）
统计一个部门中高于或者低于平均水平 20%的员工，并标识出最低薪水员工	首次编写	每选一个员工就分别做一次平均值和最小值的判断	for i in (select empno, ename, depno, sal, hiredate from emp) loop 　select avg(sal) into v_avg_dep_sal from emp where depno = i.depno;	2.24 秒	2 分 38 秒	1 小时 10 分钟
统计一个部门中高于或者低于平均水平 20%的员工，并标识出最低薪水员工	第 1 次优化	游标里每选一个员工就做一次平均值和最小值的判断，此次是合并在一起将平均值和最小值一并取了	for each_emp in c_emp_list loop select avg(emp.sal), **min (emp.sal)**, dept.dname into v_avg_dept_sal, v_min_ dept_sal,v_dname from dept, emp where dept.deptno = each_ emp.deptno and emp.deptno = dept.deptno group by dname;	1.79 秒	1 分 37 秒	50 分钟
	第 2 次优化	员工的平均工资被一直反复计算，现在把这个值放到内存中去	type dept_sal_details is **record (avg_dept_sal emp.sal%type,** **min_dept_sal emp.sal%type,** **dname dept.dname%type);** type dept_sals is table of dept_sal_details index by binary_integer; v_dept_sal dept_sals;	0.12 秒	1.25 秒	5 秒
	第 3 次优化	应用分析函数，转化成一条 SQL 语句	select empno, hiredate, sal, deptno, avg(sal) over (partition by deptno) as avg_sal, min(sal) over (partition by deptno) as min_sal from emp	0.04 秒	0.24 秒	1 秒

13.1.3　PL/SQL 的游标合并

关于游标合并，又是一种经典的优化思路，其实在前面的案例中的首次优化，就是这种思路。现在我们再举一个例子来说明，首先是环境准备，具体如下：

```
drop table t1 cascade constraints purge;
drop table t2 cascade constraints purge;
drop table t3 cascade constraints purge;
set linesize 1000
set serveroutput on size 10000

create table t1
( a int primary key, y char(80) );

create table t2
( b int primary key, a references t1, y char(80) );

create index t2_a_idx on t2(a);

create table t3
( c int primary key, b references t2, y char(80) );

create index t3_b_idx on t3(b);

insert into t1
select rownum, 'x'
  from all_objects
 where rownum <= 10000;

insert into t2
select rownum, mod(rownum,10000)+1, 'x'
  from all_objects
 where rownum <= 50000;

insert into t3
select rownum, mod(rownum,50000)+1, 'x'
  from all_objects;
```

脚本 13-10　游标合并试验前的环境准备

接下来看未合并游标的写法 1，如下：

```
Create or replace procedure proc_cur1 as
begin
    for i in 1 .. 100000
    loop
        for a in ( select t1.a, t1.y
                     from t1 where t1.a = i )
        loop
            for b in ( select t2.b, t2.a, t2.y
                         from t2 where t2.a = a.a )
            loop
                for c in ( select t3.c, t3.b, t3.y
                             from t3 where t3.b = b.b )
                loop
```

```
                    null;
                end loop;
            end loop;
        end loop;
    end loop;
end;
/
```

脚本 13-11　游标未合并前的写法

上述写法有 3 个游标，显然不合理，可以考虑合并优化，将 3 个游标合并成 1 个，改成写法 2，如下：

```
Create or replace procedure proc_cur2 as
begin
    for i in 1 .. 100000
    loop
        for x in ( select t1.a t1a, t1.y t1y,
                          t2.b t2b, t2.a t2a, t2.y t2y,
                          t3.c t3c, t3.b t3b, t3.y t3y
                     from t1, t2, t3
                    where t1.a = i
                      and t2.a (+) = t1.a
                      and t3.b (+) = t2.b )
        loop
            null;
        end loop;
    end loop;
end;
/
```

脚本 13-12　游标合并改造的写法

我们测试一下两种写法的性能，看看执行时间的差异，如下：

```
SQL> set timing on
SQL> exec proc_cur1;
PL/SQL 过程已成功完成。
已用时间:  00: 00: 01.91
SQL> exec proc_cur2;
PL/SQL 过程已成功完成。
已用时间:  00: 00: 01.25
```

脚本 13-13　游标合并与否写法的性能差异

可以看出写法 2 比写法 1 性能更高，实际情况还不止是执行时间上的差异，在占用各种内存资源方面，写法 2 也明显胜出。

13.1.4 动态 SQL

1. 动态 SQL 与绑定变量关系

一般来说，动态 SQL 的绑定变量需要 using 关键字，而静态 SQL 是自动绑定的。如果用动态 SQL 需要小心。

```
set linesize 1000
set serveroutput on size 100000

create or replace function get_value_dyn_bind
           ( p_empno in number, p_cname in varchar2 ) return varchar2
as
    l_value  varchar2(4000);
begin
    execute immediate
    'select ' || p_cname || ' from emp where empno = :x'
    into l_value
    using p_empno;

    return l_value;
end get_value_dyn_bind;
/

create or replace function get_value_dyn_nobind
           ( p_empno in number, p_cname in varchar2 ) return varchar2
as
    l_value  varchar2(4000);
begin
    execute immediate
    'select ' || p_cname || ' from emp where empno = '||p_empno
    into l_value;

    return l_value;
end get_value_dyn_nobind;
/

set linesize 1000
exec runstats_pkg.rs_start;


declare
       l_dummy varchar2(30);
begin
    for i in 1 .. 100
    loop
        for x in ( select empno from emp )
        loop
            l_dummy := get_value_dyn_bind(x.empno, 'ENAME' );
```

```
            l_dummy := get_value_dyn_bind(x.empno, 'EMPNO' );
            l_dummy := get_value_dyn_bind(x.empno, 'HIREDATE' );
        end loop;
    end loop;
end;
/
exec runstats_pkg.rs_middle


declare
        l_dummy varchar2(30);
begin
    for i in 1 .. 100
    loop
        for x in ( select empno from emp )
        loop
            l_dummy := get_value_dyn_nobind(x.empno, 'ENAME' );
            l_dummy := get_value_dyn_nobind(x.empno, 'EMPNO' );
            l_dummy := get_value_dyn_nobind(x.empno, 'HIREDATE' );
        end loop;
    end loop;
end;
/

set linesize 1000
exec runstats_pkg.rs_stop(500);



SQL> exec runstats_pkg.rs_stop(500);
Run1 ran in 223hsec
Run2 ran in 532hsec
run 1 ran in 41.92% of the time

Name                                  Run1        Run2          Diff
STAT...enqueue requests                  6       1,501         1,495
STAT...CCursor + sql area evicted        5       1,500         1,495
STAT...sql area evicted                  5       1,500         1,495
STAT...enqueue releases                  6       1,501         1,495
STAT...parse count (hard)                6       1,501         1,495
STAT...cursor authentications            0       1,501         1,501
LATCH.enqueues                          90       3,082         2,992
LATCH.enqueue hash chains               92       3,110         3,018
LATCH.kks stats                         18       6,778         6,760
LATCH.row cache objects                277      43,652        43,375
LATCH.shared pool                  150,336     207,620        57,284
LATCH.shared pool simulator        100,091     167,891        67,800
STAT...parse count (total)              10     150,006       149,996
STAT...session cursor cache hits   150,098         100      -149,998
```

```
STAT...recursive calls                  150,620       452,106       301,486
STAT...session pga memory               393,216             0      -393,216

Run1 latches total versus runs ----difference and pct
Run1       Run2       Diff       Pct
502,230    684,958    182,728   73.32%
```

脚本 13-14　动态 SQL 绑定变量与否的性能差异

从这个例子我们可以看出，get_value_dyn_bind 的写法由于用到了绑定变量（using p_empno），性能比未用绑定变量的 get_value_dyn_nobind 要好得多。

2. 动态与静态 SQL 性能差异

一般来说，动态 SQL 的性能要差于静态 SQL，因为动态 SQL 在执行时编译，而静态 SQL 是随过程一起编译的，从这个角度来说，动态 SQL 的性能会更差。除非表和列是变量，无法形成一个完全的 SQL，这时才考虑使用动态 SQL。本案例比较典型，虽然列不存在，但是是固定的 ENAME、EMPNO、HIREDATE 三个之一，这时候可以考虑转化成静态 SQL，提升性能。

```
drop table emp cascade constraints  purge;
create table emp as select * from scott.emp;
set linesize 1000
set serveroutput on size 100000

create or replace function get_value_dyn
            ( p_empno in number, p_cname in varchar2 ) return varchar2
as
    l_value  varchar2(4000);
begin
    execute immediate
    'select ' || p_cname || ' from emp where empno = :x'
    into l_value
    using p_empno;

    return l_value;
end;
/


create or replace function get_value_static( p_empno in number, p_cname in varchar2 )
return varchar2
as
    l_value  varchar2(4000);
begin
    select decode( upper(p_cname),
                   'ENAME', ename,
                   'EMPNO', empno,
                   'HIREDATE', to_char(hiredate,'yyyymmddhh24miss'))
      into l_value
```

```
      from emp
     where empno = p_empno;

    return l_value;
end;
/

set linesize 1000
set serveroutput on size 100000

exec runstats_pkg.rs_start;


declare
        l_dummy varchar2(30);
begin
    for i in 1 .. 5000
    loop
        for x in ( select empno from emp )
        loop
            l_dummy := get_value_dyn(x.empno, 'ENAME' );
            l_dummy := get_value_dyn(x.empno, 'EMPNO' );
            l_dummy := get_value_dyn(x.empno, 'HIREDATE' );
        end loop;
    end loop;
end;
/
exec runstats_pkg.rs_middle


declare
        l_dummy varchar2(30);
begin
    for i in 1 .. 5000
    loop
        for x in ( select empno from emp )
        loop
            l_dummy := get_value_static(x.empno, 'ENAME' );
            l_dummy := get_value_static(x.empno, 'EMPNO' );
            l_dummy := get_value_static(x.empno, 'HIREDATE' );
        end loop;
    end loop;
end;
/


exec runstats_pkg.rs_stop(500);

Run1 ran in 359hsec
```

```
Run2 ran in 1658hsec
run 1 ran in 21.65% of the time

Name                                                    Run1        Run2        Diff
LATCH.shared pool                                     185,670     180,308      -5,362
STAT...physical read total bytes                       49,152           0     -49,152
STAT...physical read bytes                             49,152           0     -49,152
STAT...session uga memory max                         123,452      65,464     -57,988
LATCH.shared pool simulator                            65,237       5,163     -60,074
STAT...session uga memory                              65,464           0     -65,464
STAT...session pga memory max                         131,072      65,536     -65,536

Run1 latches total versus runs ----difference and pct
Run1        Run2        Diff       Pct
807,739     742,416     -65,323 108.80%

PL/SQL 过程已成功完成。
```

脚本 13-15　动态与静态 SQL 的性能差异

从这个例子我们可以看出，动态 SQL get_value_dyn 的写法性能比静态 SQL get_value_static 的写法要差。

13.1.5　使用 10046 trace 跟踪 PL/SQL

```
set linesize 266
set timing on
set pagesize 5000
alter session set events '10046 trace name context  forever,level 12';

---此处执行你的存储过程、包等。

alter session set events '10046 trace name context off';

select d.value
|| '/'
|| LOWER (RTRIM(i.INSTANCE, CHR(0)))
|| '_ora_'
|| p.spid
|| '.trc' trace_file_name
from (select p.spid
      from v$mystat m,v$session s, v$process p
      where  m.statistic#=1 and s.sid=m.sid and p.addr=s.paddr) p,
      (select t.INSTANCE
       FROM v$thread t,v$parameter v
       WHERE v.name='thread'
       AND(v.VALUE=0 OR t.thread#=to_number(v.value))) i,
       (select value
       from v$parameter
```

```
      where name='user_dump_dest') d;

tkprof d:\app\liangjb\diag\rdbms\test11g\test11g\trace/test11g_ora_7668.trc    d:\10046.txt
sys=no sort=prsela,exeela,fchela
```

脚本 13-16　使用 10046 trace 跟踪 PL/SQL

13.2　PL/SQL 优化其他相关扩展

13.2.1　编译无法成功

很多时候我们在更新代码、编译过程包时，由于过程包正在运行中，会导致无法编译成功，这时候我们需要找到程序对应的 SID，杀了这个进程，方可编译成功，具体如下列试验脚本：

```
CREATE or replace PROCEDURE my_proc as
BEGIN
 DBMS_LOCK.sleep (300);
END;
/
--------------------------------------------------------------------------------
CREATE OR REPLACE
FUNCTION my_func
RETURN varchar2
As
Begin
  DBMS_LOCK.sleep (300);
 Return 'HELLO, WORLD';
END;
/

select my_func from dual;
--------------------------------------------------------------------------------
CREATE OR REPLACE PACKAGE my_pkg as
 Procedure my_proc;
 function my_func RETURN varchar2;
End;
/

CREATE OR REPLACE
PACKAGE BODY my_pkg as
Procedure my_proc is
 BEGIN
 NULL;
 END my_proc;

FUNCTION my_func
```

```
RETURN varchar2 As
 Begin
 Return 'HELLO, WORLD';
 END my_func;

 End my_pkg;
/
exec my_pkg.my_proc ;
select my_pkg.my_func from dual;
```

找到对应的进程，杀了后，编译即可成功（下面例子中假如找到的 sid 为 126）！

```
思路:
select * from v$access where owner='LJB' and object='MY_PROC' ----126
SELECT * FROM V$SESSION WHERE SID=126
ALTER SYSTEM KILL SESSION '126,1967' immediate;
```

脚本 13-17　PL/SQL 编译无法成功时的处理方法

13.2.2　通用脚本分享

这一节笔者想将一些常见的经典脚本拿出来和读者一起分享，主要想告诉读者我们可以利用 PL/SQL 写出一些工具来给工作带来更大的方便，比如提这样一个需求吧：有一系列普通表都有几十到几百 GB 这么大，数据从几亿到几十亿，现在想将这些表改造成分区表，用其中的时间或者其他字段来做分区，允许有一段停机时间来停这些表相关的应用，该如何做呢？

普通思路是这样的：新建一张分区表，按日期建分区，确保分区表各字段和属性都和普通表一样。然后停应用，将普通表记录插入到分区表中。然后将普通表重命名，分区表命名成原表的名字，完成任务。

这个思路看似很正常，却存在不少问题：

- 首先是命名要注意，新表和旧表的索引约束命名不能冲突，最终新表的表名要和旧表一样，这里要有手工修改的过程。
- 旧表往新表插入数据，虽然可以用并行等方式，但是脚本需要预先写好，而且速度显然不如建新表的速度快。
- 新建表要变成分区表，并且是根据输入的分区字段完成的分区，该如何实现？
- 新建表除了分区语法外，还需要考虑对应的所有字段和属性和原表一致，包括表和列的中文注释，列是否允许为空，是否有主外键约束，是否有 check，是否有索引……

存在这么多问题，怎么办呢？这时要对原先的这个思路进行优化改进，从几个方面来考虑：

- **设想思路 1**。新旧表及对应列属性的各种重命名那么麻烦，能否考虑根据顺序完成，自动重命名？在过程包中依次执行，只要确保顺序对，就没问题。

```
procedure p_rename_001 (p_tab in varchar2)
  as
  /*
```

```
  功能：将原表重命名为_yyyymmdd 格式的表名
  完善点：  要考虑 RENMAE 的目标表已存在的情况，先做判断
 */
 V_CNT_RE_TAB  NUMBER(9) :=0;
 v_sql_p_rename          varchar2(4000);
 begin
 SELECT  COUNT(*)   INTO V_CNT_RE_TAB FROM user_objects where object_name=UPPER(P_
TAB||'_'||YYYYMMDD);
   if V_CNT_RE_TAB=0 then
         v_sql_p_rename:= 'rename '||P_TAB ||' to '||P_TAB||'_'||YYYYMMDD;
      --   DBMS_OUTPUT.PUT_LINE(v_sql_p_rename);--调试使用
         p_insert_log(p_tab,'P_RENAME',v_sql_p_rename,'完成原表的重命名，改为_YYYYMMDD
形式',1);
         execute immediate(v_sql_p_rename); --这里无须做判断，rename 动作真实完成！如果后
续只是为生成脚本而不是真实执行分区操作，最后再把这个表 RENAME 回去！
   ELSE
         RAISE_APPLICATION_ERROR(-20066,'备份表'||P_TAB||'_'||YYYYMMDD||'已存在，请先
删除或重命名该备份表后再继续执行！');
       --  DBMS_OUTPUT.PUT_LINE('备份表'||P_TAB||'_'||YYYYMMDD||'已存在');
   end if;
   DBMS_OUTPUT.PUT_LINE('操作步骤 1(备份原表)-------将'||p_tab ||' 表 RENMAE 成
'||p_tab||'_'||YYYYMMDD||'，并删除其约束索引等');
 end p_rename_001;
```

脚本 13-18　分区改造前的原表重命名

- **设想思路 2。**将 insert into 表直接变成 create 分区表的形式，然后将这个新建的表的所有列属性都用 alter 的模式加上，将分区表的设置也加上。然后配合并行，速度就可以大幅提升。关于表和列属性自动增加的方法见设想 1，这个 create table+parallel 的模式很容易在 PL/SQL 中通过拼 SQL 实现。

```
procedure p_ctas_002 (p_tab in varchar2,
                      p_struct_only  in number,
                      p_deal_flag in number,
                      p_part_colum in varchar2,
                      p_parallel in number default 4,
                      p_tablespace IN VARCHAR2)
 as
  /*
  功能：用 CREATE TABLE AS SELECT 的方式从 RENAME 的_yyyymmdd 表中新建出一个只有 MAXVALUE 的初
步分区表
  完善点：要考虑并行，nologging 的提速方式，也要考虑最终将 NOLOGGING 和 PARALLEL 恢复成正常状态
 */
 v_sql_p_ctas          varchar2(4000);
 begin
     v_sql_p_ctas:='create table '||p_tab
     ||' partition by range (.'||p_part_colum||' ) ('
     || ' partition P_MAX  values less than (maxvalue))'||
     ' nologging parallel 4  tablespace '||p_tablespace||
```

```
      ' as select /*+parallel(t,'||p_parallel||')*/ *'||
      ' from '|| P_TAB||'_'||YYYYMMDD ;
     if p_struct_only=0 then
       v_sql_p_ctas:=v_sql_p_ctas ||' where 1=2';
     else
       v_sql_p_ctas:=v_sql_p_ctas ||' where 1=1';
       end if;
      --DBMS_OUTPUT.PUT_LINE(v_sql_p_ctas);--调试使用
      p_insert_log(p_tab,'p_ctas',v_sql_p_ctas,'完成 CTAS 建初步分区表',2,1);
      p_if_judge(v_sql_p_ctas,p_deal_flag);

      v_sql_p_ctas:='alter table '|| p_tab ||' logging';
      p_insert_log(p_tab,'p_ctas',v_sql_p_ctas,'将新分区表修改回 LOGGING 属性',2,2);
      p_if_judge(v_sql_p_ctas,p_deal_flag);

      v_sql_p_ctas:='alter table '|| p_tab || ' noparallel';
      p_insert_log(p_tab,'p_ctas',v_sql_p_ctas,'将新分区表修改回 NOPARALLEL 属性',2,3);
      p_if_judge(v_sql_p_ctas,p_deal_flag);
   DBMS_OUTPUT.PUT_LINE('操作步骤 2(建分区表)-------通过 CTAS 的方式从 '||p_tab||'_'||YYYYMMDD||
' 中新建'||p_tab ||'表，完成初步分区改造工作');
  end p_ctas_002;
```

脚本 13-19　用 CTAS 完成一张仅带 MAX 分区的新表

- **设想思路 3。**由于之前 create 表是将表改造成一个默认仅有一个最大 MAX 分区的分区表，接下来根据所需要的分区数来 split，如下：

```
procedure p_split_part_003 (p_tab in varchar2,
                            p_deal_flag in number,
                            p_part_nums in number default 24,
                            p_tab_tablespace IN VARCHAR2)
 as
 /*
  功能：用 CREATE TABLE AS SELECT 的方式新建出一个只有 MAXVALUE 的初步分区表进行 SPLIT，
  按月份进行切分，默认 p_part_nums 产生 24 个分区，构造 2 年的分区表，第一个分区为当前月的
  上一个月
 */
 v_first_day   date;
 v_next_day    date;
 v_prev_day    date;
 v_sql_p_split_part         varchar2(4000);
 begin
  select to_date(to_char(sysdate, 'yyyymm') || '01', 'yyyymmdd')
        into v_first_day
        from dual;
   for i in 1 .. p_part_nums loop
      select add_months(v_first_day, i) into v_next_day from dual;
      select add_months(v_next_day, -1) into v_prev_day from dual;
      v_sql_p_split_part := 'alter table '||p_tab||' split partition p_MAX at ' ||
             '(to_date(''' || to_char(v_next_day, 'yyyymmdd') ||
```

```
            ''',''yyyymmdd''))' || 'into (partition PART_' ||
            to_char(v_prev_day, 'yyyymm') || ' tablespace '|| p_tab_tablespace||',
partition p_MAX)';
     -- DBMS_OUTPUT.PUT_LINE(v_sql_p_split_part);--调试使用
      p_insert_log(p_tab,'p_split_part',v_sql_p_split_part,'分区表完成分区 SPLIT 工作
',3,i);
      p_if_judge(v_sql_p_split_part,p_deal_flag);
   end loop;
 DBMS_OUTPUT.PUT_LINE('操作步骤 3(分区操作)-------对新建的'||p_tab ||'分区表完成分区 SPLIT
工作');
 end p_split_part_003;
```

脚本 13-20　把分区新表 split 成多个分区

- **设想思路 4**。能否避免建表的麻烦，比如通过输入一个命令自动完成新表的建立；让新表自动从旧表中找到列名和对应属性，找到约束关系，并且根据时间字段完成分区，最终高效建表成功？这看似很难，其实不难，秘诀就是根据数据字典获取到这些对象的属性。

请看代码 p_tab_comments_004，得到表的注释：

```
   procedure p_tab_comments_004  (p_tab in varchar2,p_deal_flag in number)
     as
     /*
      功能：从_YYYYMMDD 备份表中得到表和字段的注释，为新分区表的表名和字段增加注释
     */
     v_sql_p_tab_comments          varchar2(4000);
     v_cnt number;
     begin
      select count(*) into v_cnt from user_tab_comments where table_name=UPPER
(P_TAB)||'_'||YYYYMMDD AND COMMENTS IS NOT NULL;
      if v_cnt>0 then
       for i in (select * from user_tab_comments where table_name=UPPER(P_TAB)||
'_'||YYYYMMDD AND COMMENTS IS NOT NULL) loop
         v_sql_p_tab_comments:='comment on table '||p_tab||' is '|| ''''||i.COMMENTS||'''';
        -- DBMS_OUTPUT.PUT_LINE(v_sql_p_deal_tab_comments);--调试使用
         p_insert_log(p_tab,'p_deal_comments',v_sql_p_tab_comments,'将新分区表的表的注
释加上',4,1);
         p_if_judge(v_sql_p_tab_comments,p_deal_flag);
       end loop;
        DBMS_OUTPUT.PUT_LINE('操作步骤 4(表的注释)-------对'||p_tab ||'表增加表名的注释内
容');
       ELSE
        DBMS_OUTPUT.PUT_LINE('操作步骤 4(表的注释)-------'||UPPER(P_TAB)||'_'||YYYYMMDD
||'并没有表注释!');
       END IF;
     end p_tab_comments_004;
```

脚本 13-21　补进分区新表的注释

请看代码 p_col_comments_005，得到列的注释：

```
procedure p_col_comments_005  (p_tab in varchar2,p_deal_flag in number)
  as
  /*
   功能：从_YYYYMMDD 备份表中得到表和字段的注释，为新分区表的表名和字段增加注释
  */
  v_sql_p_col_comments          varchar2(4000);
  v_cnt number;
  begin
    select  count(*)  into  v_cnt  from  user_col_comments  where  table_name=UPPER
(P_TAB)||'_'||YYYYMMDD AND COMMENTS IS NOT NULL;
    if v_cnt>0 then
    for  i  in  (select  *  from  user_col_comments  where  table_name=UPPER(P_TAB)||'_'
||YYYYMMDD AND COMMENTS IS NOT NULL) loop
      v_sql_p_col_comments:='comment on column '||p_tab||'.'||i.COLUMN_NAME||' is '||
''''||i.COMMENTS||'''';
      p_insert_log(p_tab,'p_deal_col_comments',v_sql_p_col_comments,'将新分区表的列的注
释加上',5,1);
      p_if_judge(v_sql_p_col_comments,p_deal_flag);
    end loop;
     DBMS_OUTPUT.PUT_LINE('操作步骤 5(列的注释)-------对'||p_tab ||'表增加列名及字段的注释内
容');
    else
     DBMS_OUTPUT.PUT_LINE('操作步骤 5(列的注释)-------'||UPPER(P_TAB)||'_'||YYYYMMDD ||'
并没有列注释!');
    end if;
  end p_col_comments_005;
```

脚本 13-22　补进分区新表所有列的注释

请看代码 p_defau_and_null_006，得到表的列中是否为空的属性：

```
  procedure p_defau_and_null_006 (p_tab in varchar2,p_deal_flag in number)
  as
  /*
   功能：从_YYYYMMDD 备份表中得到原表的 DEFAULT 值，为新分区表的表名和字段增加 DEFAULT 值
  */
  v_sql_defau_and_null          varchar2(4000);
  v_cnt  number;
  begin
  select count(*) into v_cnt  from user_tab_columns where table_name=UPPER(P_TAB)||
'_'||YYYYMMDD and data_default is not null;
    if v_cnt>0 then
    for  i  in  (select  *  from  user_tab_columns  where  table_name=UPPER(P_TAB)||'_'
||YYYYMMDD and data_default is not null) loop
     v_sql_defau_and_null:='alter  table  '||p_tab||'  modify  '||i.COLUMN_NAME  ||'
default ' ||i.data_default;
     p_insert_log(p_tab,'p_deal_default',v_sql_defau_and_null,'将新分区表的默认值加上',6);
     p_if_judge(v_sql_defau_and_null,p_deal_flag);
    end loop;
```

```
    DBMS_OUTPUT.PUT_LINE('操作步骤 6(空和默认)-------对'||p_tab ||'表完成默认 DEFAULT 值的
增加');
    else
    DBMS_OUTPUT.PUT_LINE('操作步骤 6(空和默认)-------'||UPPER(P_TAB)||'_'||YYYYMMDD ||'
并没有 DEFAULT 或 NULL 值!');
    end if;

  end p_defau_and_null_006;
```

脚本 13-23　补进分区新表所有列的 null 属性

请看代码 p_check_007，得到列的 check 属性：

```
procedure p_check_007 (p_tab in varchar2,p_deal_flag in number)
  as
  /*
   功能：从_YYYYMMDD 备份表中得到原表的 CHECK 值，为新分区表增加 CHECK 值
   另注：
    user_constraints 已经进行了非空的判断，可以略去如下类似的从 user_tab_columns 获取非空判断的
代码编写来判断是否
    for i in (select * from user_tab_columns where table_name=UPPER(P_TAB)||'_'
||YYYYMMDD and nullable='N') loop
     v_sql:='alter table '||p_tab||' modify '||i.COLUMN_NAME ||' not null';
  */
   v_sql_p_check          varchar2(4000);
   v_cnt number;
  begin
  select count(*) into v_cnt from user_constraints where table_name=UPPER(P_TAB)||
'_'||YYYYMMDD and constraint_type='C';
   if v_cnt>0 then
   for i in (select * from user_constraints where table_name=UPPER(P_TAB)||'_'||
YYYYMMDD and constraint_type='C') loop
      v_sql_p_check :='alter table '||P_TAB||'_'||YYYYMMDD ||' drop constraint ' ||
I.CONSTRAINT_NAME;
      p_insert_log(p_tab,'p_deal_check',v_sql_p_check ,'将备份出来的原表的 CHECK 删除
',7,1);
      p_if_judge(v_sql_p_check ,p_deal_flag);
      v_sql_p_check :='alter table '||p_tab||' ADD CONSTRAINT '||I.CONSTRAINT_NAME||'
CHECK ('||I.SEARCH_CONDITION ||')' ;
      p_insert_log(p_tab,'p_deal_check',v_sql_p_check ,'将新分区表的 CHECK 加上',7,2);
      p_if_judge(v_sql_p_check ,p_deal_flag);
    end loop;
    DBMS_OUTPUT.PUT_LINE('操作步骤 7(check 约束)-------对'||p_tab ||'完成 CHECK 的约束');
    else
    DBMS_OUTPUT.PUT_LINE('操作步骤 7(check 约束)-----'||UPPER(P_TAB)||'_'||YYYYMMDD ||'
并没有 CHECK!');
    end if;

  end p_check_007;
```

脚本 13-24　补进分区新表所有列的 check 约束

请看代码 p_index_008，得到表的索引：

```
procedure p_index_008 (p_tab in varchar2,p_deal_flag in number,p_idx_tablespace IN
VARCHAR2)
  as
  /*
   功能：从_YYYYMMDD 备份表中得到原表的索引信息，为新分区表增加普通索引（唯一和非唯一索引，函数索引
暂不考虑），并删除旧表索引
   难点：需要考虑联合索引的情况
  */
   v_sql_p_normal_idx          varchar2(4000);
   v_cnt number;

  begin
     SELECT count(*) into v_cnt
      from user_indexes
      where table_name = UPPER(P_TAB)||'_'||YYYYMMDD
      and index_type='NORMAL' AND INDEX_NAME NOT IN (SELECT CONSTRAINT_NAME FROM
USER_CONSTRAINTS);
  if v_cnt>0 then
    for i in
    (
      WITH T AS
      (
      select C.*,I.UNIQUENESS
      from user_ind_columns C
      ,(SELECT DISTINCT index_name,UNIQUENESS
      from user_indexes
      where table_name = UPPER(P_TAB)||'_'||YYYYMMDD
      and index_type='NORMAL'
      AND INDEX_NAME NOT IN
      (SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS)
      ) i
      where c.index_name = i.index_name
      )
      SELECT INDEX_NAME,TABLE_NAME,UNIQUENESS, MAX(substr(sys_connect_by_path(COLUMN_
NAME, ','), 2)) str ---考虑组合索引的情况
      FROM (SELECT column_name,INDEX_NAME,TABLE_NAME, row_number() over(PARTITION BY
INDEX_NAME,TABLE_NAME ORDER BY COLUMN_NAME) rn
      ,UNIQUENESS
      FROM T) t
      START WITH rn = 1
      CONNECT BY rn = PRIOR rn + 1
      AND INDEX_NAME = PRIOR INDEX_NAME
      GROUP BY INDEX_NAME,T.TABLE_NAME,UNIQUENESS
) loop
      v_sql_p_normal_idx:= 'drop index '||i.index_name;
      p_insert_log(p_tab,'p_deal_normal_idx',v_sql_p_normal_idx,'删除原表索引',8,1);
      p_if_judge(v_sql_p_normal_idx,p_deal_flag);
      DBMS_OUTPUT.PUT_LINE('操作步骤 8(处理索引)-------将'||i.table_name ||'的'||i.str||'
列的索引'||i.index_name||'删除完毕');
```

```
      if i.uniqueness='UNIQUE' then
        v_sql_p_normal_idx:='CREATE  UNIQUE  INDEX  '  ||  i.INDEX_NAME  ||  '  ON  '||
p_tab||'('||i.STR||')'||' tablespace '||p_idx_tablespace ;
       elsif i.uniqueness='NONUNIQUE' then
        v_sql_p_normal_idx:='CREATE   INDEX  ' || i.INDEX_NAME || ' ON '|| p_tab ||'
('||i.STR||')'||' LOCAL tablespace '||p_idx_tablespace ;
       end if;
     p_insert_log(p_tab,'p_deal_normal_idx',v_sql_p_normal_idx,'将新分区表的索引加上
',8,2);
     p_if_judge(v_sql_p_normal_idx,p_deal_flag);
     DBMS_OUTPUT.PUT_LINE('操作步骤 8(处理索引)-------对'||p_tab ||'新分区表'||i.STR||'列
增加索引'||i.index_name);
   end loop;
  else
    DBMS_OUTPUT.PUT_LINE('操作步骤 8(处理索引)-------'||UPPER(P_TAB)||'_'||YYYYMMDD ||'
并没有索引(索引模块并不含主键判断)!');
  end if;

 end p_index_008;
```

脚本 13-25　补进分区新表的索引

请看代码 p_pk_009，得到表的主键：

```
procedure  p_pk_009  (p_tab  in  varchar2,p_deal_flag  in  number,p_idx_tablespace  IN
VARCHAR2)
 as
 /*
  功能：从_YYYYMMDD 备份表中得到原表的主键信息，为新分区表增加主键值，并删除旧表主键
  难点：需要考虑联合主键的情况
 */
 v_sql_p_pk          varchar2(4000);
 v_cnt               number;

 begin
 SELECT count(*) into v_cnt
        from USER_IND_COLUMNS
        where index_name in (select index_name
                              from sys.user_constraints t
                             WHERE TABLE_NAME =UPPER(P_TAB)||'_'||YYYYMMDD
                               and constraint_type = 'P');
 if v_cnt>0 then
   for i in
   (WITH T AS
    (SELECT INDEX_NAME,TABLE_NAME,COLUMN_NAME
        from USER_IND_COLUMNS
        where index_name in (select index_name
                              from sys.user_constraints t
                             WHERE TABLE_NAME =UPPER(P_TAB)||'_'||YYYYMMDD
```

```
                                and constraint_type = 'P')
      )
     SELECT  INDEX_NAME,TABLE_NAME,  MAX(substr(sys_connect_by_path(COLUMN_NAME,  ','),
2)) str
    FROM  (SELECT   column_name,INDEX_NAME,TABLE_NAME,  row_number()  over(PARTITION  BY
INDEX_NAME,TABLE_NAME ORDER BY COLUMN_NAME) rn
          FROM T) t
    START WITH rn = 1
    CONNECT BY rn = PRIOR rn + 1
    AND INDEX_NAME = PRIOR INDEX_NAME
    GROUP BY INDEX_NAME,T.TABLE_NAME
) loop
      v_sql_p_pk:= 'alter table '||i.table_name||' drop constraint '||i.index_name||
' cascade';
      p_insert_log(p_tab,'p_deal_pk',v_sql_p_pk,'将备份出来的原表的主键删除',9,1);
      p_if_judge(v_sql_p_pk,p_deal_flag);
      DBMS_OUTPUT.PUT_LINE('操作步骤 9(处理主键)-------将备份出来的原表'||i.table_name||'的
'||i.str||'列的主键'||i.index_name||'删除完毕! '); ---放在 FOR 循环中效率没问题，因为主键只有一
个，只会循环一次
      v_sql_p_pk:='ALTER TABLE '||p_tab||' ADD CONSTRAINT '||I.INDEX_NAME||' PRIMARY
KEY ('||I.STR||')' ||' using index tablespace '||p_idx_tablespace ;
      p_insert_log(p_tab,'p_deal_pk',v_sql_p_pk,'将新分区表的主键加上',9,2);
      p_if_judge(v_sql_p_pk,p_deal_flag);
      DBMS_OUTPUT.PUT_LINE('操作步骤 9(处理主键)-------对'||p_tab ||'表的'||i.str||'列增加
主键'||i.index_name); ---放在 FOR 循环中效率没问题，因为主键只有一个，只会循环一次
    end loop;
   else
      DBMS_OUTPUT.PUT_LINE('操作步骤 9(处理主键)-------'||UPPER(P_TAB)||'_'||YYYYMMDD
||'并没有主键!');
   end if;

  end p_pk_009;
```

脚本 13-26　补进分区新表的主键

请看代码 p_constraint_010，得到表的约束：

```
procedure p_constraint_010 (p_tab in varchar2,p_deal_flag in number)
  as
  /*
   功能：从_YYYYMMDD 备份表中得到原表的约束，为新分区表增加约束值，并删除旧表约束
   难点：需要考虑联合外键 REFERENCE 的情况
  */
  v_sql_p_constraint          varchar2(4000);
  v_cnt  number;
  begin
  SELECT count(*) into v_cnt   FROM user_constraints where table_name=UPPER(P_TAB)||
'_'||YYYYMMDD AND CONSTRAINT_TYPE='R';
```

```
 if v_cnt>0 then
   for i in
   (with t1 as (
     SELECT  /*+no_merge */
            POSITION
           ,t.owner,t.constraint_name  as  constraint_name1,t.table_name  as  table_name1
,t.column_name as column_name1  FROM user_cons_columns t where constraint_name in
     (
     SELECT  CONSTRAINT_NAME  FROM  user_constraints  where  table_name=UPPER(P_TAB)
||'_'||YYYYMMDD AND CONSTRAINT_TYPE='R'
     )
     ),
   t2 as (
     SELECT  /*+no_merge */
            t.POSITION
           ,c.constraint_name constraint_name1
           ,t.constraint_name   as   constraint_name2,t.table_name   as   table_name2
,t.column_name as column_name2
           ,MAX(t.POSITION) OVER (PARTITION BY c.constraint_name) MAX_POSITION
       FROM user_cons_columns t
           ,user_constraints c
       WHERE c.table_name = UPPER(P_TAB)||'_'||YYYYMMDD
             AND t.constraint_name = c.r_constraint_name
             AND c.constraint_type='R'
     ),
   t3 AS (
     SELECT t1.*
           ,t2.constraint_name2
           ,t2.table_name2
           ,t2.column_name2
           ,t2.max_position
     FROM t1,t2
     WHERE t1.constraint_name1 = t2.constraint_name1 AND t1.position=t2.position)
   select  t3.*,SUBSTR(SYS_CONNECT_BY_PATH(column_name1,','),2)  as  FK,SUBSTR(SYS_
CONNECT_BY_PATH(column_name2,','),2) AS PK from t3
     WHERE POSITION=MAX_POSITION
     START WITH position=1
     CONNECT BY constraint_name1 = PRIOR constraint_name1
              AND position = PRIOR position+1) loop
   v_sql_p_constraint:= 'alter table '||p_tab||'_'||YYYYMMDD  ||' drop constraint
'||i.constraint_name1;
   p_insert_log(p_tab,'p_deal_constraint',v_sql_p_constraint,'删除原表 FK 外键' ,10,1);
   p_if_judge(v_sql_p_constraint,p_deal_flag);
   DBMS_OUTPUT.PUT_LINE('操作步骤 10(处理外键)------将备份出来的'||i.table_name1||'表的
'||i.column_name1||'列的外键'||i.constraint_name1||'删除完毕!');
   v_sql_p_constraint:= 'alter table ' || p_tab ||' add constraint '||i.constraint_
```

```
name1 || ' foreign key ( '
    ||i.fk||') references '||i.table_name2|| ' ('||i.pk||' )';
    p_insert_log(p_tab,'p_deal_constraint',v_sql_p_constraint,'将新分区表的外键加上',10,2);
    p_if_judge(v_sql_p_constraint,p_deal_flag);
    DBMS_OUTPUT.PUT_LINE('操作步骤 10(处理外键)------对'||p_tab ||'表的'||i.column_
name1||'列增加外键'||i.constraint_name1);
    end loop;
   else
    DBMS_OUTPUT.PUT_LINE('操作步骤 10(处理外键)------'||UPPER(P_TAB)||'_'||YYYYMMDD ||'并
没有外键!');
   end if;

  end p_constraint_010;
```

脚本 13-27　补进分区新表的外键及主外键约束

其他还有避免同一进程同时执行、带变量的视图、降低高水平脚本，等等，限于篇幅就不再描述了， 具体实现的细节还请扫描下面二维码。

13.3　本章习题、总结与延伸

习题 1：谈谈你对 PL/SQL 集合的优化案例的认识，并体会相关试验，截图证明。

习题 2：请自行下载分区表改造的脚本，并完成测试表的改造试验，截图证明。

习题 3：请研究带变量视图中的脚本，说说主要思路是什么。

习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

第 14 章　动手，高级写法应用让 SQL 飞

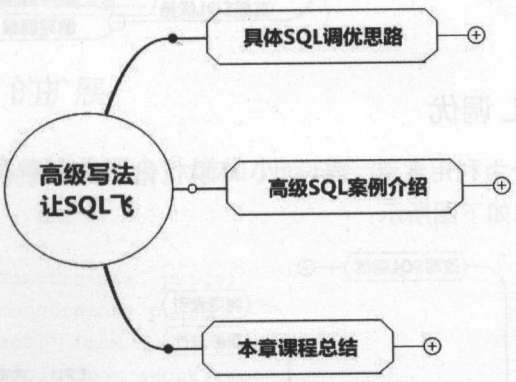
高级与否答案自在心中

同学们，现在我们开始学习高级 SQL 让性能提升的方法……

停！天哪，难道之前我们学的都是低级 SQL 吗，退学费！

冷静，其实这里说的高级 SQL，只是一种说法，大概就是一些看起来有些不寻常的特殊 SQL，比如 insert all 语句、with 子句、merge 等。这些 SQL 有一个共同的特点，就是它们都在内部被优化过了，可以通过比较简单、单步骤的语法来实现复杂和多步骤的功能，同时在性能上还能有不少提升。

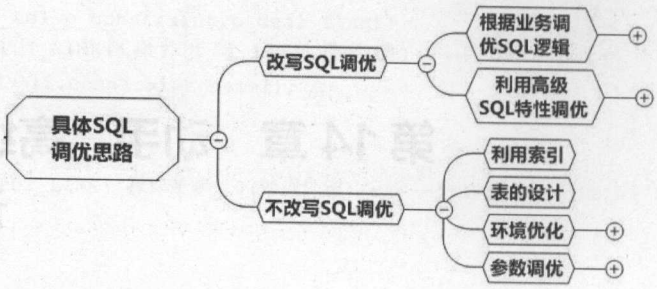
本章总体学习思路如下图所示：



14.1　具体 SQL 调优思路

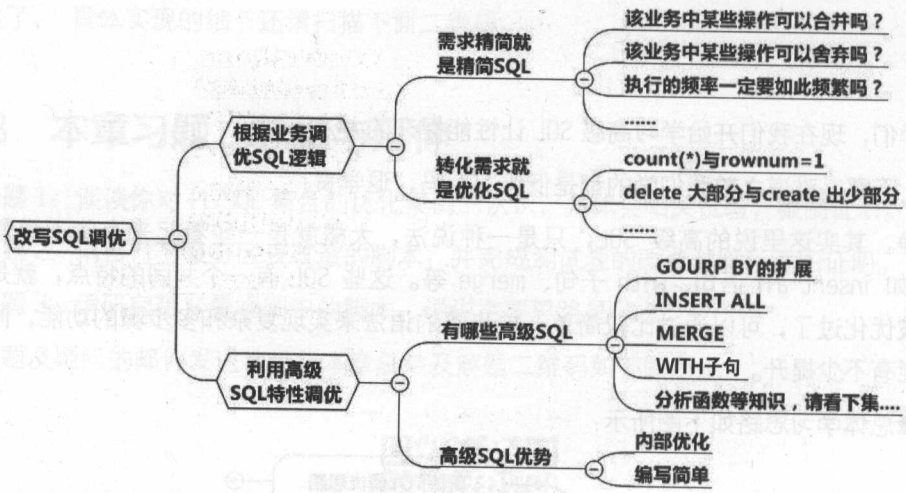
具体的 SQL 调优思路分为改写和不改写两类。一般而言，改写的代价是比较大的，原因不

止在于改写 SQL 的工作量，还在于改写好之后必须要经过测试、打补丁上线这个过程，且还承担了一定的出现 bug 的风险。



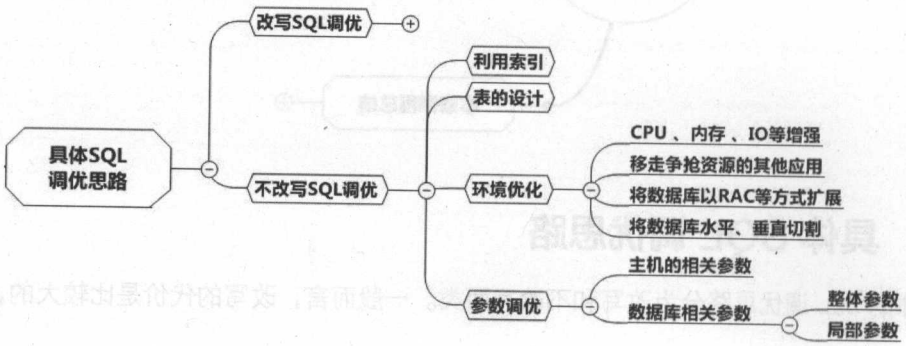
14.1.1 改写 SQL 调优

改写 SQL 调优主要分为根据业务 SQL 调优和利用高级 SQL 特性调优。具体如下：



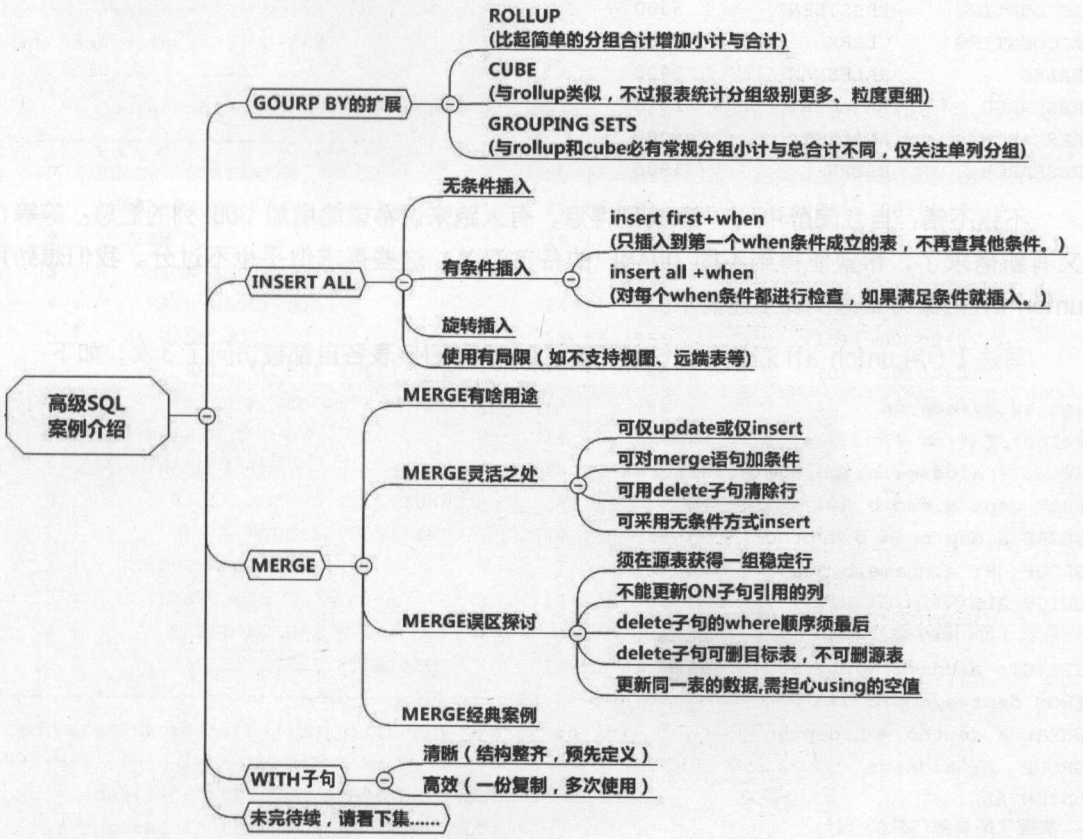
14.1.2 不改写 SQL 调优

不改写 SQL 优化可分为利用索引、表设计、环境优化、参数调优这几部分，这些都在前面的章节中有过描述，具体如下图所示：



14.2　高级 SQL 介绍与案例

下面介绍高级 SQL，具体语法本书不做阐述，主要介绍大致功能点，然后在案例中比较用高级 SQL 和普通 SQL 的区别。具体知识结构如下图所示：



14.2.1　GOURP BY 的扩展

1. ROLLUP（比起简单的分组合计增加小计与合计）

构造环境，如下：

```
drop table dept cascade constraints  purge;
drop table emp  cascade constraints purge;
create table dept as select * from scott.dept;
create table emp  as select * from scott.emp;
```

写一个最简单的根据各部门工作岗位进行汇总的工资表，如下：

```
SELECT  a.dname,b.job,SUM(b.sal) sum_sal
FROM dept a,emp b
WHERE a.deptno = b.deptno
GROUP  BY a.dname,b.job;
```

```
DNAME          JOB          SUM_SAL
-------------- --------- ----------
SALES          MANAGER         2850
SALES          CLERK            950
ACCOUNTING     MANAGER         2450
ACCOUNTING     PRESIDENT       5000
ACCOUNTING     CLERK           1300
SALES          SALESMAN        5600
RESEARCH       MANAGER         2975
RESEARCH       ANALYST         6000
RESEARCH       CLERK           1900
```

不错不错，自我陶醉中……停!先别得意，有人跑来说希望能增加 JOB 列的汇总。等等，又有新需求了，希望能得到不同 DNAME 的各自汇总。这些需求似乎也不过分。我们想到用 union all 汇聚方式来实现上述需求。

写法 1（用 union all 汇总），性能不高，DEPT 和 EMP 表各自都被访问了 3 次，如下：

```
set autotrace on
select * from (
SELECT  a.dname,b.job,SUM(b.sal) sum_sal
FROM dept a,emp b
WHERE a.deptno = b.deptno
GROUP  BY a.dname,b.job
UNION ALL
--实现了部门的小计
SELECT  a.dname,NULL, SUM(b.sal) sum_sal
FROM dept a,emp b
WHERE a.deptno = b.deptno
GROUP  BY a.dname
UNION ALL
--实现了所有部门总的合计
SELECT  NULL,NULL, SUM(b.sal) sum_sal
FROM dept a,emp b
WHERE a.deptno = b.deptno)
order by dname;

DNAME          JOB          SUM_SAL
-------------- --------- ----------
ACCOUNTING     CLERK           1300
ACCOUNTING     MANAGER         2450
ACCOUNTING     PRESIDENT       5000
ACCOUNTING                     8750
RESEARCH       CLERK           1900
RESEARCH       MANAGER         2975
RESEARCH       ANALYST         6000
RESEARCH                      10875
SALES          CLERK            950
SALES          MANAGER         2850
```

```
SALES          SALESMAN         5600
SALES                           9400
                                29025

--对应的执行计划
----------------
Plan hash value: 2979078843
----------------------------
| Id  | Operation               | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |      |    29 |   812 |    23  (22)| 00:00:01 |
|   1 |  SORT ORDER BY          |      |    29 |   812 |    23  (22)| 00:00:01 |
|   2 |   VIEW                  |      |    29 |   812 |    22  (19)| 00:00:01 |
|   3 |    UNION-ALL            |      |       |       |            |          |
|   4 |     HASH GROUP BY       |      |    14 |   756 |     8  (25)| 00:00:01 |
|*  5 |      HASH JOIN          |      |    14 |   756 |     7  (15)| 00:00:01 |
|   6 |       TABLE ACCESS FULL| DEPT |     4 |    88 |     3   (0)| 00:00:01 |
|   7 |       TABLE ACCESS FULL| EMP  |    14 |   448 |     3   (0)| 00:00:01 |
|   8 |     HASH GROUP BY       |      |    14 |   672 |     8  (25)| 00:00:01 |
|*  9 |      HASH JOIN          |      |    14 |   672 |     7  (15)| 00:00:01 |
|  10 |       TABLE ACCESS FULL| DEPT |     4 |    88 |     3   (0)| 00:00:01 |
|  11 |       TABLE ACCESS FULL| EMP  |    14 |   364 |     3   (0)| 00:00:01 |
|  12 |     SORT AGGREGATE      |      |     1 |    39 |            |          |
|* 13 |      HASH JOIN          |      |    14 |   546 |     7  (15)| 00:00:01 |
|  14 |       TABLE ACCESS FULL| DEPT |     4 |    52 |     3   (0)| 00:00:01 |
|  15 |       TABLE ACCESS FULL| EMP  |    14 |   364 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access("A"."DEPTNO"="B"."DEPTNO")
   9 - access("A"."DEPTNO"="B"."DEPTNO")
  13 - access("A"."DEPTNO"="B"."DEPTNO")
统计信息
--------
          0  recursive calls
          0  db block gets
         18  consistent gets
          0  physical reads
          0  redo size
        783  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
         13  rows processed
```

脚本 14-1　普通的分组聚合写法

写法 2（如果你会 rollup 神功，不仅书写简单，同时性能还能大幅提升），DEPT 和 EMP

表各自仅被访问了 1 次，如下：

```
set autotrace on
SELECT  a.dname,b.job, SUM(b.sal) sum_sal
FROM dept a,emp b
WHERE a.deptno = b.deptno
GROUP  BY ROLLUP(a.dname,b.job);

DNAME          JOB          SUM_SAL
-------------- --------- ----------
SALES          CLERK            950
SALES          MANAGER         2850
SALES          SALESMAN        5600
SALES                          9400
RESEARCH       CLERK           1900
RESEARCH       ANALYST         6000
RESEARCH       MANAGER         2975
RESEARCH                      10875
ACCOUNTING     CLERK           1300
ACCOUNTING     MANAGER         2450
ACCOUNTING     PRESIDENT       5000
ACCOUNTING                     8750
                               29025

rollup 写法产生的执行计划
--------------------------
Plan hash value: 1037965942
---------------------------
| Id  | Operation             | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |    14 |   756 |     8  (25)| 00:00:01 |
|   1 |  SORT GROUP BY ROLLUP|      |    14 |   756 |     8  (25)| 00:00:01 |
|*  2 |   HASH JOIN           |      |    14 |   756 |     7  (15)| 00:00:01 |
|   3 |    TABLE ACCESS FULL | DEPT |     4 |    88 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL | EMP  |    14 |   448 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("A"."DEPTNO"="B"."DEPTNO")
统计信息
--------
          0  recursive calls
          0  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
        778  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

```
          1  sorts (memory)
          0  sorts (disk)
         13  rows processed
```

脚本 14-2　分组聚合的 rollup 改写

另外，如果你想再多一个维度，比如再增加雇用年份的统计，之前用 union all 硬拼凑的方法要崩溃了吧，不过 rollup 可以轻松搞定，如下：

```
SELECT to_char(b.hiredate,'yyyy') hire_year,a.dname,b.job, SUM(sal) sum_sal
FROM dept a,emp b
WHERE a.deptno = b.deptno
GROUP BY ROLLUP(to_char(b.hiredate,'yyyy'),a.dname,b.job);

HIRE DNAME          JOB           SUM_SAL
---- -------------- --------- ----------
1980 RESEARCH       CLERK            800
1980 RESEARCH                        800
1980                                 800
1981 SALES          CLERK            950
1981 SALES          MANAGER         2850
1981 SALES          SALESMAN        5600
1981 SALES                          9400
1981 RESEARCH       ANALYST         3000
1981 RESEARCH       MANAGER         2975
1981 RESEARCH                       5975
1981 ACCOUNTING     MANAGER         2450
1981 ACCOUNTING     PRESIDENT       5000
1981 ACCOUNTING                     7450
1981                               22825
1982 ACCOUNTING     CLERK           1300
1982 ACCOUNTING                     1300
1982                                1300
1987 RESEARCH       CLERK           1100
1987 RESEARCH       ANALYST         3000
1987 RESEARCH                       4100
1987                                4100
                                   29025

执行计划
--------
Plan hash value: 1037965942
----------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |    14 |   882 |     8  (25)| 00:00:01 |
|   1 |  SORT GROUP BY ROLLUP|      |    14 |   882 |     8  (25)| 00:00:01 |
|*  2 |   HASH JOIN          |      |    14 |   882 |     7  (15)| 00:00:01 |
|   3 |    TABLE ACCESS FULL | DEPT |     4 |    88 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL | EMP  |    14 |   574 |     3   (0)| 00:00:01 |
```

```
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("A"."DEPTNO"="B"."DEPTNO")
统计信息
--------
          0  recursive calls
          0  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
       1107  bytes sent via SQL*Net to client
        427  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
         22  rows processed
```

脚本 14-3　增加分组维度的聚合用 rollup 轻松完成

注意到没？本次 SQL 多统计了一个维度，如果用 union all 叠加的方式，DEPT 和 EMP 表各自肯定要被访问 4 次，可是这次，依然是各自 1 次。而且无论是 COST 还是逻辑读，都没有增加，够帅！

再有，不止是增加维度，如果考虑更换维度的次序，这对 rollup 也是轻而易举的事，改写起来就是一个 rollup 里顺序互调一下，如下：

```
SELECT  b.job,a.dname, SUM(b.sal) sum_sal
FROM dept a,emp b
WHERE a.deptno = b.deptno
GROUP  BY ROLLUP(b.job,a.dname);

JOB       DNAME          SUM_SAL
--------- -------------- ----------
CLERK     SALES              950
CLERK     RESEARCH          1900
CLERK     ACCOUNTING        1300
CLERK                       4150
ANALYST   RESEARCH          6000
ANALYST                     6000
MANAGER   SALES             2850
MANAGER   RESEARCH          2975
MANAGER   ACCOUNTING        2450
MANAGER                     8275
SALESMAN  SALES             5600
SALESMAN                    5600
PRESIDENT ACCOUNTING        5000
PRESIDENT                   5000
                           29025
```

脚本 14-4　更换纬度次序的聚合用 rollup 轻松完成

2. CUBE（比 rollup 粒度更细）

关于 CUBE，和 rollup 类似，不过它在报表统计上分组级别更多、粒度更细，举例如下：

```
SELECT  a.dname,b.job, SUM(b.sal) sum_sal
FROM dept a,emp b
WHERE a.deptno = b.deptno
GROUP  BY CUBE(a.dname,b.job);

DNAME          JOB          SUM_SAL
-------------- --------- ----------
                            29025
               CLERK          4150
               ANALYST        6000
               MANAGER        8275
               SALESMAN       5600
               PRESIDENT      5000
SALES                         9400
SALES          CLERK           950
SALES          MANAGER        2850
SALES          SALESMAN       5600
RESEARCH                     10875
RESEARCH       CLERK          1900
RESEARCH       ANALYST        6000
RESEARCH       MANAGER        2975
ACCOUNTING                    8750
ACCOUNTING     CLERK          1300
ACCOUNTING     MANAGER        2450
ACCOUNTING     PRESIDENT      5000
```

14.2.2　INSERT ALL

1. 无条件插入

```
drop table t ;
drop table t1;
drop table t2;
create table t as select object_name,rownum as object_id from dba_objects where
rownum<=10;
create table t1 as select * from t where 1=2;
create table t2 as select * from t where 1=2;
set autotrace off
insert ALL INTO t1(object_name,object_id) into t2(object_name,object_id) select *
from t;
commit;

select count(*) from t1;
 COUNT(*)
---------
   10
```

```
select count(*) from t2;
 COUNT(*)
---------
   10
```

脚本 14-5　INSERT 无条件插入

2. 有条件插入

insert all：对于每一行数据，对每一个 when 条件都进行检查，如果满足条件就执行插入操作。

```
truncate table t1;
truncate table t2;
insert all
when object_id < 5 then
into t1(object_name,object_id)
when object_id >=5 then
into t2(object_name,object_id)
select * from t;
commit;
```

脚本 14-6　INSERT 有条件插入

insert first：对于每一行数据，只插入到第一个 when 条件成立的表，不继续检查其他条件。

```
truncate table t1;
truncate table t2;
--前面等于 1 的条件被<=5 含在内，FIRST 就表示前面插入了，后面不会再插入了。
insert first
when object_id = 1 then
into t1(object_name,object_id)
when object_id <=5 then
into t2(object_name,object_id)
select * from t;
commit;
```

脚本 14-7　INSERT 有条件插入(insert first)

3. 旋转插入

构造环境：

```
drop table sales_source_data purge;
create table sales_source_data (
employee_id number(6),
week_id number(2),
sales_mon number(8,2),
sales_tue number(8,2),
sales_wed number(8,2),
sales_thur number(8,2),
sales_fri number(8,2)
```

```
);
insert into sales_source_data values (176,6,2000,3000,4000,5000,6000);
commit;

drop table sales_info purge;
create table sales_info (
employee_id number(6),
week number(2),
sales number(8,2)
);
```

行转列插入（pivoting insert）：

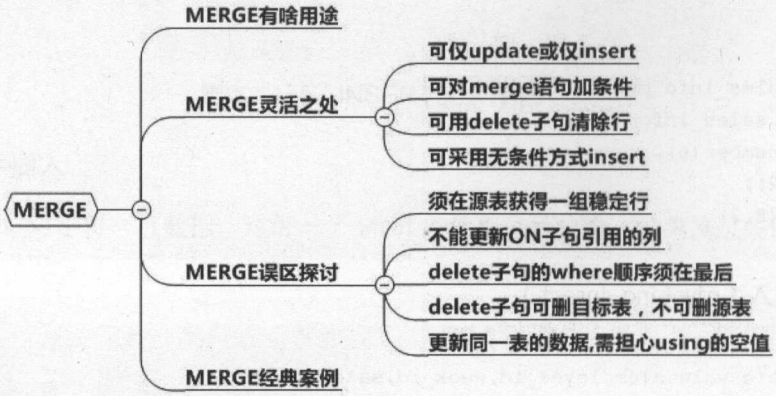
```
insert all
into sales_info values(employee_id,week_id,sales_mon)
into sales_info values(employee_id,week_id,sales_tue)
into sales_info values(employee_id,week_id,sales_wed)
into sales_info values(employee_id,week_id,sales_thur)
into sales_info values(employee_id,week_id,sales_fri)
select employee_id,week_id,sales_mon,sales_tue,
sales_wed,sales_thur,sales_fri
from sales_source_data;
select * from sales_source_data;
EMPLOYEE_ID    WEEK_ID  SALES_MON  SALES_TUE  SALES_WED SALES_THUR  SALES_FRI
----------- ---------- ---------- ---------- ---------- ---------- ----------
       176          6       2000       3000       4000       5000       6000

select * from sales_info;
EMPLOYEE_ID       WEEK      SALES
----------- ---------- ----------
        176          6       2000
        176          6       3000
        176          6       4000
        176          6       5000
        176          6       6000
```

脚本 14-8　INSERT 行转列插入

4. 使用有局限（如不支持视图、远端表等）

- 只能对表执行多表插入，不能对视图或物化视图执行。
- 不能对远端表执行多表插入。
- 不能使用表集合表达式。
- 不能超过 999 个目标列。
- 在 RAC 环境中或目标表是索引组织表或目标表上建有 BITMAP 索引时，多表插入不能并行执行。
- 多表插入不支持执行计划稳定性。
- 多表插入语句中的子查询不能使用序列。

14.2.3 MERGE



1. MERGE 有啥用途

先看一个简单的需求：从 T1 表更新数据到 T2 表中，如果 T2 表的 NAME 在 T1 表中已存在，就将 MONEY 累加，如果不存在，将 T1 表的记录插入到 T2 表中。

大家知道，按一般逻辑思路，该需求至少需要 UPDATE 和 INSERT 两条 SQL 才能完成，如考虑在 PL/SQL 中用纯编程语言思路实现，则必须还要考虑判断的逻辑，这样显得更麻烦了。而 MERGE 语句可直接用单条 SQL 简洁明快地实现此业务逻辑，具体如下：

```
MERGE INTO T2
USING T1
ON (T1.NAME=T2.NAME)
WHEN MATCHED THEN
UPDATE
SET T2.MONEY=T1.MONEY+T2.MONEY
WHEN NOT MATCHED THEN
INSERT
VALUES (T1.NAME,T1.MONEY);
```

脚本 14-9　MERGE 的传统用法

2. MERGE 灵活之处

（1）UPDATE 和 INSERT 动作可只出现其一（Oracle 9i 要求必须同时出现！）

可选择仅 UPDATE 目标表：

```
SQL> MERGE INTO T2
  2  USING T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
  6  SET T2.MONEY=T1.MONEY+T2.MONEY;
Done
```

脚本 14-10　MERGE 仅 UPDATE 目标表

也可选择仅仅 INSERT 目标表而不做任何 UPDATE 动作：

```
SQL> MERGE INTO T2
  2  USING T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN NOT MATCHED THEN
  5  INSERT
  6  VALUES (T1.NAME,T1.MONEY);
Done
```

脚本 14-11　MERGE 仅仅 INSERT 目标表

（2）可对 MERGE 语句加条件

```
SQL> MERGE INTO T2
  2  USING T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
  6  SET T2.MONEY=T1.MONEY+T2.MONEY
  7  WHERE T1.NAME='A' ---此处表示对 MERGE 的条件进行过滤
  8  ;
Done
```

脚本 14-12　可对 MERGE 语句加条件

（3）可用 DELETE 子句清除行

在这种情况下，首先是要获取满足 T1.NAME=T2.NAME 的记录，如果 T2.NAME='A'并不满足 T1.NAME=T2.NAME 过滤出的记录集，那这个 DELETE 是不会生效的。在满足的条件下，可以删除目标表的记录。

```
MERGE INTO T2
USING T1
ON (T1.NAME=T2.NAME)
WHEN MATCHED THEN
UPDATE
SET T2.MONEY=T1.MONEY+T2.MONEY
DELETE WHERE (T2.NAME = 'A');
```

脚本 14-13　MERGE 可用 DELETE 子句清除行

（4）可采用无条件方式插入

方法很简单，在 ON 关键字处写上恒不等条件（如 1=2）后，MATCHED 语句的 INSERT 就变为无条件 INSERT 了，同于 INSERT...SELECT 的写法，具体如下：

```
SQL> MERGE INTO T2
  2  USING T1
  3  ON (1=2)
  4  WHEN NOT MATCHED THEN
  5  INSERT
```

```
  6  VALUES (T1.NAME,T1.MONEY);
Done
```

脚本 14-14　MERGE 可采用无条件方式插入

3. MERGE 误区探讨

（1）无法在源表中获得一组稳定的行

在 MERGE INTO T2 USING T1 ON 的 MERGE 表达式中，如果一条 T2 记录被连接到多条 T1 记录，就产生了 ORA-30926 错误。构造 T1、T2 表进行如下试验，此次 T1 表中增加了('A',30)的记录，如下：

```
DROP TABLE T1;
CREATE TABLE T1 (NAME VARCHAR2(20),MONEY NUMBER);
INSERT INTO T1 VALUES ('A',10);
INSERT INTO T1 VALUES ('A',30);
INSERT INTO T1 VALUES ('B',20);
DROP TABLE T2;
CREATE TABLE T2 (NAME VARCHAR2(20),MONEY NUMBER);
INSERT INTO T2 VALUES ('A',30);
INSERT INTO T2 VALUES ('C',20);
COMMIT
```

继续执行如下

```
SQL> MERGE INTO T2
  2  USING T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
  6  SET T2.MONEY=T1.MONEY+T2.MONEY
  7  ;
ORA-30926: 无法在源表中获得一组稳定的行
```

脚本 14-15　MERGE 无法在源表中获得一组稳定的行

对于 Oracle 中的 MERGE 语句，应该保证 ON 中的条件的唯一性，当 T1.NAME='A'时，T2 表记录对应到了 T1 表的两条记录，所以就出错了。

解决方法很简单，可对 T1 表和 T2 表的关联字段建主键，这样基本上就不可能出现上面的问题，而且一般而言，MERGE 语句的关联字段互相有主键，MERGE 的效率将比较高！

或者是将 T1 表的 ID 列做一个聚合，这样归并成单条，也能避免此类错误。如：

```
SQL> MERGE INTO T2
  2  USING (SELECT NAME,SUM(MONEY) AS MONEY FROM T1 GROUP BY NAME)T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
  6  SET T2.MONEY=T1.MONEY+T2.MONEY
```

```
  7  ;
Done
```

脚本 14-16　MERGE 无法在源表中获稳定行的规避

但是这样的改造需要注意，因为有可能改变了最终的需求。此外需要引起注意的是，在 MERGE INTO T2 USING T1 ON 的 MERGE 表达式中，如果反过来，一条 T1 记录被连接到多条 T2 记录，是可以导致多条 T2 记录都被更新而不会出错的！继续构造 T1、T2 表进行试验，此次是在 T2 表中增加了('A',40)的记录，如下：

```
DROP TABLE T1;
CREATE TABLE T1 (NAME VARCHAR2(20),MONEY NUMBER);
INSERT INTO T1 VALUES ('A',10);
INSERT INTO T1 VALUES ('B',20);
DROP TABLE T2;
CREATE TABLE T2 (NAME VARCHAR2(20),MONEY NUMBER);
INSERT INTO T2 VALUES ('A',30);
INSERT INTO T2 VALUES ('A',40);
INSERT INTO T2 VALUES ('C',20);
COMMIT
```

继续执行，发现执行可以成功并没有报无法在源表中获得一组稳定的行的 ORA-30926 错误

```
SQL>  MERGE INTO T2
  2      USING T1
  3      ON (T1.NAME=T2.NAME)
  4      WHEN MATCHED THEN
  5      UPDATE
  6      SET T2.MONEY=T1.MONEY+T2.MONEY
  7      ;
Done
SQL> COMMIT;
Commit complete
```

查看 T2 表，发现 T2 表中 NAME=A 的 2 条记录都被更新了：

```
SQL> SELECT * FROM T2;
NAME                      MONEY
-------------------- ----------------------------------
A                            40
A                            50
C                            20
```

脚本 14-17　MERGE 中一对多的陷阱

（2）DELETE 子句的 WHERE 顺序必须在最后

```
SQL> MERGE INTO T2
  2  USING T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
```

```
  6  SET T2.MONEY=T1.MONEY+T2.MONEY
  7  DELETE WHERE (T2.NAME = 'A')
  8  WHERE T1.NAME='A';
/
ORA-00933: SQL 命令未正确结束
```

改为如下即可：

```
SQL> MERGE INTO T2
  2  USING T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
  6  SET T2.MONEY=T1.MONEY+T2.MONEY
  7  WHERE T1.NAME='A'
  8  DELETE WHERE (T2.NAME = 'A')
  9  ;
Done
```

脚本 14-18　MERGE 中 DELETE 子句的 WHERE 顺序

注：只要是 MERGE 语句，UPDATE 和 DELETE 两者必须要出现其一，所以上面的脚本是不能省略 UPDATE 而只做 DELETE 的。另外 WHERE (T2.NAME = 'A')的括号可以省略。

（3）DELETE 子句只可以删除目标表，而无法删除源表

这里需要引起注意，无论 DELETE WHERE (T2.NAME = 'A')这个写法中的 T2 是否改写为 T1，效果都一样，都是对目标表进行删除！

```
SQL> SELECT * FROM T1;
NAME                    MONEY
-------------------- ------------------------------
A                            10
B                            20
SQL> SELECT * FROM T2;
NAME                    MONEY
-------------------- ------------------------------
A                            30
C                            20
SQL> MERGE INTO T2
  2  USING T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
  6  SET T2.MONEY=T1.MONEY+T2.MONEY
  7  DELETE WHERE (T2.NAME = 'A' )
  8  ;
Done
SQL> SELECT * FROM T1;
NAME                    MONEY
--------------------------------
```

```
A                              10
B                              20
SQL> SELECT * FROM T2;
NAME                         MONEY
-------------------- -----------
C                              20
```

可以看出目标表的 A 记录被删除了，但是如果把 DELETE WHERE (T2.NAME = 'A')修改为 DELETE WHERE (T1.NAME = 'A')，是否就会把源表的 T1 记录给删除了呢？试验如下：

```
Rollback complete
SQL> MERGE INTO T2
  2  USING T1
  3  ON (T1.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
  6  SET T2.MONEY=T1.MONEY+T2.MONEY
  7  DELETE WHERE (T2.NAME = 'A' )
  8  ;
Done
SQL> SELECT * FROM T1;
NAME                         MONEY
-------------------- --------------
A                              10
B                              20
SQL> SELECT * FROM T2;
NAME                         MONEY
-------------------- ---------------
C                              20
```

脚本 14-19　MERGE 中 DELETE 子句不能删除源表

发现其实 T1 源表的记录还是保留着，只是目标表被删除了。

（4）更新同一张表的数据，需担心 USING 的空值

```
SQL> SELECT * FROM T2;
NAME                         MONEY
-------------------- -------------
A                              30
C                              20
```

需求为对 T2 表进行自我更新，如果在 T2 表中发现 NAME='D'的记录，就将该记录的 MONEY 字段更新为 100，如果 NAME='D'的记录不存在，则自动增加 NAME='D'的记录。

根据语法完成如下代码：

```
SQL> MERGE INTO T2
  2  USING (SELECT * FROM t2 WHERE NAME='D') T
  3  ON (T.NAME=T2.NAME)
  4  WHEN MATCHED THEN
  5  UPDATE
```

```
  6  SET T2.MONEY=100
  7  WHEN NOT MATCHED THEN
  8  INSERT
  9  VALUES ('D',200)
 10  ;
Done
```

但是查询发现，本来 T 表应该因为 NAME='D'不存在而要增加记录，但是实际却根本无变化。

```
SQL> SELECT * FROM T2;
NAME                          MONEY
-----------------------------------
A                               30
C                               20
```

原因是 USING 后面必须包含要更新或插入的行。而第一个 USING (SELECT * FROM t2 WHERE NAME='D') T 中根本没有这一行，可进行如下改造巧妙实现需求：

```
SQL> MERGE INTO T2
  2  USING (SELECT COUNT(*) CNT FROM t2 WHERE NAME='D') T
  3  ON (T.CNT<>0)
  4  WHEN MATCHED THEN
  5  UPDATE
  6  SET T2.MONEY=100
  7  WHEN NOT MATCHED THEN
  8  INSERT
  9  VALUES ('D',100)
 10  ;
Done
SQL> SELECT * FROM T2;
NAME                          MONEY
-------------------- --------------------
A                               30
C                               20
D                              100
```

脚本 14-20　MERGE 中 USIING 无记录的应对

4. MERGE 经典案例

（1）案例 1——互换值

需求为：将如下 TEST 记录的 ID=1 的 NAME 的值改为 ID=2 的 NAME 的值，把 ID=2 的 NAME 的值改为 ID=1 的 NAME 的值。

```
drop table test;
create table test (id number,name varchar2(20));
insert into test values (1,'a');
insert into test values (2,'b');
COMMIT;
```

```
SQL> SELECT * FROM test;
        ID NAME
---------- --------------------
         1 a
         2 b
```

如果执行如下：

```
UPDATE TEST SET NAME =(SELECT NAME FROM TEST WHERE ID=2) WHERE ID=1;
```

此时 ID=1 的 NAME 值已改变了，就不可能用如下来更新了：

```
UPDATE TEST SET NAME =(SELECT NAME FROM TEST WHERE ID=1) WHERE ID=1;
```

如果是过程就很简单了，可以把原先的值先存储起来。但是是否单条 SQL 就一定不行呢？

其实单条 SQL 是可以解决的，可考虑灵活利用 MERGE 特性！思路是，可考虑构造出一个虚拟表 T，然后再根据此虚拟 T 表和真实的 TEST 表进行 MERGE 更新，就可以方便快捷地完成任务了。构造虚拟表的方法类似如下：

```
SQL> SELECT 1 id,(SELECT name FROM test WHERE id=2) name FROM DUAL
  2         UNION ALL
  3         SELECT 2,(SELECT name FROM test WHERE id=1) FROM DUAL
  4  ;
        ID NAME
---------- --------------------
         1 b
         2 a
```

有了此思路，结合前面所学的 MERGE 知识，就可以通过如下简洁优雅的代码完成更新：

```
SQL> MERGE INTO TEST
  2  USING (SELECT 1 id,(SELECT name FROM test WHERE id=2) name FROM DUAL
  3         UNION ALL
  4         SELECT 2,(SELECT name FROM test WHERE id=1) FROM DUAL
  5         ) t
  6  ON (test.id = t.id)
  7  WHEN MATCHED THEN UPDATE set TEST.name = t.name
  8  ;
Done
SQL> SELECT * FROM test;
        ID NAME
---------- --------------------
         1 b
         2 a
```

脚本 14-21　用 MERGE 完成值的互换更新

注：如果是 9i 固定需要 INSERT，则需要加上如下内容：

```
WHEN NOT MATCHED THEN
INSERT   VALUES (1,'a')
```

本案例用的是 MERGE 方法，当然，构造虚拟表也是一个非常重要的思路，如果只是查询

出改变后的结果而不是真实地进行更新，就可以不采用 MERGE，可以直接采用如下方式取出结果：

```
SQL> rollback;
Rollback complete
SQL>
SQL> WITH T AS
  2  (SELECT 1 id,(SELECT name FROM test WHERE id=2) name FROM DUAL
  3         UNION ALL
  4         SELECT 2,(SELECT name FROM test WHERE id=1) FROM DUAL
  5         )
  6  SELECT test.id,t.name FROM test ,t
  7  WHERE test.id=t.id;
        ID NAME
---------- --------------------
         1 b
         2 a
```

脚本 14-22　只是展现取值互换的写法

（2）案例 2——互换值优化

通过 MERGE 可以得到一个非常有用的思想，即：**只要能查出更新后的结果集，就可利用该结果集来更新原表记录，也就是 MERGE+ROWID 方式。**感谢 NEWKID 给予的指点，他精于使用此类方法，下面案例 3 中的复杂 MERGE 更新例子即来自 NEWKID 的精彩脚本。

本案例是案例 1 的延伸，但改变了案例 1 的处理思路，不再采用构造虚拟表 T 来关联 TEST 表的方式，而是直接把真实结果用 SELECT 的方式取出，然后利用这个结果集更新回原表中。

```
SQL> merge into  test using
  2  (
  3  WITH T AS
  4   (SELECT 1 id,(SELECT name FROM test WHERE id=2) name FROM DUAL
  5          UNION ALL
  6          SELECT 2,(SELECT name FROM test WHERE id=1) FROM DUAL
  7          )
  8   SELECT test.id,test.rowid as rn ,t.name FROM test ,t
  9   WHERE test.id=t.id
10  )      n
 11 on(test.rowid=n.rn)
 12  when matched then  update
 13  set test.name=n.name;
SQL> SELECT * FROM test;
        ID NAME
---------- --------------------
         1 a
         2 b
```

脚本 14-23　MERGE 与 ROWID 的结合优化

注：直接更新一个子查询的写法也可行，但是却有很多限制，稍微复杂的查询都易出错。此时用 MERGE 是最好的办法，结合 ROWID 的方式，可快速准确地利用一个已查询出的结果集来更新自己，这是一个非常好的思路的扩展，希望对大家有帮助。

（3）案例 3——库存更新

最后，举一个 USING 里面有复杂的连接、聚合、分析函数的综合性例子来加深读者的印象，从而使读者更深入地理解 MERGE 的强大功能！脚本选自 NEWKID 在答网友提问时的一次精彩回复，提问需求如下：

```
declare
cursor c1 is
SELECT art_no,stock
FROM  tb_fin_art_stock
WHERE  run_date=to_date('&日期','yyyymmdd')-1  and art_no in (158756);
t_art tb_fin_art_stock.art_no%type;
t_stock tb_fin_art_STOCK.stock%type;
begin
open c1;
loop
        fetch c1 into t_art,t_stock;
        exit when c1%notfound;
 update tb_fin_art_stock set stock=t_stock+gor_qty+return_qty-sale_qty+stock_corr+DEL_CORR
                 WHERE run_date=to_date('&日期','yyyymmdd')
                        and art_no=t_art;
        commit;
end loop;
close c1;
end;
```

以上代码，我想更新 tb_fin_art_stock 这个表中某个货号某一天往后所有记录的 stock(库存)字段的值，每天的 stock 是根据前一天的 stock 字段的值加进货减销售得出来的。现在只能一天一天更新。我想问的是，如何能输入日期范围， 比如 10 号到 20 号的记录，根据 9 号更改 10 号，根据 10 号再改 11 号，以此类推，一次就更新了。

MERGE 精彩解决方案如下，有兴趣的读者自行研究，相信必有收获！

```
MERGE INTO tb_fin_art_stock t
USING (SELECT t.ROWID rid
                      ,t2.stock+SUM(gor_qty+return_qty-sale_qty+stock_corr+DEL_CORR)
OVER(PARTITION BY art_no ORDER BY run_date) AS stock
       FROM tb_fin_art_stock t
                  ,(SELECT  art_no,stock  FROM  tb_fin_art_stock  WHERE  run_date  =
lv_start_date-1) t2
       WHERE t.run_date BETWEEN lv_start_date AND lv_end_date
                  AND t.art_no = t2.art_no
       ) n
USING (t.ROWID = n.rid)
WHEN MATCHED THEN UPDATE SET t.stock = n.stock;
```

脚本 14-24　MERGE 解决库存更新问题

14.2.4 WITH 子句

1. 清晰（结构整齐，预先定义）

语法：

```
with
alias_name1 as     (subquery1),
alias_name2 as     (subQuery2),
...
alias_nameN as     (subQueryN)
select col1,col2…… col3
     from alias_name1,alias_name2…,alias_nameN
```

如：

```
WITH
    Q1 AS (SELECT 3 + 5 S FROM DUAL),
    Q2 AS (SELECT 3 * 5 M FROM DUAL),
    Q3 AS (SELECT S, M, S + M, S * M FROM Q1, Q2)
SELECT * FROM Q3;
```

2. 高效（一份复制，多次使用）

构造环境：

```
drop table t_with purge;
CREATE TABLE T_WITH AS SELECT ROWNUM ID, A.* FROM DBA_SOURCE A WHERE ROWNUM < 100001;
```

写法 1：

```
SET autotrace traceonly
Set linesize 1000

SELECT ID, NAME FROM T_WITH
WHERE ID IN
(SELECT MAX(ID) FROM T_WITH
 UNION ALL
 SELECT MIN(ID) FROM T_WITH
 UNION ALL
 SELECT TRUNC(AVG(ID)) FROM T_WITH
);
非 with 子句的执行计划
----------------------
Plan hash value: 647530712
--------------------------
| Id  | Operation              | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |           |     3 |   129 |  1382   (1)| 00:00:17 |
|*  1 |  HASH JOIN             |           |     3 |   129 |  1382   (1)| 00:00:17 |
|   2 |   VIEW                 | VW_NSO_1  |     3 |    39 |  1035   (1)| 00:00:13 |
```

```
|   3 |    HASH UNIQUE          |          |     3 |    39 |  1035   (67)| 00:00:13 |
|   4 |     UNION-ALL           |          |       |       |             |          |
|   5 |      SORT AGGREGATE     |          |     1 |    13 |             |          |
|   6 |       TABLE ACCESS FULL| T_WITH   | 91060 |  1156K|   345   (1)| 00:00:05 |
|   7 |      SORT AGGREGATE     |          |     1 |    13 |             |          |
|   8 |       TABLE ACCESS FULL| T_WITH   | 91060 |  1156K|   345   (1)| 00:00:05 |
|   9 |      SORT AGGREGATE     |          |     1 |    13 |             |          |
|  10 |       TABLE ACCESS FULL| T_WITH   | 91060 |  1156K|   345   (1)| 00:00:05 |
|  11 |   TABLE ACCESS FULL     | T_WITH   | 91060 |  2667K|   345   (1)| 00:00:05 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("ID"="MAX(ID)")
统计信息
--------

          0  recursive calls
          0  db block gets
       4969  consistent gets
          0  physical reads
          0  redo size
        558  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          3  rows processed
```

脚本 14-25　非 WITH 子句的普通写法性能

写法 2：

```
WITH
 AGG AS (SELECT MAX(ID) MAX, MIN(ID) MIN, TRUNC(AVG(ID)) AVG FROM T_WITH)
 SELECT ID, NAME FROM T_WITH
 WHERE ID IN
 (
 SELECT MAX FROM AGG
 UNION ALL
 SELECT MIN FROM AGG
 UNION ALL
 SELECT AVG FROM AGG
 );

with 子句的执行计划
----------------------
Plan hash value: 3855430689
---------------------------
| Id |Operation                | Name                      |Rows  |Bytes |Cost (%CPU)|   Time |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT         |                           |    3 |  129 |  697   (1)|00:00:09|
```

```
|   1| TEMP TABLE TRANSFORMATION |                          |      |      |          |        |
|   2|  LOAD AS SELECT           | SYS_TEMP_0FD9D660D_4164695 |      |      |          |        |
|   3|   SORT AGGREGATE          |                          |    1 |   13 |          |        |
|   4|    TABLE ACCESS FULL      | T_WITH                   |91060 | 1156K|   345    (1)|00:00:05|
|*  5|  HASH JOIN                |                          |    3 |  129 |   352    (1)|00:00:05|
|   6|   VIEW                    | VW_NSO_1                 |    3 |   39 |     6    (0)|00:00:01|
|   7|    HASH UNIQUE            |                          |    3 |   39 |     6   (67)|00:00:01|
|   8|     UNION-ALL             |                          |      |      |          |        |
|   9|      VIEW                 |                          |    1 |   13 |     2    (0)|00:00:01|
|  10|       TABLE ACCESS FULL   | SYS_TEMP_0FD9D660D_4164695 |    1 |   13 |     2    (0)|00:00:01|
|  11|      VIEW                 |                          |    1 |   13 |     2    (0)|00:00:01|
|  12|       TABLE ACCESS FULL   | SYS_TEMP_0FD9D660D_4164695 |    1 |   13 |     2    (0)|00:00:01|
|  13|      VIEW                 |                          |    1 |   13 |     2    (0)|00:00:01|
|  14|       TABLE ACCESS FULL   | SYS_TEMP_0FD9D660D_4164695 |    1 |   13 |     2    (0)|00:00:01|
|  15|   TABLE ACCESS FULL       | T_WITH                   |91060 | 2667K|   345    (1)|00:00:05|
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("ID"="MAX")

统计信息
--------

          2  recursive calls
          8  db block gets
       2496  consistent gets
          1  physical reads
        556  redo size
        558  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          3  rows processed
```

脚本 14-26　WITH 子句写法的性能

写法 1 的 consistent gets 为 4969，而使用 WITH 子句的写法 2 时 consistent gets 为 2496，可以看出写法 2 更为高效。大家可能注意到写法 2 的执行计划中的 SYS_TEMP_0FD9D660D_4164695，这表示 ORACLE 将 WITH 的部分进行了缓存，实现了一次复制，多次使用，这就是性能提升的关键之处。

14.3　本章习题、总结与延伸

习题 1：说说这些高级 SQL 的优势在哪里。

习题 2：你还知道哪些其他高级 SQL？并举例说明。

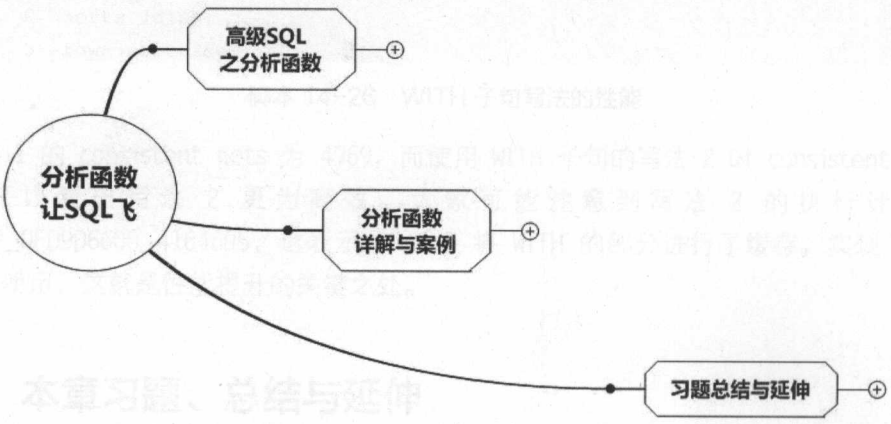
本章习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

第 15 章　动手，分析函数让 SQL 飞

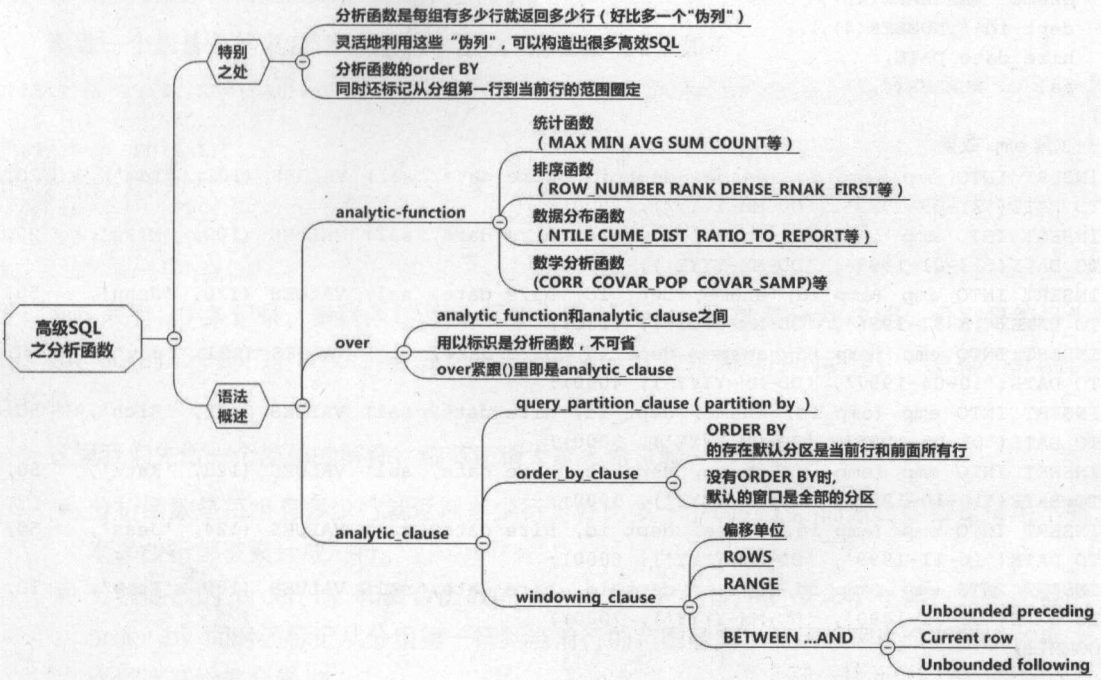
分析函数是世上最优雅的写法

分析函数是 Oracle 8i 引入的一个全新的概念，其为我们分析数据提供了一种简单高效的处理方式。在分析函数出现之前，我们必须使用自联查询、子查询或者内联视图，甚至是复杂的存储过程实现的语句，现在只要一条简单的 SQL 语句即可实现，而且在执行效率方面也有相当大的提高。

Oracle 分析函数实现了一些我们需要编码才能实现的统计功能，这对于简化我们的开发工作有很大的帮助，特别是在开发 BI 报表时有意想不到的效果；同时我们也可以利用分析函数来实现一些特殊的需求。



15.1　高级 SQL 之分析函数



15.1.1　语法概述

1. 分析函数的参数

分析函数可有 0~3 个参数。参数可以是任何数字类型或是可以隐式转换为数字类型的数据类型。Oracle 根据最高数字优先级确定函数参数，并且隐式地将需要处理的参数转换为数字类型。函数的返回类型也为数字类型，除非此函数另有说明。

2. 分析函数的语法

over analytic_clause 用以指明函数操作的是一个查询结果集。也就是说分析函数是在 from、where、group by 和 having 子句之后才开始计算的。因此在选择列或 order by 子句中可以使用分析函数。为了过滤分析函数计算的查询结果，可以将它作为子查询嵌套在外部查询中，然后在外部查询中过滤其查询结果。

15.1.2　特别之处

分析函数的语法和聚合函数有点类似，不过实际情况却差别很大。下面做一组试验来证明，首先看构造环境，如下：

```
drop table emp purge;
CREATE TABLE emp
```

```
(
  emp_id    NUMBER(6),
  ename  VARCHAR2(45),
  dept_id   NUMBER(4),
  hire_date DATE,
  sal    NUMBER(8,2)
);
--创建 emp 数据
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (101, 'Tom',    20,
TO_DATE('21-09-1989', 'DD-MM-YYYY'), 2000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (102, 'Mike',   20,
TO_DATE('13-01-1993', 'DD-MM-YYYY'), 8000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (120, 'John',   50,
TO_DATE('18-07-1996', 'DD-MM-YYYY'), 1000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (121, 'Joy',    50,
TO_DATE('10-04-1997', 'DD-MM-YYYY'), 4000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (122, 'Rich',   50,
TO_DATE('01-05-1995', 'DD-MM-YYYY'), 3000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (123, 'Kate',   50,
TO_DATE('10-10-1997', 'DD-MM-YYYY'), 5000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (124, 'Jess',   50,
TO_DATE('16-11-1999', 'DD-MM-YYYY'), 6000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (100, 'Stev',   10,
TO_DATE('01-01-1990', 'DD-MM-YYYY'), 7000);
COMMIT;

set linesize 2000
set pagesize 2000
col emp_id format 999
col dept_id format 99
col ename format a5
```

接下来，写一个分析函数语句查询结果，请看语句 1，如下：

```
SELECT
  emp_id,ename,dept_id,hire_date,sal,
  SUM(sal) OVER (PARTITION BY dept_id ORDER BY hire_date) sum_sal,
  SUM(sal) OVER (PARTITION BY dept_id ) sum_sal2,
  SUM(sal) OVER ( ) sum_sal3
  FROM emp;

EMP_ID ENAME DEPT_ID HIRE_DATE              SAL    SUM_SAL   SUM_SAL2   SUM_SAL3
------ ----- ------- -------------- ---------- ---------- ---------- ----------
   100 Stev       10 01-1月 -90           7000       7000       7000      36000
   101 Tom        20 21-9月 -89           2000       2000      10000      36000
   102 Mike       20 13-1月 -93           8000      10000      10000      36000
   120 John       50 18-7月 -96           1000       1000      19000      36000
   121 Joy        50 10-4月 -97           4000       5000      19000      36000
   123 Kate       50 10-10月-97           5000      10000      19000      36000
   124 Jess       50 16-11月-99           6000      16000      19000      36000
   122 Rich       50                      3000      19000      19000      36000
```

可以看出，emp 表总共有 10 条记录，该语句应用分析函数查询返回 10 条记录。另外请注意，sum_sal、sum_sal2、sum_sal3 这三个返回值各不相同，差异在于有无 partition by 的分

区和有无 order by ，这些都影响到了结果。这里最明显的感觉就是，这个 order by 并非单纯对结果进行排序，而是连返回值都不一样了。

再看一个最普通的聚合语句的查询结果，请看语句 2，如下：

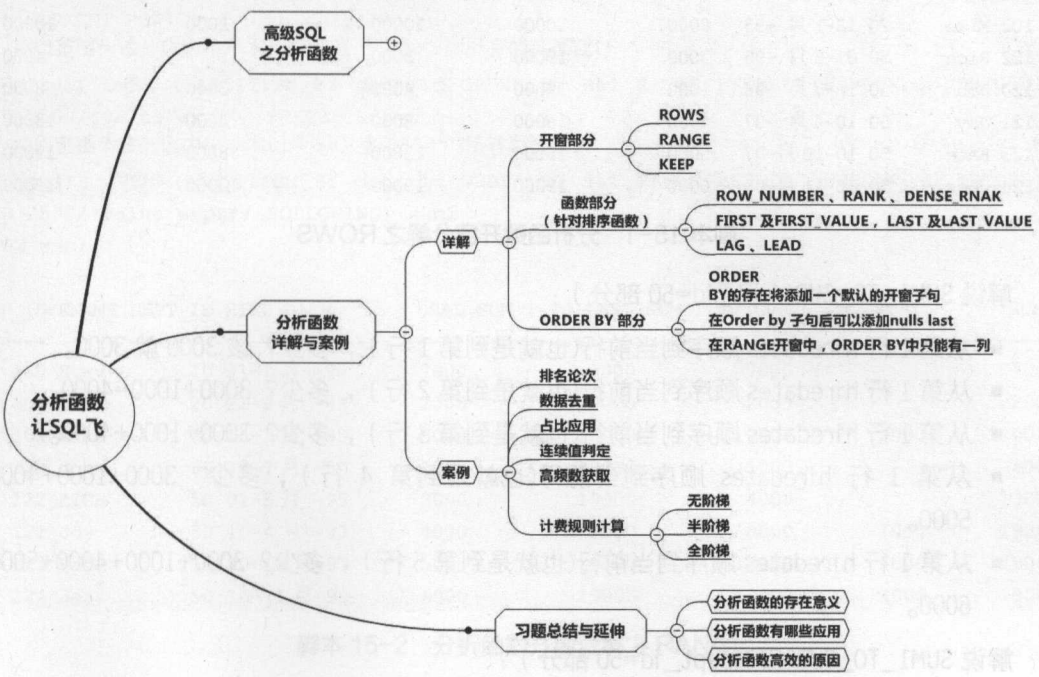
```
SELECT dept_id ,SUM(sal) FROM emp GROUP BY dept_id ORDER BY  dept_id;

DEPT_ID   SUM(SAL)
-------   --------
     10       7000
     20      10000
     50      19000
```

emp 表有 10 条记录，最终语句返回 3 条，因为 dept_id 就是分为 3 组，分别是编号为 10、20、50 的三个部门。order by 在这里显然就是一个让结果排序的动作，一点都不影响输出的值。

这里我们先做一个简单的解释，细节还请大家多做试验，多多揣摩：

- 分析函数是每组有多少行就返回多少行（好比多一个伪列）。而聚合函数是每组不管有多少行都被聚合成一行。
- 分析函数的 order by 和聚合函数的 order by 它们排序的概念不一样，分析函数的 order by 同时还标记从分组第一行到当前行的范围圈定，然后根据这个范围，依据这个顺序开始聚合累加。

15.2　分析函数详解与案例



15.2.1 学习详解

1. 开窗部分

（1）开窗分解之 ROWS

```
SELECT
 emp_id,ename,dept_id,hire_date,sal,
-- 以下均为首先按 dept_id 进行分组，其次按照 hire_date 进行排序，且所有统计不能跨越其所在分区，故不再重复
-- 窗口范围为该分区的第一行到该分区的最后一行，与 sum_sal_part 等同
 SUM(sal) OVER (PARTITION BY dept_id ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) sum_1_to_last,
-- 窗口范围为该分区的第一行到本行，与 sum_sal_part_order 等同
 SUM(sal) OVER (PARTITION BY dept_id ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) sum_1_to_cur,
-- 窗口范围为该分区的第一行到本行前一行，统计的是第一行到本行前一行薪资的累计
 SUM(sal) OVER (PARTITION BY dept_id ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND 1/*value_expr*/ PRECEDING) sum_1_to_curbef1,
 -- 窗口范围为该分区的第一行到本行后一行，统计的是第一行到本行后一行薪资的累计
 SUM(sal) OVER (PARTITION BY dept_id ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING) sum_1_to_curaft1
FROM emp order by dept_id,hire_date;
```

EMP_ID	ENAME	DEPT_ID	HIRE_DATE	SAL	SUM_1_TO_LAST	**SUM_1_TO_CUR**	SUM_1_TO_CURBEF1	SUM_1_TO_CURAFT1
100	Stev	10	01-1月 -90	7000	7000	7000		7000
101	Tom	20	21-9月 -89	2000	10000	2000		10000
102	Mike	20	13-1月 -93	8000	10000	10000	2000	10000
122	Rich	50	01-5月 -95	3000	19000	3000		4000
120	John	50	18-7月 -96	1000	19000	4000	3000	8000
121	Joy	50	10-4月 -97	4000	19000	8000	4000	13000
123	Kate	50	10-10月-97	5000	19000	13000	8000	19000
124	Jess	50	16-11月-99	6000	19000	19000	13000	19000

脚本 15-1　分析函数开窗分解之 ROWS

解说 SUM1_TO_CUR(dept_id=50 部分）：

- 从第 1 行 hiredates 顺序到当前行(也就是到第 1 行），多少？该 3000 就 3000。
- 从第 1 行 hiredates 顺序到当前行(也就是到第 2 行），多少？3000+1000=4000。
- 从第 1 行 hiredates 顺序到当前行(也就是到第 3 行），多少？3000+1000+4000。
- 从第 1 行 hiredates 顺序到当前行(也就是到第 4 行），多少？3000+1000+4000+5000。
- 从第 1 行 hiredates 顺序到当前行(也就是到第 5 行），多少？3000+1000+4000+5000+6000。

解说 SUM1_TO_CURAFT1(dept_id=50 部分）：

- 从第 1 行 hiredates 顺序到当前行后 1 行(也就是到第 2 行），多少？300+1000。
- 从第 1 行 hiredates 顺序到当前行后 1 行(也就是到第 3 行），多少？3000+1000+4000。
- 从第 1 行 hiredates 顺序到当前行后 1 行(也就是到第 4 行），多少？3000+1000+4000+5000。
- 从第 1 行 hiredates 顺序到当前行后 1 行(也就是到第 5 行），多少？3000+1000+4000+5000+6000。
- 从第 1 行 hiredates 顺序到当前行后 1 行(也就是到第 6 行，第 6 行没记录了，那结果和上一次一样）。

（2）开窗分解之 RANGE

- RANGE 窗口仅对 NUMBERS 和 DATES 起作用，因为不可能从 VARCHAR2 中增加或减去 *N*个单元。
- 在 RANGE 的开窗中，ORDER BY 中只能有一列；ROWS 开窗的 ORDER BY 可以有多列。

```
SELECT
 emp_id,ename,dept_id,hire_date,sal,
-- 后面均为以 dept_id 分组，再按 hire_date 排序，且所有统计不能跨分区，由于是逻辑范围，因此 PRECEDING 和 FOLLOWING 表达式有符号
-- 窗口范围为该分区的第一行到该分区的最后一行，与 sum_sal_part 等同,在非条件表达式中等同于 ROWS
 SUM(sal) OVER (PARTITION BY dept_id ORDER BY sal  RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) sum_1_to_last,
-- 窗口范围为该分区的第一行到本行，与 sum_sal_part_order 等同,在非条件表达式中等同于 ROWS
 SUM(sal) OVER (PARTITION BY dept_id ORDER BY sal  RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) sum_1_to_cur,
-- 窗口范围为该分区内小于本记录 sal 少 2500 的所有的薪资累计
 SUM(sal) OVER (PARTITION BY dept_id ORDER BY sal RANGE BETWEEN UNBOUNDED PRECEDING AND 2500/*value_expr*/ PRECEDING) sum1,
-- 窗口范围为该分区内小于本记录 sal 多 2500 的所有薪资累计
 SUM(sal) OVER (PARTITION BY dept_id ORDER BY sal RANGE BETWEEN UNBOUNDED PRECEDING AND 2500/*value_expr*/ FOLLOWING) sum2
FROM emp;

EMP_ID ENAME DEPT_ID HIRE_DATE        SAL SUM_1_TO_LAST SUM_1_TO_CUR       SUM1       SUM2
------ ----- ------- -------------- ----- ------------ ------------ ---------- ----------
   100 Stev       10 01-1月 -90      7000         7000         7000                  7000
   101 Tom        20 21-9月 -89      2000        10000         2000                  2000
   102 Mike       20 13-1月 -93      8000        10000        10000       2000      10000
   120 John       50 18-7月 -96      1000        19000         1000                  4000
   122 RICH       50 01-5月 -95      3000        19000         4000                 13000
   121 Joy        50 10-4月 -97      4000        19000         8000       1000      19000
   123 Kate       50 10-10月-97      5000        19000        13000       1000      19000
   124 Jess       50 16-11月-99      6000        19000        19000       4000      19000
```

脚本 15-2　分析函数开窗分解之 RANGE

解说 SUM1（dept_id=50 部分）：

- 比 1000-2500 少的数字有木有，没有。
- 比 3000-2500 少的数字有木有，没有，该组第 1 排的 1000 也比 500 大。
- 比 4000-2500 少的数字有木有，有啊，该组第 1 排的 1000 就比 1500 小。
- 比 5000-2500 少的数字有木有，有啊，该组第 1 排的 1000 就比 2500 小。
- 比 6000-2500 少的数字有木有，有啊，该组第 1 排和第 2 排哦。

解说 SUM2（dept_id=50 部分）：

- 比 1000+2500 少的数字有木有，有啊，1000 和 3000。
- 比 3000+2500 少的数字有木有，有啊，1 到 4 排都是。
- 比 4000+2500 少的数字有木有，有啊，该组都是。
- 比 5000+2500 少的数字有木有，有啊，该组都是。
- 比 6000+2500 少的数字有木有，有啊，该组都是。

（3）开窗分解之 KEEP

KEEP 为聚合函数的特殊关键字。当聚合函数 MIN、MAX、SUM、AVG、COUNT、VARIANCE 和 STDDEV 使用 KEEP 且和 DENSE_RANK FIRST /DENSE_RANK LAST 一起使用时，获取一组中排名第一或者排名最后的记录。必须由 order by 子句来排序。后面也可以接 over（）分析函数部分。Min（col2）keep（dense_rank first order by col1）保留按 col1 排名第一的 col2 的最小值。Min（col2）keep（dense_rank first order by col1）over (partition by col3) 按 col3 分组保留按 col1 排名各组第一的 col2 的最小值。

```
---需要注意的是，KEEP 只能与 DENSE_RANK FIRST、DENSE_RANK LAST 搭配使用。
SELECT emp_id,ename,dept_id,hire_date,sal,
 DENSE_RANK() OVER(PARTITION BY dept_id ORDER BY sal) DENSE_RANK,
 MIN(hire_date) KEEP (DENSE_RANK FIRST ORDER BY sal) OVER(PARTITION BY dept_id)
min_first,
 MIN(hire_date) KEEP (DENSE_RANK LAST ORDER BY sal) OVER(PARTITION BY dept_id)
min_last,
 MAX(hire_date) KEEP (DENSE_RANK FIRST ORDER BY sal) OVER(PARTITION BY dept_id)
max_first,
 MAX(hire_date) KEEP (DENSE_RANK LAST ORDER BY sal) OVER(PARTITION BY dept_id) max_last
 FROM emp;
EMP_ID ENAME DEPT_ID HIRE_DATE     SAL DENSE_RANK MIN_FIRST    MIN_LAST     MAX_FIRST
MAX_LAST
------ ----- ------- ---------------------------------------------------------------
  100 Stev     10   01-1月 -90     7000    1   01-1月 -90   01-1月 -90   01-1月 -90   01-1月 -90
  101 Tom      20   21-9月 -89     2000    1   21-9月 -89   13-1月 -93   21-9月 -89   13-1月 -93
  102 Mike     20   13-1月 -93     8000    2   21-9月 -89   13-1月 -93   21-9月 -89   13-1月 -93
  120 John     50   18-7月 -96     1000    1   18-7月 -96   16-11月-99   10-4月 -97   16-11月-99
  121 Joy      50   10-4月 -97     1000    1   18-7月 -96   16-11月-99   10-4月 -97   16-11月-99
  122 Rich     50   01-5月 -95     3000    2   18-7月 -96   16-11月-99   10-4月 -97   16-11月-99
```

```
  123 Kate     50   10-10月-97     5000    3   18-7月 -96   16-11月-99   10-4月 -97   16-11月-99
       ss      50   16-11月-99     6000    4   18-7月 -96   16-11月-99   10-4月 -97   16-11月-99
```

脚本 15-3　分析函数开窗分解之 KEEP

2. 函数部分（针对排序函数）

准备环境：

```
drop table emp purge;

CREATE TABLE emp
(
  emp_id    NUMBER(6),
  ename  VARCHAR2(45),
  dept_id   NUMBER(4),
  hire_date DATE,
  sal    NUMBER(8,2)
);

--创建 emp 数据
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (101, 'Tom',     20,
TO_DATE('21-09-1989', 'DD-MM-YYYY'), 2000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (102, 'Mike',    20,
TO_DATE('13-01-1993', 'DD-MM-YYYY'), 8000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (120, 'John',    50,
TO_DATE('18-07-1996', 'DD-MM-YYYY'), 1000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (121, 'Joy',     50,
TO_DATE('10-04-1997', 'DD-MM-YYYY'), 4000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (122, 'Rich',    50,
TO_DATE('01-05-1995', 'DD-MM-YYYY'), 4000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (123, 'Kate',    50,
TO_DATE('10-10-1997', 'DD-MM-YYYY'), 4000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (124, 'Jess',    50,
TO_DATE('16-11-1999', 'DD-MM-YYYY'), 6000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (100, 'Stev',    10,
TO_DATE('01-01-1990', 'DD-MM-YYYY'), 7000);
COMMIT;

set linesize 2000
set pagesize 2000
col emp_id format 999
col dept_id format 99
col sal format 9999
col ename format a5
col hire_date FORMAT DATE
col fir_val FORMAT a10
col fir_val_desc FORMAT a10
col last_val FORMAT a10
col last_val_desc FORMAT a10
```

（1）ROW_NUMBER、RANK 及 DENSE_RANK

说到分析函数，不得不提到的是分组排名的 3 个函数：row_number、rank、dense_rank。三者的差异为：

row_number 函数返回一个唯一的值，当碰到相同数据时，排名按照记录集中记录的顺序依次递增。

dense_rank 函数返回一个唯一的值，除非碰到相同数据时，此时所有相同数据的排名都是一样的。

rank 函数返回一个唯一的值，除非遇到相同的数据时，此时所有相同数据的排名是一样的，同时会在最后一条相同记录和下一条不同记录的排名之间空出排名。

试验如下：

```
SELECT
 emp_id,ename,dept_id,hire_date,sal,
 ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY sal) AS row_number,
 RANK() OVER (PARTITION BY dept_id ORDER BY sal) AS rank,
 DENSE_RANK() OVER (PARTITION BY dept_id ORDER BY sal) AS dense_rank
FROM emp;

EMP_ID ENAME DEPT_ID HIRE_DATE         SAL ROW_NUMBER       RANK DENSE_RANK
------ ----- ------- -------------- ----- ---------- ---------- ----------
   100 Stev       10 01-1月 -90      7000          1          1          1
   101 Tom        20 21-9月 -89      2000          1          1          1
   102 Mike       20 13-1月 -93      8000          2          2          2
   120 John       50 18-7月 -96      1000          1          1          1
   123 Kate       50 10-10月-97      4000          2          2          2
   122 Rich       50 01-5月 -95      4000          3          2          2
   121 Joy        50 10-4月 -97      4000          4          2          2
   124 Jess       50 16-11月-99      6000          5          5          3
```

脚本 15-4　分析函数的函数部分（ROW_NUMBER 等）

以 DEPT_ID=50 来看，ROW_NUMBER 这个列取值为 1、2、3、4、5，而 RANK 却为 1、2、2、2、5。这是为啥呢？因为这几行对应的 SAL 都是 4000，并列了，RANK 的差异就是并列要取一样排名。最后看看 DENSE_RANK 列，取值为 1、2、2、2、3，很显然它和 RANK 的差异就在最后一行上，DENSE_RANK 是没有跳号的，并列名次随后，依然是加 1。

（2）FIRST_VALUE 与 LAST_VALUE 等

接下来要说 FIRST_VALUE 和 LAST_VALUE 两个函数，其中 FIRST_VALUE 返回一个排序数据集合的第一行，而 LAST_VALUE 则返回一个排序数据集合的最后一行。

```
SELECT
 emp_id,ename,dept_id,hire_date,sal,
 FIRST_VALUE(ename) OVER(PARTITION BY dept_id ORDER BY sal ) AS fir_val,
```

```
 FIRST_VALUE(ename) OVER(PARTITION BY dept_id ORDER BY sal DESC) AS fir_val_desc,
 LAST_VALUE(ename) OVER(PARTITION BY dept_id ORDER BY sal ) AS last_val,
 LAST_VALUE(ename) OVER(PARTITION BY dept_id ORDER BY sal DESC) AS last_val_desc
FROM emp;

EMP_ID ENAME DEPT_ID HIRE_DATE          SAL FIR_VAL    FIR_VAL_DE LAST_VAL   LAST_VAL_D
------ ----- ------- -------------- ----- ---------- ---------- ---------- ----------
   100 Stev        10 01-1月 -90      7000 Stev       Stev       Stev       Stev
   101 Tom         20 21-9月 -89      2000 Tom        Mike       Tom        Tom
   102 Mike        20 13-1月 -93      8000 Tom        Mike       Mike       Mike
   120 John        50 18-7月 -96      1000 John       Jess       John       John
   121 Joy         50 10-4月 -97      4000 John       Jess       Rich       Rich
   123 Kate        50 10-10月-97      4000 John       Jess       Rich       Rich
   122 Rich        50 01-5月 -95      4000 John       Jess       Rich       Rich
   124 JEss        50 16-11月-99      6000 John       Jess       Jess       Jess
```

脚本 15-5　分析函数的函数部分（FIRST_VALUE 等）

请以 DEPT_ID=50，来观察 FIR_VAL 和 LAST_VAL 这两列的取值，体会其中的差异。

（3）LAG 与 LEAD 等

这两个函数也是分析函数中极其重要、常用的函数，其中 LEAD（field，n）按 over 里面的规则排序，并取排序的当前记录 field 的下 *n* 个数值，而 LAG 则相反。

```
SELECT
 emp_id,ename,dept_id,hire_date,sal,
 LAG(sal) OVER (ORDER BY hire_date) AS prev_sal1,
 LEAD(sal) OVER (ORDER BY hire_date) AS next_sal1,
 LAG(sal, 1, 0) OVER (ORDER BY hire_date) AS prev_sal2,
 LEAD(sal, 1,0) OVER (ORDER BY hire_date) AS next_sal2,
 LAG(sal, 1, 0) OVER (partition BY dept_id ORDER BY hire_date) AS prev_sal3,
 LEAD(sal, 1,0) OVER (partition BY dept_id ORDER BY hire_date) AS next_sal3,
 LAG(sal, 2, 999) OVER (partition BY dept_id ORDER BY hire_date) AS prev_sal4,
 LEAD(sal, 2,999) OVER (partition BY dept_id ORDER BY hire_date) AS next_sal4
FROM emp;
EMP_ID ENAME DEPT_ID HIRE_DATE   SAL   PREV_SAL1 NEXT_SAL1 PREV_SAL2 NEXT_SAL2 PREV_SAL3 NEXT_SAL3 PREV_SAL4
NEXT_SAL4
-----------------------------------------------------------------------------------------------------------
   101 Tom     20   21-9月 -89  2000          7000         0       7000          0       8000        999        999
   100 Stev    10   01-1月 -90  7000    2000  8000      2000       8000          0          0        999        999
   102 Mike    20   13-1月 -93  8000    7000  3000      7000       3000       2000          0        999        999
   122 Rich    50   01-5月 -95  3000    8000  1000      8000       1000          0       1000        999       4000
   120 John    50   18-7月 -96  1000    3000  4000      3000       4000       3000       4000        999       5000
   121 Joy     50   10-4月 -97  4000    1000  5000      1000       5000       1000       5000       3000       6000
   123 Kate    50   10-10月-97  5000    4000  6000      4000       6000       4000       6000       1000        999
   124 Jess    50   16-11月-99  6000    5000                5000          0       5000          0       4000        999
```

脚本 15-6　分析函数的函数部分（LAG 与 LEAD 等）

请先认真观察 PREV_SAL1 和 NEXT_SAL1 的取值，并和 SAL 列进行比较，就能发现其中的奥妙了。

3. ORDER BY 部分

分析函数的 ORDER BY 部分是一个分析函数最容易犯错的地方，其和普通 SQL 带的 ORDER BY 的含义是有很大差异的。分析函数中使用 ORDER BY 时，将添加一个默认的开窗子句，这意味着计算中所使用的行的集合是当前分区中当前行和前面所有行，在没有 ORDER BY 时，默认的窗口是全部的分区。我们来做试验体会一下差异。

准备环境：

```
drop table emp purge;

CREATE TABLE emp
(
  emp_id    NUMBER(6),
  ename  VARCHAR2(45),
  dept_id   NUMBER(4),
  hire_date DATE,
  sal    NUMBER(8,2)
);

--创建 emp 数据
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (101, 'Tom',      20,
TO_DATE('21-09-1989', 'DD-MM-YYYY'), 2000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (102, 'Mike',     20,
TO_DATE('13-01-1993', 'DD-MM-YYYY'), 8000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (120, 'John',     50,
TO_DATE('18-07-1996', 'DD-MM-YYYY'), 1000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (121, 'Joy',      50,
TO_DATE('10-04-1997', 'DD-MM-YYYY'), 4000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (122, 'Rich',   50,  NULL,
3000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (123, 'Kate',     50,
TO_DATE('10-10-1997', 'DD-MM-YYYY'), 5000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (124, 'Jess',     50,
TO_DATE('16-11-1999', 'DD-MM-YYYY'), 6000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (100, 'Stev',     10,
TO_DATE('01-01-1990', 'DD-MM-YYYY'), 7000);
COMMIT;
```

ORDER BY 部分的代码示例：

```
set linesize 2000
set pagesize 2000
col emp_id format 999
col dept_id format 99
col ename format a5

SELECT
```

```
  emp_id,ename,dept_id,hire_date,sal,
  SUM(sal) OVER (PARTITION BY dept_id ORDER BY hire_date) sum_sal1,
  SUM(sal) OVER (PARTITION BY dept_id ORDER BY hire_date DESC) sum_sal2,
  SUM(sal) OVER (PARTITION BY dept_id ORDER BY hire_date DESC nulls LAST) sum_sal3,
  SUM(sal) OVER (PARTITION BY dept_id ) sum_sal4,
  SUM(sal) OVER ( ) sum_sal5
  FROM emp;
```

EMP_ID	ENAME	DEPT_ID	HIRE_DATE	SAL	SUM_SAL1	SUM_SAL2	SUM_SAL3	SUM_SAL4	SUM_SAL5
100	Stev	10	01-1月 -90	7000	7000	7000	7000	7000	36000
101	Tom	20	21-9月 -89	2000	2000	10000	10000	10000	36000
102	Mike	20	13-1月 -93	8000	10000	8000	8000	10000	36000
120	John	50	18-7月 -96	1000	1000	19000	16000	19000	36000
121	Joy	50	10-4月 -97	4000	5000	18000	15000	19000	36000
123	Kate	50	10-10月-97	5000	10000	14000	11000	19000	36000
124	Jess	50	16-11月-99	6000	16000	9000	6000	19000	36000
122	Rich	50		3000	19000	3000	19000	19000	36000

脚本 15-7　分析函数的 order by 部分

还是以 DEPT_ID=50 的列为例，请结合比对 SAL 列，认真观察 SUM_SAL1 列的取值，我们会发现 SUM_SAL1 值是根据 SAL 的排序依次累加的，5000=1000+4000，而 10000=1000+4000+5000，等等。但是我们观察没有 ORDER BY 的 SUM_SAL4 列，19000 的取值显然是 DEPT_ID=50 所在列的 SAL 值的全部累加。

15.2.2　案例分享

1. 排名论次

环境准备：

```
drop table emp purge;
CREATE TABLE emp
(
  emp_id    NUMBER(6),
  ename  VARCHAR2(45),
  dept_id   NUMBER(4),
  hire_date DATE,
  sal    NUMBER(8,2)
);
--创建 emp 数据
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (101, 'Tom',     20,
TO_DATE('21-09-1989', 'DD-MM-YYYY'), 2000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (102, 'Mike',    20,
TO_DATE('13-01-1993', 'DD-MM-YYYY'), 8000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (120, 'John',    50,
TO_DATE('18-07-1996', 'DD-MM-YYYY'), 1000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (121, 'Joy',     50,
```

```
TO_DATE('10-04-1997', 'DD-MM-YYYY'), 1000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (122, 'Rich',    50,
TO_DATE('01-05-1995', 'DD-MM-YYYY'), 4000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (123, 'Kate',    50,
TO_DATE('10-10-1997', 'DD-MM-YYYY'), 4000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (124, 'Jess',    50,
TO_DATE('16-11-1999', 'DD-MM-YYYY'), 7000);
INSERT INTO emp (emp_id, ename, dept_id, hire_date, sal) VALUES (100, 'Stev',    10,
TO_DATE('01-01-1990', 'DD-MM-YYYY'), 7000);
COMMIT;
set linesize 2000
set pagesize 2000
col emp_id format 999
col dept_id format 99
col sal format 9999
col ename format a5
col hire_date FORMAT DATE

SELECT * from emp;
EMP_ID ENAME DEPT_ID HIRE_DATE         SAL
------------------ ----------------------
   101 Tom         20 21-9月 -89       2000
   102 Mike        20 13-1月 -93       8000
   120 John        50 18-7月 -96       1000
   121 Joy         50 10-4月 -97       1000
   122 Rich        50 01-5月 -95       4000
   123 Kate        50 10-10月-97       4000
   124 Jess        50 16-11月-99       7000
   100 Stev        10 01-1月 -90       7000
```

语句 1（用普通语句来实现找各组收入最低的），如下：

```
WITH t as
 (SELECT dept_id, min(sal) as min_sal FROM emp GROUP BY dept_id)
select emp.emp_id, emp.ename, emp.dept_id, emp.hire_date,emp.sal
  from emp, t
 where emp.dept_id = t.dept_id
   and emp.sal = t.min_sal;

EMP_ID ENAME DEPT_ID HIRE_DATE         SAL
-----------------------------------------
   101 Tom         20 21-9月 -89       2000
   121 Joy         50 10-4月 -97       1000
   120 John        50 18-7月 -96       1000
   100 Stev        10 01-1月 -90       7000
普通 SQL 写法的执行计划
----------------------
Plan hash value: 2230095667
---------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
```

```
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     1 |    98 |     8  (25)| 00:00:01 |
|*  1 |  HASH JOIN           |      |     1 |    98 |     8  (25)| 00:00:01 |
|   2 |   TABLE ACCESS FULL  | EMP  |     8 |   576 |     3   (0)| 00:00:01 |
|   3 |   VIEW               |      |     8 |   208 |     4  (25)| 00:00:01 |
|   4 |    HASH GROUP BY     |      |     8 |   208 |     4  (25)| 00:00:01 |
|   5 |     TABLE ACCESS FULL| EMP  |     8 |   208 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
         14  consistent gets
          0  physical reads
          0  redo size
        786  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          4  rows processed
```

脚本 15-8　用普通 SQL 实现各组最低收入统计

语句 2，（用分析函数实现找各组收入最低的）：

```
Set autotrace on
SELECT  emp_id, ename, dept_id, hire_date,sal
  FROM (SELECT emp.*,
               dense_rank() OVER(PARTITION BY dept_id ORDER BY sal ) AS  N
          FROM emp)
 WHERE N = 1;

EMP_ID ENAME DEPT_ID HIRE_DATE         SAL
------ ----- ------- -------------- -----
   100 Stev        10 01-1月 -90      7000
   101 Tom         20 21-9月 -89      2000
   120 John        50 18-7月 -96      1000
   121 Joy         50 10-4月 -97      1000
------------------------------------------
Plan hash value: 3291446077
---------------------------
| Id  | Operation                | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |     8 |   680 |     4  (25)| 00:00:01 |
|*  1 |  VIEW                    |      |     8 |   680 |     4  (25)| 00:00:01 |
|*  2 |   WINDOW SORT PUSHED RANK|      |     8 |   576 |     4  (25)| 00:00:01 |
|   3 |    TABLE ACCESS FULL     | EMP  |     8 |   576 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
统计信息
```

```
-------      .
          0  recursive calls
          0  db block gets
          7  consistent gets
          0  physical reads
          0  redo size
        786  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
 rows processed
```

脚本 15-9　用分析函数实现各组最低收入统计

比较可以看出，写法 1 的 consistent gets 为 12，而分析函数的写法 2 的 consistent gets 为 7，显然分析函数的性能更高。再仔细观察执行计划我们可以看出，分析函数的 EMP 表仅被访问一次，而普通的写法 1 则访问 2 次。

2. 数据去重

环境准备：

```
DROP TABLE t purge ;
CREATE TABLE t AS SELECT * FROM dba_objects WHERE rownum<=10;
UPDATE t SET object_id=rownum;
UPDATE t SET object_id=3 WHERE object_id<=3;
UPDATE t SET object_id=4 WHERE object_id>=4 AND object_id<=6;
COMMIT;
```

方法 1：

```
SET autotrace ON
SET linesize 2000
SET pagesize 2000

delete from t
where rowid <
(select max(rowid) from t t2
where t.object_id = t2.object_id
);

已删除 4 行。

执行计划
--------
Plan hash value: 3804998351
---------------------------
| Id  | Operation             | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
```

```
|   0 | DELETE STATEMENT      |          |     1 |    50 |     8  (25)| 00:00:01 |
|   1 |  DELETE               | T        |       |       |            |          |
|*  2 |   HASH JOIN           |          |     1 |    50 |     8  (25)| 00:00:01 |
|   3 |    VIEW               | VW_SQ_1  |     6 |   150 |     4  (25)| 00:00:01 |
|   4 |     SORT GROUP BY     |          |     6 |   150 |     4  (25)| 00:00:01 |
|   5 |      TABLE ACCESS FULL| T        |     6 |   150 |     3   (0)| 00:00:01 |
|   6 |    TABLE ACCESS FULL  | T        |     6 |   150 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("T"."OBJECT_ID"="ITEM_1")
       filter(ROWID<"MAX(ROWID)")

Note
-----
   - dynamic sampling used for this statement (level=2)


统计信息
--------
         50  recursive calls
          6  db block gets
         40  consistent gets
          0  physical reads
          0  redo size
        684  bytes sent via SQL*Net to client
        659  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          4  rows processed
```

脚本 15-10　数据去重的普通写法

方法 2（注意，这里的写法和上面的并不完全等价，上面是随便删除，保留 rowid 最大一条。下面是保留时间最新的，上面既然可以随便删了，下面的写法肯定也是符合要求的。其中 row_number 必须要有 order by 关键字）：

```
delete t
 where rowid in (select rid
                   from (select rowid rid,
                                row_number()  over(partition  by  object_id  ORDER  by
created desc) rn
                           from t)
                  where rn > 1);
已删除 4 行。

执行计划
```

```
--------
Plan hash value: 2005107446
---------------------------
| Id |Operation                    | Name     |Rows  |Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0|DELETE STATEMENT             |          |    1 |   24 |     6  (34)| 00:00:01 |
|   1| DELETE                      | T        |      |      |            |          |
|   2|  NESTED LOOPS               |          |    1 |   24 |     6  (34)| 00:00:01 |
|   3|   VIEW                      | VW_NSO_1 |    6 |   72 |     4  (25)| 00:00:01 |
|   4|    SORT UNIQUE              |          |    1 |  150 |            |          |
|*  5|     VIEW                    |          |    6 |  150 |     4  (25)| 00:00:01 |
|   6|      WINDOW SORT            |          |    6 |  204 |     4  (25)| 00:00:01 |
|   7|       TABLE ACCESS FULL     | T        |    6 |  204 |     3   (0)| 00:00:01 |
|   8|   TABLE ACCESS BY USER ROWID| T        |    1 |   12 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - filter("RN">1)
Note
-----
   - dynamic sampling used for this statement (level=2)
统计信息
--------
         48  recursive calls
          6  db block gets
         38  consistent gets
          0  physical reads
          0  redo size
        684  bytes sent via SQL*Net to client
        813  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          3  sorts (memory)
          0  sorts (disk)
          4  rows processed
```

脚本 15-11　数据去重的分析函数写法

虽然从脚本 15-10 和脚本 15-11 的结果比较来看，第一种方法效率也还可以，不过如果是多个字段重复，第一种方法就显得有些麻烦了，而且如果需求是根据日期保留重复记录中最新的一个，则第一种方法也显得非常繁琐，此时还是建议用分析函数的第二种写法。

不过这里要记住的一点是，要是表记录非常大，比如有几千万条记录，而重复记录的数量又特别巨大，这时候可以考虑直接把不重复的数据单独建出来，也是一个灵活的好办法。

3. 占比应用

```
SELECT
 emp_id,ename,dept_id,hire_date,sal,
 ratio_to_report(sal) OVER () as pct11,
 ratio_to_report(sal) OVER (partition by dept_id) as pct2
FROM emp;
```

```
EMP_ID ENAME DEPT_ID HIRE_DATE          SAL       PCT1L       PCT2
------ ----- ------- -------------- ----- ---------- ----------
   100 Stev       10 01-1月 -90      7000 .194444444          1
   101 Tom        20 21-9月 -89      2000 .055555556         .2
   102 Mike       20 13-1月 -93      8000 .222222222         .8
   124 Jess       50 16-11月-99      6000 .166666667 .315789474
   123 Kate       50 10-10月-97      5000 .138888889 .263157895
   122 Rich       50 01-5月 -95      3000 .083333333 .157894737
   120 John       50 18-7月 -96      1000 .027777778 .052631579
         y        50 10-4月 -97      4000 .111111111 .210526316
```

脚本 15-12　分析函数实现占比应用

4. 连续值判定

需求描述及环境准备：

```
drop table t purge;
create table t (id1 int,id2 int ,id3 int);
insert into t (id1 ,id2,id3) values (1,45,89);
insert into t (id1 ,id2,id3) values (2,45,89);
insert into t (id1 ,id2,id3) values (3,45,89);
insert into t (id1 ,id2,id3) values (8,45,89);
insert into t (id1 ,id2,id3) values (12,45,89);
insert into t (id1 ,id2,id3) values (36,45,89);
insert into t (id1 ,id2,id3) values (22,45,89);
insert into t (id1 ,id2,id3) values (23,45,89);
insert into t (id1 ,id2,id3) values (89,45,89);
insert into t (id1 ,id2,id3) values (92,45,89);
insert into t (id1 ,id2,id3) values (91,45,89);
insert into t (id1 ,id2,id3) values (90,45,89);
commit;

SQL> select * from t;
       ID1        ID2        ID3
---------- ---------- ----------
         1         45         89
         2         45         89
         3         45         89
         8         45         89
        12         45         89
        36         45         89
        22         45         89
        23         45         89
        89         45         89
        92         45         89
        91         45         89
        90         45         89
```

```
--需求 1：将连续数据查找出来，要达到如下效果：
ID1 ID2 ID3
-----------
1 45 89
2 45 89
3 45 89
22 45 89
23 45 89
89 45 89
90 45 89
91 45 89
92 45 89

--需求 2： 要求查出连续数据，并且要写出最小值和最大值及连续的个数，效果如下：
1   3   3
22  23  2
89  92  4
```

用分析函数分步骤完成：

```
select t.*,
               lag(id1,1,0) over(order by id1) av, ---构造出伪列 av
               lead(id1,1,0) over(order by id1) ev ---构造出伪列 ev
          from t;


      ID1         ID2         ID3          AV          EV
--------- ---------- ---------- ---------- ----------
        1          45          89          0           2
        2          45          89          1           3
        3          45          89          2           8
        8          45          89          3          12
       12          45          89          8          22
       22          45          89         12          23
       23          45          89         22          36
       36          45          89         23          89
       89          45          89         36          90
       90          45          89         89          91
       91          45          89         90          92
       92          45          89         91           0

--思考中间环节（构造）

SELECT id1,
       id2,
       id3,
       ROW_NUMBER() OVER(ORDER BY id1) - ID1 AS group_id
FROM t                 ;

       ID1         ID2         ID3   GROUP_ID
```

```
---------- ---------- ---------- ----------
         1         45         89          0
         2         45         89          0
         3         45         89          0
         8         45         89         -4
        12         45         89         -7
        22         45         89        -16
        23         45         89        -16
        36         45         89        -28
        89         45         89        -80
        90         45         89        -80
        91         45         89        -80
        92         45         89        -80

--在构造的中间环节的基础上，实现了需求 1

SELECT id1, id2, id3
  FROM (
        SELECT id1, id2, id3, COUNT(*) OVER(PARTITION BY group_id) CNT
          FROM (
                SELECT id1,
                       id2,
                       id3,
                       ROW_NUMBER() OVER(ORDER BY id1) - ID1 AS group_id
                  FROM t
                )
        )
 WHERE CNT > 1
 ORDER BY id1;


--同样在构造的中间环节的基础上，实现了需求 2

SELECT MIN(id1), MAX(id1), COUNT(*)
  FROM (SELECT id1,
               id2,
               id3,
               ROW_NUMBER() OVER(ORDER BY id1) - ID1 AS group_id
          FROM t)
HAVING COUNT(*) > 1
 GROUP BY group_id
 ORDER BY 1;
```

脚本 15-13　分析函数实现连续值判定

这里向大家说明一下，分析函数的最大优势就是可以进行数据的多次构造、利用。因此分步实现再逐步合并是分析函数解决复杂 SQL 逻辑的利器。

5. 高频数获取

需求描述：

```
SELECT sal,COUNT(*) repeat_num
            FROM emp
           GROUP BY sal;

       SAL REPEAT_NUM
---------- ----------
      1000          1
      4000          3
      2000          1
      8000          1
      6000          1
      7000          1
```

这里需求就是希望能找到这个 4000 的工资，这是频率最高的。

```
SET autotrace ON
SET linesize 1000
SET pagesize 2000

SELECT sal
  FROM
  (
    SELECT sal,RANK() OVER(ORDER BY repeat_num DESC) rank_repeat_num
          FROM
          (
            SELECT sal,COUNT(*) repeat_num
              FROM emp
             GROUP BY sal
          )
  )
 WHERE rank_repeat_num=1;
  SAL
-----
 4000

用分析函数写法的执行计划
------------------------
Plan hash value: 214435687
-------------------------
| Id  | Operation                | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |     8 |   208 |     5  (40)| 00:00:01 |
|*  1 |  VIEW                    |      |     8 |   208 |     5  (40)| 00:00:01 |
|*  2 |   WINDOW SORT PUSHED RANK|      |     8 |   104 |     5  (40)| 00:00:01 |
|   3 |    HASH GROUP BY         |      |     8 |   104 |     5  (40)| 00:00:01 |
|   4 |     TABLE ACCESS FULL    | EMP  |     8 |   104 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("RANK_REPEAT_NUM"=1)
   2 - filter(RANK() OVER ( ORDER BY COUNT(*) DESC )<=1)
统计信息
--------
          0  recursive calls
          0  db block gets
          7  consistent gets
          0  physical reads
          0  redo size
        418  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 15-14　分析函数实现高频数获取

不用分析函数，写法更麻烦且性能更差，如下：

```
select sal
  from (SELECT sal, COUNT(*) as repeat_num FROM emp GROUP BY sal) t
 where t.repeat_num =
       (select max(repeat_num)
          from (SELECT sal, COUNT(*) as repeat_num FROM emp GROUP BY sal));

  SAL
-----
 4000

用普通写法的执行计划
--------------------
Plan hash value: 3118377657
---------------------------
| Id  | Operation             | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |     8 |   104 |     4  (25)| 00:00:01 |
|*  1 |  FILTER               |      |       |       |            |          |
|   2 |   HASH GROUP BY       |      |     8 |   104 |     4  (25)| 00:00:01 |
|   3 |    TABLE ACCESS FULL  | EMP  |     8 |   104 |     3   (0)| 00:00:01 |
|   4 |   SORT AGGREGATE      |      |     1 |    13 |            |          |
|   5 |    VIEW               |      |     8 |   104 |     4  (25)| 00:00:01 |
|   6 |     SORT GROUP BY     |      |     8 |   104 |     4  (25)| 00:00:01 |
|   7 |      TABLE ACCESS FULL| EMP  |     8 |   104 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(COUNT(*)= (SELECT MAX("REPEAT_NUM") FROM  (SELECT "SAL"
             "SAL",COUNT(*) "REPEAT_NUM" FROM "EMP" "EMP" GROUP BY "SAL")
             "from$_subquery$_003"))
```

```
统计信息
--------
          0  recursive calls
          0  db block gets
         14  consistent gets
          0  physical reads
          0  redo size
        418  bytes sent via SQL*Net to client
        415  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
 rows processed
```

脚本 15-15　普通写法实现高频数获取的不足之处

观察分析函数的写法脚本 15-14 的逻辑读为 7，而普通写法的脚本 15-15 的逻辑读为 14，性能差异明显，大家认真观察可以发现原因，脚本 15-15 的 EMP 表被访问了 2 次，而脚本 15-14 由于分析函数自身的内部优化，让 EMP 表仅被访问 1 次。

6. 计费规则设计

（1）需求描述

结算规则配置方案：

- 用 IF_jieti 字段来区分是否是阶梯，0 为非阶梯，1 为半阶梯，2 为全阶梯。
- 用 price_type 字段来区分算法是百分比还是固定值 （无论全阶梯、半阶梯还是非阶梯算法，最终都是离不开百分比和固定值选择）。
- IF_jieti 字段和 price_type 字段的组合可以定义出全部的配置类型。

（2）名词解释

全阶梯就是比如 0～50 不分成，给 0 元；50～100，4∶6 开；101～200，5∶5 开；201 以上 6∶4 开。

这个时候有 300 元，是 0.4×100+100×0.5+100×0.6=150，这个就是全阶梯（并非 0.4×50+100×0.5+100×0.6）。

如果这个时候是 20 元呢，那 SP 的钱就是 0，没得分！

半阶梯是 300 元总金额不要再去截断了，直接乘以落在那段的比例：300×0.6=180。

以下脚本建立配置表中暂且配置了 4 个 SP，分别为 a、b、c、d，分别对应以下四个类型。

测试脚本全部扔到 COMMAND 下执行，然后看结果就好了！

```
drop table test_rule purge;
create table TEST_RULE
(
```

```
  IF_jieti  number,
  SP_NAME    VARCHAR2(10),
  PRICE_TYPE NUMBER,
  SP_PARAM   NUMBER,
  TEL_PARAM  NUMBER,
  TOTAL1     NUMBER,
  TOTAL2     NUMBER
);
-- Add comments to the columns
comment on column TEST_RULE.IF_JIETI
  is '用来判断是否为阶梯，0 为非阶梯，1 为半阶梯（不需分段算），2 为全阶梯（需分段算）';
comment on column TEST_RULE.SP_NAME
  is 'sp 名';
comment on column TEST_RULE.PRICE_TYPE
  is '分成类型（1 表示比率，2 为固定金额）';
comment on column TEST_RULE.SP_PARAM
  is 'sp 比率或金额';
comment on column TEST_RULE.TEL_PARAM
  is '电信比率或金额';
comment on column TEST_RULE.TOTAL1
  is '金额范围开始';
comment on column TEST_RULE.TOTAL2
  is '金额范围开始';

insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (0,'a', 1, 40, 60, null, null);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (0,'b', 2, 800, null, null, null);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (1,'c', 1, 30, 70, 0, 500);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (1,'c', 1, 40, 60, 501, 1000);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (1,'c', 1, 50, 50, 1001, 2000);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (1,'c', 1, 60, 40, 2001, 9999999);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (2,'d', 2, 0, null, 0, 500);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (2,'d', 1, 40, 60, 501, 1000);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (2,'d', 1, 50, 50, 1001, 2000);
insert into TEST_RULE (if_jieti,sP_NAME, PRICE_TYPE, SP_PARAM, TEL_PARAM, TOTAL1,
TOTAL2) values (2,'d', 1, 60, 40, 2001, 9999999);
commit;

--略去一堆字段
drop table ticket purge;
create table TICKET
(
  ID_NAME     number,
```

```
  SP_NAME     VARCHAR2(10),
  TOTAL_PRICE NUMBER
);

insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (121,'a', 100);
insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (222,'b', 1000);
insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (387,'c', 100 );
insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (645,'c', 600 );
insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (555,'c', 1200);
insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (987,'d', 100 );
insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (333,'d', 600 );
insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (221,'d', 1200);
insert into TICKET (ID_NAME,SP_NAME, TOTAL_PRICE) values (528,'d', 5000);
commit;

SP_NAME    TOTAL_PRICE   SP_PRICE TOTAL_PRICE       PAID
---------- ----------- ---------- ---------- ----------
a                  100                             40
b                 1000                            800
c                  100                             30
c                  600                            240
c                 1200                            600
d                  100                              0
d                  600                            240
d                 1200                            500
d                 5000                           2700
```

脚本 15-16　计费规则案例的环境准备

（3）分步实现

分解思路，步骤 1：

```
SET linesize 2000
SET pagesize 2000

select t.* ,LAG(total1) OVER (PARTITION BY SP_NAME ORDER BY TOTAL1) start_val from
test_rule t;
  IF_JIETI SP_NAME    PRICE_TYPE   SP_PARAM  TEL_PARAM     TOTAL1     TOTAL2  START_VAL
---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
         0 a                   1         40         60
         0 b                   2        800
         1 c                   1         30         70          0        500
         1 c                   1         40         60        501       1000          0
         1 c                   1         50         50       1001       2000        501
         1 c                   1         60         40       2001    9999999       1001
         2 d                   2          0                   0        500
         2 d                   1         40         60        501       1000          0
         2 d                   1         50         50       1001       2000        501
         2 d                   1         60         40       2001    9999999       1001
已选择 10 行。
```

脚本 15-17　计费规则案例实现分解 1

步骤 2：

```
SELECT r.*, DECODE(LAG(total1) OVER (PARTITION BY SP_NAME ORDER BY TOTAL1),0,1,total1)
AS start_val
         FROM TEST_RULE r;

  IF_JIETI SP_NAME    PRICE_TYPE   SP_PARAM  TEL_PARAM     TOTAL1     TOTAL2  START_VAL
---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
         0 a                   1         40         60
         0 b                   2        800
         1 c                   1         30         70          0        500          0
         1 c                   1         40         60        501       1000          1
         1 c                   1         50         50       1001       2000       1001
         1 c                   1         60         40       2001    9999999       2001
         2 d                   2          0                    0        500          0
         2 d                   1         40         60        501       1000          1
         2 d                   1         50         50       1001       2000       1001
         2 d                   1         60         40       2001    9999999       2001

已选择 10 行。
```

脚本 15-18　计费规则案例实现分解 2

步骤 3：

```
SELECT r.*
      ,SUM((TOTAL2-start_val+1)*sp_param/100)  OVER  (PARTITION  BY  SP_NAME  ORDER  BY
TOTAL1) - (TOTAL2-start_val+1)*sp_param/100
       AS ACCUM
  FROM   (SELECT   r.*,   DECODE(LAG(total1)   OVER   (PARTITION   BY   SP_NAME   ORDER   BY
TOTAL1),0,1,total1) AS start_val
         FROM TEST_RULE r
       ) r;

IF_JIETI SP_NAME   PRICE_TYPE SP_PARAM  TEL_PARAM    TOTAL1     TOTAL2 START_VAL    ACCUM
-------- ---------- ---------- -------- ---------- ---------- ---------- ---------- ----------
       0 a                   1       40         60
       0 b                   2      800
       1 c                   1       30         70          0        500          0          0
       1 c                   1       40         60        501       1000          1      150.3
       1 c                   1       50         50       1001       2000       1001      550.3
       1 c                   1       60         40       2001    9999999       2001     1050.3
       2 d                   2        0                    0        500          0          0
       2 d                   1       40         60        501       1000          1          0
       2 d                   1       50         50       1001       2000       1001        400
       2 d                   1       60         40       2001    9999999       2001        900

已选择 10 行。
```

脚本 15-19　计费规则案例实现分解 3

步骤 4：

```
WITH new_rule AS
(SELECT r.*
```

```
      ,SUM((TOTAL2-start_val+1)*sp_param/100) OVER (PARTITION BY SP_NAME ORDER BY
TOTAL1) - (TOTAL2-start_val+1)*sp_param/100
       AS ACCUM
  FROM (SELECT r.*, DECODE(LAG(total1) OVER (PARTITION BY SP_NAME ORDER BY
TOTAL1),0,1,total1) AS start_val
          FROM TEST_RULE r
       ) r
)
SELECT t.*
      ,(CASE WHEN n.if_jieti =2 THEN
            (t.total_price-start_val+1)*n.sp_param/100 + n.accum
        ELSE DECODE(n.price_type,1,round(t.total_price*n.sp_param/100,2)
                   ,n.sp_param
                   )
        END) as paid
  FROM ticket t, new_rule n
WHERE n.sp_name = t.sp_name
       AND (if_jieti = 0 OR if_jieti>0 AND t.total_price BETWEEN n.TOTAL1 AND
n.TOTAL2) ;

 ID_NAME SP_NAME    TOTAL_PRICE    PAID
----------------------------------------
  121     a              100         40
  222     b             1000        800
  387     c              100         30
  645     c              600        240
  555     c             1200        600
  987     d              100          0
  333     d              600        240
  221     d             1200        500
  528     d             5000       2700
```

脚本 15-20　计费规则案例实现分解 4

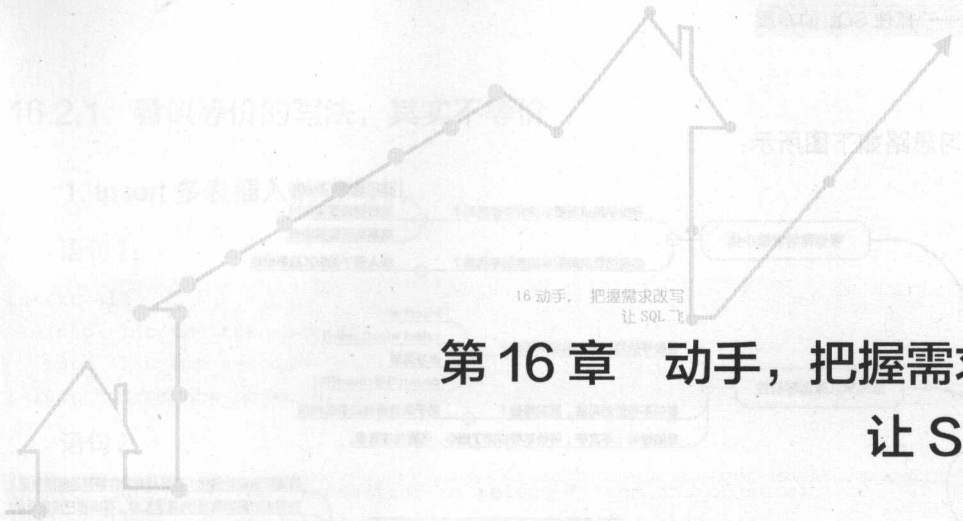
15.3　本章习题、总结与延伸

习题 1：说明分析函数的存在意义。

习题 2：分析函数有哪些应用？

习题 3：说明分析函数高效的原因。

习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：



第 16 章　动手，把握需求改写让 SQL 飞

入木三分你需要读懂本质

来，先看一段对话吧：

老师，帮我优化一下这个 SQL 吧。

好的。

小王，你是不是给错 SQL 了，这 SQL 也就 1s 出来了，逻辑读也不过 20。

老师，就这语句，这语句每小时要执行 1 千万次啊！

哦，这样，为啥要执行这么多次，你去了解一下。那我再看看，好吗？

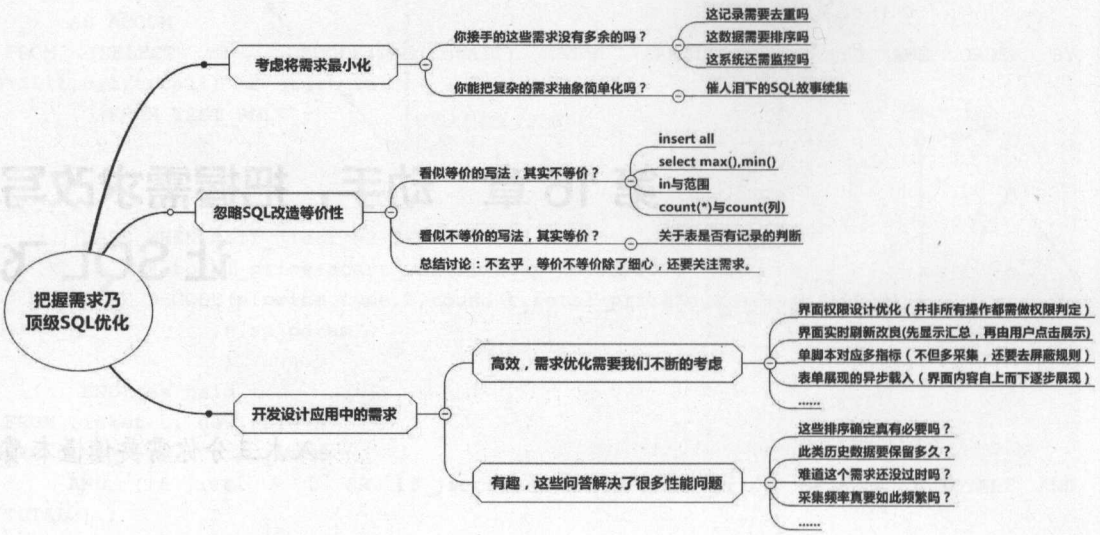
好的。

我们来看看这段短短的对话，其实内在还是有很多玄机的。首先就是老师为啥会说这个语句给错了，因为他看到语句执行得很快，逻辑读也很少，觉得 SQL 优化空间不大，用户的感知应该也不会太明显。其次，当他听说语句执行次数达到每小时上千万次时，又要再去看看，因为累积起来，对系统总体的性能影响是很大的。比如逻辑读从 20 降到 10，那 1 千万次意味着每小时系统的逻辑读减少了 1 亿个，这是多惊人的数字啊。最后老师同时还让小王去了解为什么要执行这么多次，这意味着老师想从业务层面入手，了解这 1 千万次执行的必要性，看看能否直接将执行次数降下来。

这段对话是非常经典的对话，折射出“了解需求，厘清业务关系”是非常重要的！这个语句如果能根据业务或者特定的语法进行等价改写，或许就能再提升。而如果能根据对业务的准确理解，巧妙地降低了执行的次数，这个优化工作就已经可以结束了。

总结起来就是，你要确保具备等价改造 SQL 的意识和本领，然后再大胆结合业务，把握真正的需求，完成优化改造。

总体学习思路如下图所示：



16.1 考虑需求最小化

在开发设计中，需求最小化是非常重要的，非常实用，套路如下：

当你拿到一个业务需求的时候，尽可能地把它转化成和业务无关的表达式，最重要的思路就是建一张表，字段不要太多，能满足需求就可以了，然后插入你要的数据，告诉开发人员我们需要展现什么样的结果。

笔者之前遇到一个这样的情况，一个开发人员向我咨询如何实现某个项目需求最高效，整整交流了 30 分钟我还是没明白他的需求，他向我说了很多无用的背景知识。后来我让他建表插数据直接亲手展现结果，才明白了他的意思。原来是要行列转化，然后我用一条 SQL 帮他实现了，测试结果正确。他要做的就是，把这个测试表的表名改成他的表名，列名改成他的列名，就 OK 了。

16.2 千万弄清 SQL 改造的等价性

除了需求最小化外，还有一点非常重要，那就是一定要注意代码改写的等价性。这里其实陷阱重重，很多时候，看似等价，其实不等价，看似不等价，其实根据业务来看，却是等价的。

16.2.1　看似等价的写法，其实不等价

1. Insert 多表插入的玄与机

语句 1：

```
insert all
   into  ljb_tmp_transaction
   into  ljb_tmp_session
select * from dba_objects;
```

语句 2：

```
insert  into  ljb_tmp_transaction as select * from dba_objects;
insert  into  ljb_tmp_session as select * from dba_objects;
```

脚本 16-1　Insert 多表插入的玄与机

语句 1 是不是有点眼熟？想想在前面哪儿出现过？OK，问题来了，请问，这两个语句等价吗？

很显然，语句 1 和语句 2 是不等价的。因为语句 1 是写在一起的，ljb_tmp_transaction 和 ljb_tmp_session 两表的数据一定是一样多。而语句 2 是分开写的，由于有时间差，在不同的时刻，dba_objects 的记录是变化的，所以 ljb_tmp_transaction 和 ljb_tmp_session 两表的记录不一定是相等的。

2. max 及 min 写法的分与合

构造环境：

```
drop table t purge;
create table t as select * from dba_objects;
alter table t add constraint pk1_object_id primary key (OBJECT_ID);
set autotrace on
set linesize 1000
```

语句 1：

```
select min(object_id),max(object_id) from t;
```

语句 2（分开写）：

```
select max(object_id) from t;
select min(object_id) from t;
```

脚本 16-2　max 及 min 写法的分与合

请问，语句 1 和语句 2 这两种写法等价吗？

很显然，语句 1 和语句 2 是不等价的。因为语句 1 的 max 和 min 写在一起，无论如何 max 的值都不会小于 min 的值。不过如果像语句 2 一样分开写，由于两个语句执行的时候有时间差，或许此处得到的 max 值比 min 值更小，都有可能。

3. In 和><写法之间的同与异

构造环境：

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id,object_type);
UPDATE t SET OBJECT_ID=20 WHERE ROWNUM<=26000;
UPDATE t SET OBJECT_ID=21 WHERE OBJECT_ID<>20;
COMMIT;
set linesize 266
set pagesize 1
alter session set statistics_level=all ;
```

语句 1：

```
select  /*+index(t,idx_object_id)*/ * from t  where object_TYPE='TABLE'  AND OBJECT_ID >=
20 AND OBJECT_ID<= 21;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------
|Id |Operation                    |Name          |Starts|E-Rows|A-Rows|   A-Time  |Buffers|
-----------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1|      |  2925 |00:00:00.03|   1103|
| 1 | TABLE ACCESS BY INDEX ROWID|T              |    1| 2126 |  2925 |00:00:00.03|   1103|
|*2 |  INDEX RANGE SCAN           |IDX_OBJECT_ID|    1|  320 |  2925 |00:00:00.02|    730|
-----------------------------------------------------------------------------------------
```

语句 2：

```
Select /*+index(t,idx_object_id)*/ * from t where object_TYPE='TABLE'  AND  OBJECT_ID IN
(20,21);
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------
|Id|Operation                    |Name          |Starts|E-Rows|A-Rows| A-Time   |Buffers|
-----------------------------------------------------------------------------------------
| 1|INLIST ITERATOR              |              |    1 |      | 2920|00:00:00.01|   563|
| 2| TABLE ACCESS BY INDEX ROWID|t              |    2 |   2592| 2920|00:00:00.01|   563|
|*3|  INDEX RANGE SCAN           |IDX_OBJECT_ID |    2 |     1| 2920|00:00:00.01|   214|
-----------------------------------------------------------------------------------------
```

脚本 16-3　In 和><写法之间的同与异

请问，语句 1 和语句 2 这两种写法等价吗?

4. count 列和*结论的对与错

构造环境：

```
drop table t purge;
create table t as select * from dba_objects;
create index idx_obj_id on t(object_id);
update t set object_id =null where rownum<=2;
set linesize 1000
```

```
set autotrace on
```

看看 count(*)的写法：

```
SQL> select count(*) from t;
  COUNT(*)
----------
    111113
执行计划
--------
Plan hash value: 2966233522
---------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   306   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    |   109K|   306   (1)| 00:00:04 |
-----------------------------------------------------------------------
Note
-----
   - dynamic sampling used for this statement
统计信息
--------
          0  recursive calls
          0  db block gets
       1723  consistent gets
          0  physical reads
          0  redo size
        421  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

再看看 count(列)的写法：

```
SQL> select count(object_id) from t;
COUNT(OBJECT_ID)
----------------
          111111
执行计划
--------
Plan hash value: 1232025837
---------------------------
| Id  | Operation             | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |            |     1 |    13 |    48   (3)| 00:00:01 |
|   1 |  SORT AGGREGATE       |            |     1 |    13 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX_OBJ_ID |   109K|  1390K|    48   (3)| 00:00:01 |
-------------------------------------------------------------------------------------
```

```
Note
----
   - dynamic sampling used for this statement
统计信息
--------
          0  recursive calls
          0  db block gets
        254  consistent gets
          0  physical reads
          0  redo size
        429  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

脚本 16-4　count 列和*结论的对与错

请问你关注到什么了？是不是说，哎呀，count(列)好快啊！嗯，具体快慢的原因我们不想去了解，我只想问你，你的眼中只有快吗？请看看两个语句查询的结果分别是什么。

16.2.2　看似不等价的写法，其实等价

关于表是否有记录的判断：

```
begin
select count(*) into v_cnt from t1 ;
if v_cnt>0
then  …A 逻辑…
else
then  …B 逻辑…
End;
```

我来翻译一下这段需求：获取 t1 表的记录数，判断是否大于 0，如果大于 0 走 A 逻辑，否则就走 B 逻辑。因此代码就如上所示来实现了。真正的需求是这样吗？其实应该是这样的：只要 t1 表有记录就走 A 逻辑，否则走 B 逻辑。两者有区别吗？其实区别还是很大的，前者可是强调获取记录数，我们是不是一定要遍历整个表得出一个记录数才知道是否大于 0？

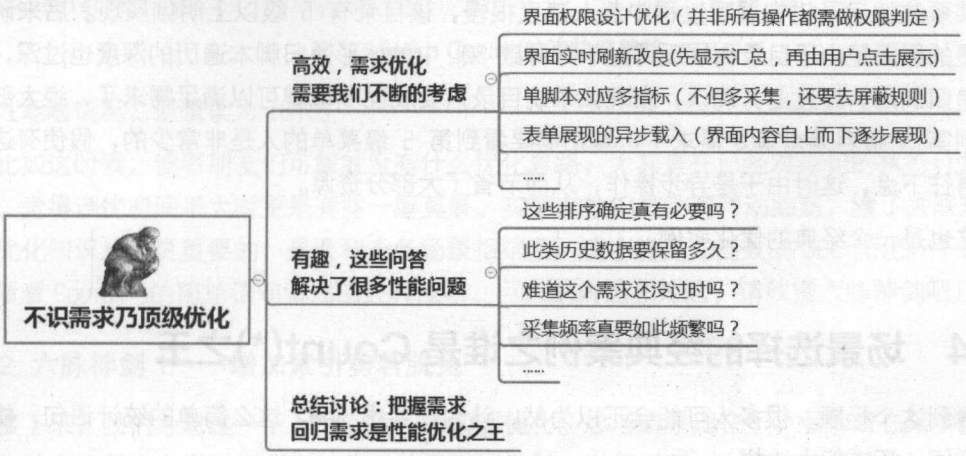
真正理解了需求，我们会这样实现，只要从 t1 表中成功获取到第一条记录，就可以停止检索了，表示该表有记录了，难道事实不是这样吗？

因此原先的 SQL1 从 Select count(*) from t1; 被改造为：

```
Select count(*) from t1 where rownum=1;
begin
select count(*) into v_cnt from t1 where rownum=1;
if v_cnt=1
then  …A 逻辑…
else
```

```
then  …B 逻辑…
End;
```

16.3　开发设计应用中的需求



16.3.1　界面权限设计优化

优化前，用户登录进系统，要对他所有菜单的权限进行鉴权；优化后，用户只有访问到具体某个菜单时，才对他是否有这个菜单的权限进行鉴权。这样优化后让系统少跑了很多 SQL，因为很多用户只是登录到首页看看公告，根本就没有再访问某些具体菜单。

16.3.2　界面汇总与展现

比如某账户登录某页面，会提示有 XX 条消息、XX 个未处理工单等汇总信息，这时候我们如果点击进去，就可以看到具体的明细。实际上很多人可能并不点击进去看明细，看到这类汇总信息就够了。

然而，现实中有很多系统，在用户未登录进去看明细的时候，已经完成了明细的载入，这必然导致页面加载速度变慢。如果异步操作，由于会有大量的人只看汇总不看明细，这无形之中可使系统少执行大量的 SQL，并且也让页面初始载入的速度大幅提升。

16.3.3　界面实时刷新改良

某界面有 3 部分数据来源。比如第一部分是公司岗位、工作年限等，它们来源于公司系统；第二部分是身份证信息、有无犯罪记录、户口所在地等，来源于公安系统；第 3 部分是学历、教育履历等，来源于教育系统。

优化前，3 部分数据全部载入后才展现给用户看；优化后，第 1 部分先展现，等用户眼睛

看完第 1 部分看中间的第 2 部分信息时，第 2 部分已经展现了，等用户看完第 2 部分的信息转到尾部的第 3 部分信息时，第 3 部分也展现完毕了。

16.3.4 目录树菜单的优化

某系统的目录树在展现过程中载入速度很慢，该目录有 5 级以上明细展现。后来研究发现，慢的原因是由于目录树展现得过于深导致 SQL 中的树形递归脚本遍历的深度也过深，产生大量递归调用。经过业务确认，发现该系统目录只要展现 3 级就可以满足需求了，绝大部分人了解到第 3 级目录就满足需求了，真正需要看到第 5 级菜单的人是非常少的，假使有这个需求，再往下点，这时由于是异步操作，从而节省了大部分资源。

这也是一个经典的优化案例。

16.4 场景选择的经典案例之谁是 Count(*)之王

看到这个标题，很多人可能会不以为然，认为作者很 low，这么简单的统计语句，能有啥优化空间，还性能大比拼……

哦，其实简单的背后不简单，来，一起看看如何“不择手段”，让 count(*) 飞起来。不过这里关键是让读者去思考，为什么能飞。

16.4.1 优化过程

1. 普通思路

首先我们来看一种最简单的方式，就是什么手段都没用的情况下，SQL 的性能如何，具体如下：

```
drop table t purge;
create table t as  select * from dba_objects;
alter table T modify OBJECT_NAME not null;
select count(*) from t;
set autotrace traceonly
set linesize 1000
set timing on
select COUNT(*) FROM T;
执行计划
--------
Plan hash value: 2966233522
---------------------------
| Id | Operation           | Name | Rows    | Cost (%CPU) | Time      |
-----------------------------------------------------------------------
|  0 | SELECT STATEMENT    |      |       1 |    292   (1)|  00:00:04 |
|  1 |  SORT AGGREGATE     |      |       1 |             |           |
```

```
|  2 |   TABLE ACCESS FULL | T    |  92256 |    292   (1)|  00:00:04 |
---------------------------------------------------------------------------
统计信息
--------
 0     recursive calls
0     db block gets
1048  consistent gets
```

脚本 16-5　Count(*)的普通思路

性能啥情况，逻辑读为 1048。

比如这时候，读者朋友们可能就没有什么优化思路，尤其是在已经对语句的执行计划做过分析，觉得调优空间不大时更是有些一筹莫展。实际上他们是没有开动脑筋，除了去联想 SQL 相关优化知识外，更重要的，是要和业务场景相结合，这才能完成高效的 SQL 优化。下面我们一起看看 Count(*)的简单语句背后的经典故事、巨大的内涵和外延，请欣赏六脉神剑吧！

2. 六脉神剑 1——增加索引势若脱兔

接下来，我们考虑建一个 btree 索引来实现优化。思考的原理在于，假如表在某字段上建索引，并且确保这个索引列非空，此时的索引就类似一张仅拥有一个字段的“瘦表”，所以肯定可以得到统计条数。又因为这“瘦表”体积小，所以性能肯定要高得多，具体试验如下：

```
drop table t purge;
create table t as  select * from dba_objects;
alter table T modify OBJECT_NAME not null;
create  index idx_object_name on t(object_name);
set autotrace traceonly
set timing on
select count(*) from t;
执行计划
--------
Plan hash value: 1178070731
-------------------------------------------------------------------------------
| Id | Operation             | Name            | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|  0 | SELECT STATEMENT      |                 |     1 |   105   (1)| 00:00:02 |
|  1 |  SORT AGGREGATE       |                 |     1 |            |          |
|  2 |   INDEX FAST FULL SCAN| IDX_OBJECT_NAME | 92256 |   105   (1)| 00:00:02 |
-------------------------------------------------------------------------------
统计信息
---------
0  recursive calls
0  db block gets
372  consistent gets
```

脚本 16-6　Count(*)用到普通索引

性能啥情况，逻辑读从 1048 减少到 372。

3. 六脉神剑 2——位图索引风驰电掣

性能提升明显，似乎一下子受到鼓舞。在特定的情况下，我们还有更厉害的招式，那就是使用位图索引来提升性能。原理在于，位图索引是根据位图索引列的取值来分类判断，比如性别字段，不是男就是女。索引块中的男块中存的值不是 0 就是 1 的比特值，占空间极小。个小也不影响从中得到统计条数，所以性能同样能大幅提升，试验如下：

```
drop table t purge;
create table t as  select * from dba_objects;
Update t  Set object_name='abc';
Update t Set object_name='evf' Where rownum<=20000;
create bitmap index idx_object_name on t(object_name);
set autotrace traceonly
set timing on
select count(*) from t;
执行计划
--------
Plan hash value: 1696023018
---------------------------
|Id|Operation                    |Name             |Rows   |Cost (%CPU) |Time      |
-------------------------------------------------------------------------------------
| 0|SELECT STATEMENT             |                 |     1|     5   (0)| 00:00:01 |
| 1| SORT AGGREGATE              |                 |     1|            |          |
| 2|  BITMAP CONVERSION COUNT    |                 | 92256|     5   (0)| 00:00:01 |
| 3|   BITMAP INDEX FAST FULL SCAN | IDX_OBJECT_NAME |      |            |          |
-------------------------------------------------------------------------------------
统计信息
--------
0  recursive calls
0  db block gets
6  consistent gets
```

脚本 16-7　Count(*)用到位图索引

性能啥情况，逻辑读从 372 瞬间缩减为 6。

4. 六脉神剑 3——物化视图蹑影追风

到了这个层面，还能再优化吗？当然可以，给大家介绍一个秘密武器：物化视图！这是什么东西，其实说白了就是一种空间换时间的原理。我们预先把一些统计结果都计算好，在计算的时候直接就把结果拿来用，不过如果基表数据变化了，预先计算好的统计结果也要跟着设法变化，否则就会出现不一致，这点是需要注意的。

请看试验，其中 MV_COUNT_T 就是物化视图，大家注意看，执行的语句明明是 select

count(*) from t，在执行计划中却看不到访问 T 表，而是访问 mv_count_t 视图。具体如下：

```
DROP materialized view MV_COUNT_T;
drop table t purge;
create table t as  select * from dba_objects;
Update t  Set object_name='abc';
Update t Set object_name='evf' Where rownum<=20000;
create  materialized view  mv_count_t
          build immediate
          refresh on commit
          enable query rewrite
            as
          select count(*) FROM T;
set autotrace traceonly
set linesize 1000
select COUNT(*) FROM T;
/
执行计划
--------
Plan hash value: 3655891017
---------------------------
|Id| Operation                    |Name       | Rows  | Bytes | Cost (%CPU)|Time     |
-------------------------------------------------------------------------------------
| 0| SELECT STATEMENT             |           |     1 |    13 |     3   (0)|00:00:01 |
| 1|  MAT_VIEW REWRITE ACCESS FULL|MV_COUNT_T|     1 |    13 |     3   (0)|00:00:01 |
-------------------------------------------------------------------------------------
统计信息
--------
     0  recursive calls
     0  db block gets
     3  consistent gets
```

脚本 16-8　Count(*)用到物化视图

从结果来看，惊喜更大，性能啥情况，逻辑读从 6 缩减为 3。

5. 六脉神剑 4——缓存结果奔逸绝尘

这下不能再提升了吧。其实如果用心，还可以再提升，是不是有些不可思议？这里再引入一个技术手段：缓存结果集。这也是类似空间换时间，SQL 在设定了 result_cache 的参数后，首次执行会略慢一些，因为会将这个 SQL 的执行结果放在共享内存里，然后下次执行同样的命令时，直接从共享内存中获取结果。如果基表数据有变化，共享内存中的结果就会失效。

来，我们试验一下这个写法对性能有啥影响(记住，以下是执行第 2 次后的结果)，请看试验如下：

```
drop table t purge;
create table t as  select * from dba_objects;
select count(*) from t;
```

```
set linesize 1000
set autotrace traceonly
select /*+ result_cache */ count(*) from t;
执行计划
----------------------------
Plan hash value: 2966233522
-------------------------------------------------------------------------------------------
|Id  | Operation           | Name                        | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT    |                             |     1 |    23   (0)| 00:00:01 |
|  1 |  RESULT CACHE       | 034w33ddtzk0mc7kvwt84zayrc|       |            |          |
|  2 |   SORT AGGREGATE    |                             |     1 |            |          |
|  3 |    TABLE ACCESS FULL| T                           | 10000 |    23   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
统计信息
----------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
```

脚本 16-9　Count(*)用到缓存结果集

这下简直是用震惊来形容了。执行第 2 遍时，缓存结果集让逻辑读从 3 缩减到 0！没错，你没看错，真的是 0。其实这也好理解，因为直接拿结果来用，所以为 0 也可以理解。物化视图好歹需要访问这个视图对象，所以没有 0 也是可以理解的。真的好神奇的性能提升！

6. 六脉神剑 5——业务理解电光火石

这下应该不能再优化了吧？

不对，还可以继续！我们如果站在业务的角度去理解这个 SQL，可能会看到一件很奇怪的事情，那就是可能大家会看到如下无法理解的 SQL：

```
select count(*) from t where rownum=1;
```

这有些莫名其妙，这是什么意思？嗯，我来告诉你吧，假如这个 count(*)只是为了判断该表是否有记录，然后根据大于 0 来走某逻辑，等于 0 来走另外一个逻辑，你该如何做？

想明白没？需求如果真是如此，判断 rownum=1 是否有 1 条不是一样的道理吗？问题是，这样的改写优化导致表不管多大，永远只访问第 1 条，速度问题还需要纠结吗？

7. 六脉神剑 6——分析需求终极之剑

还能优化吗？估计读者已经不敢再往下想了，我的答案是：当然，还能优化！

比如我们考虑这么一种可能性，这个 Count(*)统计条数语句，是多余的！这并不是异想开天，比如我们统计的结果在整个业务逻辑中并没有用，或者说它的作用被别的条件重叠替代。然后我们怎么办？

直接砍了这条语句，这里没有 SQL，还有谁能超越？

这真的是：绝招！

16.4.2　优化总结

下表是优化的各个步骤和性能提升的效果展示，从这里可以看出，其实优化是一种非常灵活的技术手段，要灵活结合各种场景进行优化。建普通索引的手段看起来还算普通，但是接下来建位图索引就要注意了，这是在特定场合下使用的手段，如果这列的重复度很低就不适合建位图索引；而物化视图则是应用在数据要求不怎么及时的场景下；接下来是缓存结果集的手段，如果表频繁更新，也是不适合的。

不过，关键是，在特定的场合下，这些手段有可能被允许使用，如果你的脑子里没有这些念头，你是永远不会在合适的场景选择合适的技术的。

六脉神剑 5 将 SQL 改写得面目全非，是的，你不认识这个写法，但是在某种业务场景下，这个写法认识你。

不过六脉神剑 6 才是真正震撼读者的地方，现实中确实是有非常多的 SQL，是多余的！

count(*)性能大比拼				
手　段	**命　令**	**主要原理**	**详细说明**	**性能情况**
啥没做		全表扫描	OLTP 中，通常是最慢的方式	逻辑读为 1048
六脉神剑 1：增加索引	create index idx_object_name on t(object_name);	从全表扫描转成全索引扫描	因为索引一般比表小得多，所以全表扫描转成全索引扫描，性能能大幅度提升	逻辑读为 372
六脉神剑 2：位图索引	create bitmap index idx_object_name on t(object_name);	从 BTREE 索引扫描转成位图索引扫描	位图索引的大小比 BTREE 索引要小得多，所以位图索引扫描快	逻辑读为 6
六脉神剑 3：物化视图	create materialized view mv_count_t build immediate refresh on commit enable query rewrite as select count(*) FROM T;	空间换时间	要注意，如果数据要求比较实时，就不适用	逻辑读为 3
六脉神剑 4：缓存结果	select /*+ result_cache */ count(*) from t;	直接把查询结果拿来用	要注意，如果数据频繁更新，就不适用	逻辑读为 0

续表

count(*)性能大比拼				
手　段	命　令	主要原理	详细说明	性能情况
六脉神剑 5：业务理解	select count(*) from t where rownum=1;	如果 count(*)只是为了判断条数，就加上 rownum=1 来判断是否为 1	业务需求转换，获取条数有的时候只是为了看看表是否为空，这时候是否是 1 条和是否大于 0 其实是一样的	不言而喻
六脉神剑 6：分析需求	据说，这个 Count(*)统计条数语句，是多余的！直接砍了这条语句，这里没有 SQL			无敌！

16.5　本章习题、总结与延伸

习题 1：请说说如下两个语句是否等价，如果等价，这个改写有意义吗？

```
select min(object_id),max(object_id) from t;
select max, min  from
(select max(object_id) max from t ) a,
(select min(object_id) min from t  )  b;
```

习题 2：说说下面两个语句有可能等价吗？意义何在？

```
select * from t where object_type='VIEW' and OWNER='LJB';
select /*+INDEX(T,idx_union)*/ * from t T where object_type='VIEW' and OBJECT_ID IN
(20,21,22) AND OWNER='LJB';
```

习题及疑问的邮件发送地址与本章总结及解题二维码如下图所示：

第17章 总结与延伸：从勿信讹传到洞若观火

辨别识真伪其实很简单

整本书都写完了，从优化方法论到具体的优化各种知识。不过，还有一些其他的认识需要在全书的最后做一个补充，那就是：如何让自己有一个清醒冷静的头脑。这是非常有必要的，因为在现在的互联网时代，到处都充斥着虚假错误的信息，不明真相的人很容易瞎转或者盲从，不判断真伪，到最后甚至给自己造成了很大的损失。SQL 优化也不例外，网络上很多所谓的优化宝典等描述的建议都是错误的或者是过时的，你信吗？ 如果不信，请看随后的关于 SQL 优化的各种谣传。都是有脚本有真相哦。

想弄清楚这所有的疑问，一起跟我往下看，先从 SQL 优化的讹传开始。

17.1 SQL 优化的各个误区

17.1.1 COUNT(*)与 COUNT(列)的传言

真伪?

讲台上，某知名讲师正唾沫横飞地讲述 SQL 优化的宝典……

关于 SQL 优化是有很多技巧的，来，我们来说优化宝典第 8 式，获得表记录数的语句优化。这里记住，如果写成了 COUNT(*)，性能就会比较低，千万不能用！那我们要写成什么呢？我们要写成 COUNT（列），然后这列要有索引，这样速度就可以快很多，大家记住没有?

记住了!

等等，老师，我觉得有点奇怪，从得到条数来看，COUNT(*)更好记啊。

所以，这是老师教你们的优化宝典啊，记住老师说的，就能让 SQL 跑得更快。

学生们纷纷点头。

老师，如果这一列删除了，表还在，我们写 select(列)岂不是表记录还出不来呢？

这……，这怎么会随便删除列呢？你想多了！

学生们继续纷纷点头。

事实到底是怎样的呢？我们看看具体的试验。

验证！

难道 count(列)真比 count(*)快？我们分别做两个试验，具体见脚本 17-1、脚本 17-2，如下：

```
drop table t purge;
create table t as select * from dba_objects;
alter table T modify object_id  null;
update t set object_id =rownum ;
set timing on
set linesize 1000
set autotrace on
SELECT count(*) from t;
  COUNT(*)
----------
   73218
执行计划
--------
Plan hash value: 2966233522
-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 81431 |   292   (1)| 00:00:04 |
-------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1048  consistent gets
```

脚本 17-1　表无索引时，count(*)的性能

```
select count(object_id) from t;
COUNT(OBJECT_ID)
----------------
     73218
执行计划
```

```
--------
Plan hash value: 2966233522
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    13 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |    13 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 81431 |  1033K|   292   (1)| 00:00:04 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1048  consistent gets
```

脚本 17-2　表无索引时，count(列)的性能

看来 count(列)比 count(*) 更快是谣传，明明是一样快嘛，真相是这样吗？

NO!NO!NO!请继续往下看。来来，建个索引看看，请看脚本 17-3 和脚本 17-4，如下：

```
create index idx_object_id on t(object_id);
select count(*) from t;
  COUNT(*)
----------
   73218
执行计划
--------
Plan hash value: 2966233522
-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   292   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T    | 81431 |   292   (1)| 00:00:04 |
-------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
       1048  consistent gets
```

脚本 17-3　表有索引时，count(*)的性能

```
select count(object_id) from t;
COUNT(OBJECT_ID)
----------------
      73218
执行计划
--------
Plan hash value: 1131838604
```

```
---------------------------
| Id  | Operation             | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               |     1 |    13 |    49   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |               |     1 |    13 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX_OBJECT_ID | 81431 |  1033K|    49   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        170  consistent gets
```

脚本 17-4　表有索引时，count(列)的性能

哇，原来真的是用 COUNT(列) 比 COUNT(*)要快啊，因为 COUNT(*)不能用到索引，而 COUNT(列)可以，看来宝典 108 式是真理啊！

真相是如此吗？NONONO!还请看官继续往下看，请看脚本 17-5 和脚本 17-6，具体如下：

```
alter table T modify object_id  not  null;
select count(*) from t;
  COUNT(*)
----------
     73218
执行计划
--------
Plan hash value: 1131838604
---------------------------
| Id  | Operation             | Name          | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               |     1 |    49   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |               |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| IDX_OBJECT_ID | 81431 |    49   (0)| 00:00:01 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        170  consistent gets
```

脚本 17-5　表有索引且索引列非空时，count(*)的性能

```
select count(object_id) from t;
COUNT(OBJECT_ID)
----------------
     73218
执行计划
--------
Plan hash value: 1131838604
---------------------------
```

```
| Id  | Operation             | Name          | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               |     1 |    49   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |               |     1 |             |          |
|   2 |   INDEX FAST FULL SCAN| IDX_OBJECT_ID | 81431 |    49   (0)| 00:00:01 |
-------------------------------------------------------------------------------
统计信息
--------
          0  recursive calls
          0  db block gets
        170  consistent gets
```

脚本 17-6　表有索引且索引列非空时，count(列)的性能

很显然，速度是一样快！一点都不奇怪，我们在前面索引章节中学过索引三大特性后，这个应该很容易理解的。

结论！

结论：

count(*)与count(列)的试验结果			
试验条件	count(*) 开销	count(列)开销	性能比较
无索引	cost是292，逻辑读是1048	cost是292，逻辑读是1048	一样快
有索引，且索引列的属性允许为空	cost是292，逻辑读是1048	cost是49，逻辑读是170	count(列)快
有索引，且索引列的属性不允许为空	cost是49，逻辑读是170	cost是49，逻辑读是170	一样快

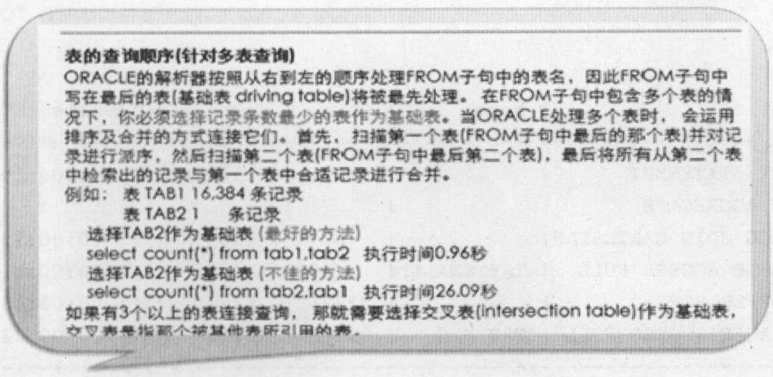
看来在没有索引的情况下，count(列)和 count(*)其实一样快。如果有索引，索引列是非空的，count(*)可用到索引，此时一样快！

终于得出结论了，从这个结论来看，知名讲师的 SQL 优化宝典是错的！

17.1.2　谈 SQL 编写顺序之流言蜚语

真伪？

讲台上，唾沫继续横飞……



同学们，我们继续 SQL 优化宝典第 18 式。你们在写 SQL 时，一定要注意 SQL 书写的顺序，这对性能会有巨大的影响，比如 tab1 记录数有 16384 条，tab2 只有 1 条。那你就要把小表 tab2 写在最后的位置，因为写在最后的表会被先驱动，比如 select * from tab2, tab1 where……这样就出问题了！应该怎么写？

老师，是不是写成 select * from tab1, tab2 where ...

很棒，大家都记住了吗？

记住了！

等等，老师，我觉得有点奇怪，我们每次都要记住这些表的大小，不是很累吗？

所以，这是老师教你们的优化宝典啊，记住老师说的，就能让 SQL 跑得更快。

学生们纷纷点头。

老师，如果有一天小表变成大表，大表变成小表，我们是不是要改代码呢？

这……，这怎么会随便小表变大表，大表变小表呢？你想多了！

学生们继续纷纷点头。

验证！

事实到底是怎样的呢？我们看看具体的试验。

具体见脚本 17-7，我们发现无论表的顺序如何更换，性能是一样的：

```
drop table tab_big;
drop table tab_small;
create table tab_big  as select * from dba_objects where rownum<=30000;
create table tab_small  as select * from dba_objects where rownum<=10;
set autotrace traceonly
set linesize 1000
set timing on
select count(*) from tab_big,tab_small   ;
select count(*) from tab_small,tab_big   ;

--两个的执行计划都一样!

执行计划
--------
| Id  | Operation             | Name      | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |     1 |  1184   (1)| 00:00:15 |
|   1 |  SORT AGGREGATE       |           |     1 |            |          |
|   2 |   MERGE JOIN CARTESIAN|           |   337K|  1184   (1)| 00:00:15 |
|   3 |    TABLE ACCESS FULL  | TAB_SMALL |    10 |     3   (0)| 00:00:01 |
|   4 |    BUFFER SORT        |           | 33776 |  1181   (1)| 00:00:15 |
|   5 |     TABLE ACCESS FULL | TAB_BIG   | 33776 |   118   (0)| 00:00:02 |
-------------------------------------------------------------------------
```

```
Note
-----
   - dynamic sampling used for this statement (level=2)
统计信息
--------
          7  recursive calls
          0  db block gets
        503  consistent gets
```

脚本 17-7　访问两表，顺序无关性能

咋回事呢？老师不是都做过试验来证明自己了吗？其实这里的真相是：在基于规则的时代，Oracle 由于没有出现先进的基于代价的优化方法，优化器只能进行这么落后的优化。我们继续看看脚本 17-8，如下：

```
select /*+rule*/ count(*) from tab_big,tab_small ;

执行计划
--------
| Id  | Operation            | Name       |
----------------------------------------
|   0 | SELECT STATEMENT     |            |
|   1 |  SORT AGGREGATE      |            |
|   2 |   NESTED LOOPS       |            |
|   3 |    TABLE ACCESS FULL| TAB_SMALL  |
|   4 |    TABLE ACCESS FULL| TAB_BIG    |
----------------------------------------
统计信息
--------
          1  recursive calls
          0  db block gets
       4223  consistent gets

select /*+rule*/ count(*) from tab_small,tab_big ;
执行计划
--------
| Id  | Operation            | Name       |
----------------------------------------
|   0 | SELECT STATEMENT     |            |
|   1 |  SORT AGGREGATE      |            |
|   2 |   NESTED LOOPS       |            |
|   3 |    TABLE ACCESS FULL| TAB_BIG    |
|   4 |    TABLE ACCESS FULL| TAB_SMALL  |
----------------------------------------
Note
-----
   - rule based optimizer used (consider using cbo)
统计信息
--------
          1  recursive calls
```

```
         0  db block gets
     90422  consistent gets
```

脚本 17-8　基于规则，访问两表的顺序会影响性能

看明白了，显然上一条性能好于下一条，在脚本 17-8 中，我们特意将语句改成基于规则的写法，性能终于出现巨大差异，看来知名讲师做试验这事，是真的。

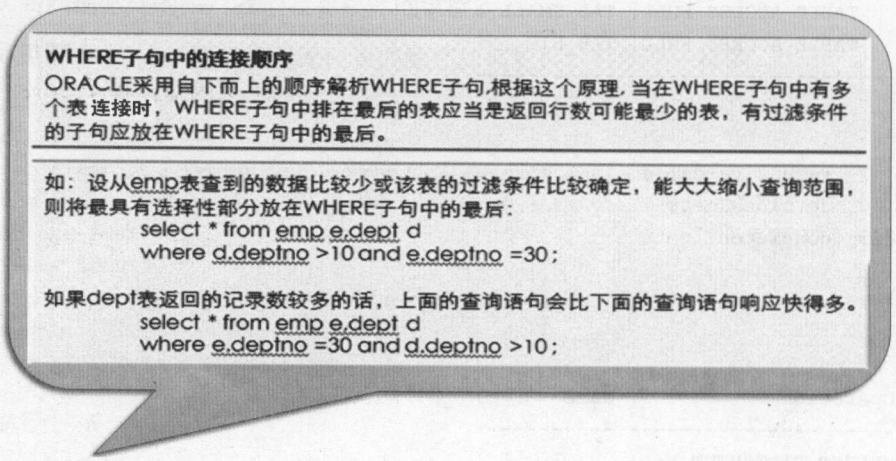
结论！

结论：

原来表连接顺序的说法早就过时了，那是基于规则的时代，现在我们是基于代价的。

讲台上，激情继续燃烧……

同学们，我们继续 SQL 优化宝典第 19 式。你们在写 SQL 时，一定要注意 SQL 书写中 where 的条件顺序，切记要将过滤条件中返回记录少的条件写在 where 的最后。这也会对性能有巨大的影响，比如 deptno=30 返回的记录数比 deptno>10 返回的记录数少，我们就要这么写代码：where deptno>10 and deptno=30。请大家要牢记。



学生们纷纷点头。

等等，老师，我觉得有点奇怪。如果有一天 deptno=30 返回的记录数比 deptno>10 返回的记录数多，我们是不是要改代码呢？

这……

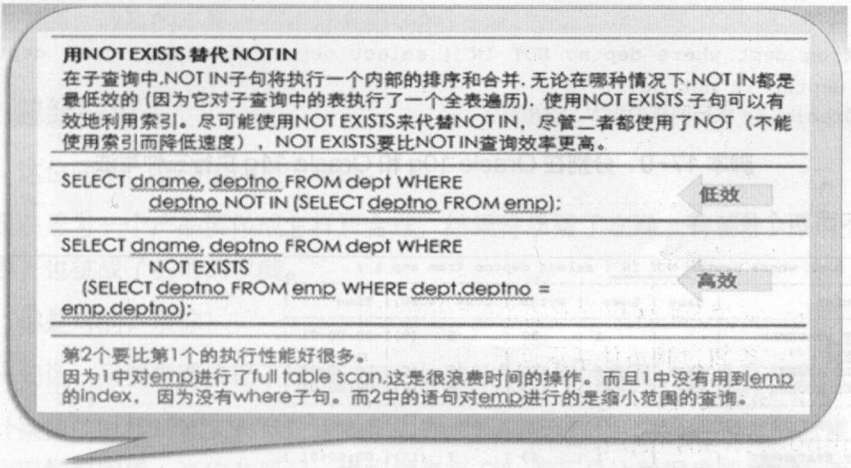
亲爱的读者，这个试验我们不用再去做了，你们应该知道结论。

17.1.3　IN 与 EXISTS 之争

真伪?

讲台上，宝典神功还在继续.....

同学们，我们继续 SQL 优化宝典第 108 式。你们在写 SQL 时，千万要注意，NOT IN 的写法效率比较低，我们要尽量考虑用 NOT EXISTS 来代替。请大家要牢记!



同学们纷纷点头。

等等，老师，我觉得有点奇怪，既然 NOT EXISTS 比 NOT IN 更高效，那 Oracle 为何要推出 NOT IN 呢?

这.....

我觉得我们不需要记这个啊，如果 NOT IN 真的很低效，那 Oracle 可以悄悄地把 NOT IN 的写法改成 NOT EXISTS，不就好了?

这.....

我忍你很久了，你是对面培训机构派来砸场子的吧，来人，把他给我拖出去!

验证!

事实到底是怎样的呢? 我们来看看具体的试验。

具体见脚本 17-9，我们分别在 Oracle 10g 和 Oracle 11g 里做试验，发现两种截然不同的情况:

```
---首先在 Oracle10g 环境试验
select * from v$version;
drop table emp purge;
drop table dept purge;
create table emp as select * from scott.emp;
```

```
create table dept as select * from scott.dept;
set timing on
set linesize 1000
set autotrace traceonly
--写法 1
select * from dept where deptno NOT IN ( select deptno from emp ) ;
--写法 2
select * from dept where not exists ( select deptno from emp where
emp.deptno=dept.deptno) ;
--写法 3
select * from dept where deptno NOT IN ( select deptno from emp where deptno is not
null) and deptno is not null;
--接下来在 Oracle11g 执行同样的脚本，这里略去
```

脚本 17-9　分别在 Oracle 10g 和 Oracle 11g 执行三种写法

请看下面两个图：

```
select * from dept where deptno NOT IN ( select deptno from emp ) ;
-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    30 |     5   (0)| 00:00:01 |
|*  1 |  FILTER            |      |       |       |            |          |
|   2 |   TABLE ACCESS FULL| DEPT |     4 |   120 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMP  |    13 |   169 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------
select * from dept where not exists ( select deptno from emp where emp.deptno=dept.deptno) ;
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    43 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN ANTI    |      |     1 |    43 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPT |     4 |   120 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
select * from dept where deptno NOT IN ( select deptno from emp where deptno is not null) and deptno is not null;
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    43 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN ANTI    |      |     1 |    43 |     7  (15)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| DEPT |     4 |   120 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------


select * from dept where deptno NOT IN ( select deptno from emp ) ;
-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     4 |   172 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN ANTI NA |      |     4 |   172 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPT |     4 |   120 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
select * from dept where not exists ( select deptno from emp where emp.deptno=dept.deptno) ;
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     4 |   172 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN ANTI    |      |     4 |   172 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPT |     4 |   120 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
SQL> select * from dept where deptno NOT IN ( select deptno from emp where deptno is not null) and deptno is not null;
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     4 |   172 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN ANTI    |      |     4 |   172 |     7  (15)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| DEPT |     4 |   120 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
```

结论！

结论：

一般来说，anti 的反连接算法比 filter 更高效，但是在 10g 时，Oracle 的这个算法不完善，必须要指定非空，才可以让 not in 用 anti 算法。**在 11g 的时候，这个情况已经改变了，无论 not in 还是 not exists，无论是否列为空，都可以走到 Oracle 比较先进高效的 anti 反连接算法。**

救命，老师，我真的不是对面派过来砸场的！

17.1.4　总结探讨

是的，这位同学，我相信你是无辜的。

你只是在思考一个产品要做成怎样更合理，结果你质疑了在网上有着悠久历史的 SQL 优化宝典的内容，也挑战了名师的权威。

但是，你是对的，顶你！

记住一句话：只要觉得流程不顺畅，用户体验效果不好的东西，都是有问题的。

你统计条数的时候对还要去记一些无关的列而感到奇怪，当你编写 SQL 的时候对还要去关注表的大小而觉得困惑，当你非要记一堆规则来写 SQL 而不是让数据库智能地去适应使你觉得不解。这些其实就是流程不顺畅，用户剔牙效果不好！

Don't cry any more!能质疑，我挺你！你也看完我们的系列试验和结论了，愿意随我来看更多的试验吗？想脑洞大开，Come on with me 。

17.2　误区背后的话题扩展

谢谢老师，你要带我看什么？

我们的系列试验和结论，对你有帮助吗？

有啊，收获很大。质疑、试验、确认，然后真相浮出水面，这种感觉真好！

很好，我检验你一下，COUNT(*)和 COUNT(列)谁快谁慢？

在表的某非空列有索引的情况下……

停，你脑洞未开，在做性能比较时，你想过这两个语句等价吗？

啥，等价？

17.2.1　话题扩展之等价与否优先

这位同学，如果表的某列，比如列 1 允许为空，空记录有 3 条，此时如果表的记录有 100

条，那就意味着 COUNT(列 1)返回 97 条，COUNT(*)返回 100 条，请问，两个表的记录都不等，又如何比较性能的优劣呢？

这……

17.2.2 话题扩展之颠覆误区观点

这位同学，如果在不考虑执行计划走索引的情况下，我说 count(*)其实比 count(列)快，你相信吗？

啥？这和之前 SQL 优化宝典第 8 式的结论完全相反！虽然宝典是错的，但是我们的试验没有得出这个相反的结论啊。

嗯，想知道真相，就动手试验证明吧。

如何动手？如何做试验？

这样，老师构造出一张表，比如构造 25 个列，让 SQL 从第一列一直 COUNT 到最后一个列，然后再执行 COUNT(*)，我们来比较一下这些语句执行的时长。

这个如何实现？还有，COUNT 的时间不是都很短吗，会容易辨别出来吗？

嗯，当然可以，来，看看老师的试验，请看脚本 17-10，如下：

```
验证脚本 1 （先构造出表和数据）
SET SERVEROUTPUT ON
SET ECHO ON
---构造出有 25 个字段的表 t
DROP TABLE t;
DECLARE
  l_sql VARCHAR2(32767);
BEGIN
  l_sql := 'CREATE TABLE t (';
  FOR i IN 1..25
  LOOP
    l_sql := l_sql || 'n' || i || ' NUMBER,';
  END LOOP;
  l_sql := l_sql || 'pad VARCHAR2(1000)) PCTFREE 10';
  EXECUTE IMMEDIATE l_sql;
END;
/
----将记录往这个表 T 中填充
DECLARE
  l_sql VARCHAR2(32767);
BEGIN
  l_sql := 'INSERT INTO t SELECT ';
  FOR i IN 1..25
  LOOP
    l_sql := l_sql || '0,';
  END LOOP;
```

```
  l_sql := l_sql || 'NULL FROM dual CONNECT BY level <= 10000';
  EXECUTE IMMEDIATE l_sql;
  COMMIT;
END;
/
--验证脚本 2（依次访问该表各字段验证）
execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t')
SELECT num_rows, blocks FROM user_tables WHERE table_name = 'T';
--以下动作观察执行速度，比较发现 COUNT(*)最快，COUNT(最大列）最慢
DECLARE
  l_dummy PLS_INTEGER;
  l_start PLS_INTEGER;
  l_stop PLS_INTEGER;
  l_sql VARCHAR2(100);
BEGIN
  l_start := dbms_utility.get_time;
  FOR j IN 1..1000
  LOOP
    EXECUTE IMMEDIATE 'SELECT count(*) FROM t' INTO l_dummy;
  END LOOP;
  l_stop := dbms_utility.get_time;
  dbms_output.put_line((l_stop-l_start)/100);

  FOR i IN 1..25
  LOOP
    l_sql := 'SELECT count(n' || i || ') FROM t';
    l_start := dbms_utility.get_time;
    FOR j IN 1..1000
    LOOP
      EXECUTE IMMEDIATE l_sql INTO l_dummy;
    END LOOP;
    l_stop := dbms_utility.get_time;
    dbms_output.put_line((l_stop-l_start)/100);
  END LOOP;
END;
/
```

脚本 17-10　访问不同列的性能测试的环境构造

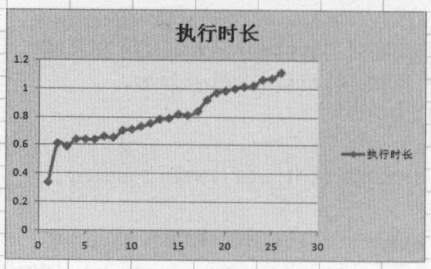
以下是输出结果：

```
.34
.63
.59
.62
.63
.62
.64
.66
.65
```

```
.69
.7
.73
.78
.75
.8
.82
.85
.85
.92
.91
.94
.96
.99
1.06
1.06
1.07
1.11
PL/SQL 过程已成功完成。
```

通过试验我们发现，COUNT(*)速度最快，其次是 COUNT 列 1，最慢的是 COUNT(列 20)。在 EXCEL 中进行统计，会形成一个如下趋势图：

count各列	执行时长
*	0.34
列1	0.61
列2	0.59
列3	0.64
列4	0.64
列5	0.64
列6	0.66
列7	0.65
列8	0.7
列9	0.71
列10	0.73
列11	0.75
列12	0.78
列13	0.79
列14	0.82
列15	0.81
列16	0.84
列17	0.92
列18	0.97
列19	0.98
列20	1
列21	1.01
列22	1.02
列23	1.06
列24	1.07
列25	1.11



结论：

COUNT(列)的速度随着列偏移的位置而越来越来越慢。COUNT(*)最快！

这是什么原理？因为优化器的 cost 算法是和列偏移量有关的，列越靠后，性能越低，count(*) 和列偏移量无关，所以性能最佳。

是不是有些吃惊？笔者做此试验无非就是因为了解了 Oracle 优化器的关于列偏移量的部分算法，然后才想到构造试验来证明。

这个试验并不难，无非就是三个要点：1.构造出有多个字段的表并填充数据；2.用结束时间减去开始时间的方式，在循环中分别记录下 COUNT(*)和 COUNT 各个列的用时；3.为了避免执行速度过快而影响判断，让每个 SQL 都执行了 1000 遍以上，从而减小外部环境造成的影响。

是不是做这个试验也是一个动脑的过程？然后，我们其实还可以推出一个结论，那就是最经常访问的列，最好设计在靠前的列偏移位置，这样能提升应用的性能！

17.3　全书完，致读者

哇，原来还有这么多延伸！老师，你收了我吧，今天我脑洞大开，愿意跟随你左右！

欢迎！

亲爱的读者们，至此全书就要结束了。

这本书安排的章节远远不够覆盖所有的 SQL 优化知识，具体的 SQL 优化内容也无法做到详尽地展开，一方面限于篇幅，另一方面也限于笔者的知识水平。不过，我认为详尽的优化百科大全实际对读者来说并不是最重要的。我将讹传辨识与思维探索作为全书的最后一个章节，就是想告诉大家：质疑探索比学习更重要！

然后，请你看看全书目录，难道你不觉得，SQL 优化不只是一门技术，更是一门艺术！将基于实战的优化方法论融合在美丽的思维导图里，我们看到的，是优化之美！

老师，你好像让我有点感动了！

嗯，有收获吗？

有，收获，不止 SQL 优化！

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

收获，不止SQL优化 抓住SQL的本质

敬彬兄再次出书，依然是脑图逻辑为先，用语通俗易懂，细节深入浅出。我仔细拜读了第1、2、17章，敬彬兄不仅将SQL优化需要使用的工具做了全面详实的介绍，更结合他在不同行业的实际案例，用诙谐笔法娓娓道来。强烈推荐给还在优化之路上奋斗的DBA、开发人员们，你定会如书名所言，《收获，不止SQL优化》！

杨志洪

"DBA+社群"发起人，新炬网络首席布道师，Oracle ACE，《Oracle核心技术》译者

继上一本《收获，不止Oracle》书后，由梁敬彬、梁敬弘兄弟合作的《收获，不止SQL优化》再次问世了。感慨两位兄弟在技术之路上孜孜不倦的追求和无私的分享。

梁敬弘是我的学生，学业专精，为人善良热心，是一个非常不错的小伙子。哥哥则精于实战，善于总结，在业内是一个极为知名的数据库专家。两位兄弟联手完成的新书必然是数据库领域的精品，值得大家去学习和体会。在此，预祝本书的出版获得成功，同时也祝兄弟二人在事业上取得更大的成就。

黄连生

清华大学计算机系教授，博士生导师

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：张月萍
封面设计：王　乐

上架建议：计算机数据库

ISBN 978-7-121-31436-0

定价：88.00元