ORACLE 性能优化思维与方法论

贾代平　吴丽娟　著

科学出版社

北　京

内 容 简 介

随着数据量的不断增加，传统的数据服务系统面临性能退化的挑战。作者在理论和实践的基础上从数据服务的核心层、逻辑层、物理层、实例层、管理层深入探讨 ORACLE 系统的性能调优问题，向读者展示其中的技术原理、手段和思维方式，并试图上升到方法论的层面揭示性能调优的技术路线图。利用此路线图，读者在处理性能问题时，按图索骥就可以找到解决工程问题的法门。核心层介绍了执行计划（Execution Plan）；逻辑层、物理层分别从数据的存储结构和数据的访问方式上探讨引发性能问题的主要方面；实例层和管理层从数据服务系统整体运行的角度探讨性能问题。本书虽然结合 ORACLE 系统探讨性能问题，但书中讨论的调优技术、思维方式和解决问题的系列方法具有一般性，可供其他数据服务系统参考和借鉴。

本书读者对象为数据服务系统的规划、设计人员，ORACLE 系统的技术专家和高级用户；同时对数据管理、应用开发、系统维护等 DT 时代相关的 IT 从业人员也具有重要的参考价值。

图书在版编目(CIP)数据

ORACLE 性能优化思维与方法论/贾代平, 吴丽娟著. —北京：科学出版社，2017

ISBN 978-7-03-055672-1

Ⅰ.①O… Ⅱ.①贾…②吴… Ⅲ. ①关系数据库系统 Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字（2017）第 292821 号

责任编辑：赵丽欣 常晓敏 / 责任校对：马英菊

责任印制：吕春珉 / 封面设计：东方人华平面设计部

科学出版社 出版

北京东黄城根北街 16 号

邮政编码：100717

http://www.sciencep.com

印刷

科学出版社发行 各地新华书店经销

*

2017 年 12 月第 一 版 开本：787×1092 1/16

2017 年 12 月第一次印刷 印张：18 1/2

字数：421 000

定价：69.00 元

（如有印装质量问题，我社负责调换〈 〉）

销售部电话 010-62136230 编辑部电话 010-62134021

前　言

信息系统的本质是数据服务，而 ORACLE 是数据服务领域当之无愧的主角。本书研究的是基于这个主角的性能问题。欢迎进入 ORACLE 性能调优的世界。

传统地，人们构建信息系统，关注的是“功能”（Function），系统设计和应用开发主要为“功能”服务。随着数据量的增加和人们对“功能”品质要求的提升，“性能”（Performance）问题逐渐浮出水面，而且在各种数据服务系统百花齐放的情况下，性能成为在市场中致胜的关键要素之一。还记得中国铁路客户服务中心网站（www.12306.cn）的经历吗？系统花费巨资开发的火车票务服务的“功能”，却没有经受住“性能”的考验，投入运行的前三年，曾几度瘫痪。如果不是独家经营，消费者早就会将其弃如敝屣了。这是数据服务系统性能问题的经典案例。

关注“性能”问题，不仅可以提升数据系统的服务质量，还可以实现 IT 资源的有效利用，减少不必要的投资。在实践中见到很多的类似案例，如当系统遭遇性能瓶颈时，建设者往往考虑的是增加投资、实施服务器扩容。事实上，只有当必需的某种吞吐量（Throughput）得不到满足时，系统才需要扩容。大多数情况下通过性能优化、提高资源的均衡利用率，性能瓶颈可以得到缓解甚至消除。

“性能”问题虽然与“功能”有关，但它在功能问题之上。从软件的角度，对于功能，需要调试（Debug），而对于性能，需要调优（Tune）。功能与性能、调试与调优，都只是一字之差，但其思维方式和处理方法却存在显著区别。调试的目的是发现软件功能上的缺陷（Bug），可以让软件在运行过程中暂停，以进一步观察内部的细节信息，因此它可以是静态的；而调优的目的是发现软件在运行中的潜在问题，需要在过程中发现瓶颈，因此它是动态的。调试是解决局部的问题，而调优大多数情况下是要解决系统性的问题。在调优实践中，往往缓解或消除了某一环节的问题，却可能会引发另外一些环节的问题。消除了一个瓶颈，另外一些瓶颈又会出现。因此在处理性能问题时，既要熟悉系统的每个环节，又要对不同环节之间的协作方式有充分的了解；既要熟悉局部运行的细节，又要理解不同环节之间的轻重缓急。由此可以看出，处理调优问题要比处理调试问题难度更大，对 IT 人员的要求更高。工程实践中的调优过程不太可能一蹴而就，本质上是一个动态迭代的过程。

曾几何时，在注重“功能”的年代，性能问题是个定性的问题，与用户的主观感受有关，性能的优劣仅影响用户体验而已。ORACLE 很早就显示出，性能问题不仅仅是用户体验问题，而是影响到用户选择的问题。因此在 ORACLE 产品升级过程中，越来越重视性能优化技术的开发。经过几个版本的更新与发展，ORACLE 系统的性能优化问题已经从早期的定性的问题发展为可以定量研究的科学问题。目前，ORACLE 系统的大部分的性能问题都可以定量地去度量，用数据说话。

基于对性能问题的上述理解，本书将 ORACLE 性能调优问题归纳为核心层、逻辑层、物理层、实例层、管理层 5 个不同的层次，并进一步分解为 10 章内容（第 2 章至第 11 章），每一章讨论不同的优化方法与技术。首先，将系统全局的观点贯穿全书，在介绍具体的技术细节之前，注重讨论调优的思维方式和解决问题的一般方法。其次，都会在每一章节讨

论不同的概念、模型、观测和实验，以便展示其中的技术细节。下面简要介绍各章的主要内容。

第 1 章介绍基于 ORACLE 系统性能优化的总体框架，阐述性能与资源消耗的基本概念及其相互关系。第 2 章（核心层）讨论 ORACLE 性能调优的度量及其主要的跟踪技术，这是分析性能问题的基本手段。第 3 章（核心层）集中讨论 ORACLE 系统的优化器及其决策环境，这是数据服务系统的核心部件，它直接决定用户指令的执行效率。第 4 章（逻辑层）包括对优化器输出结果的分析与讨论、查看用户指令的执行过程、理解和分析执行过程的每一个环节及其资源消耗，这是性能调优的基本功。第 5 章（物理层）从数据库对象的物理存储的角度解析数据访问的性能问题，重点解析堆表的存储结构及影响其访问性能的主要方面。

现代数据服务系统离不开索引的支持，第 6 章（逻辑层、物理层）专门讨论 B 树索引的结构及其访问方式，是软件开发人员掌握 ORACLE 性能优化的关键一环。第 7 章（逻辑层、物理层）在堆表和 B 树索引的基础上，解析 ORACLE 系统常用对象的段（Segment）结构及其数据访问。第 8 章（逻辑层、实例层）集中讨论多用户环境和多进程环境的特殊问题，它们既可以阻碍性能，也可以提升性能，关键看如何利用。第 9 章（实例层）专门讨论 ORACLE 系统的运行结构，应重点关注内存结构的使用及对 ORACLE 系统性能产生的影响，这个影响是全局性的。第 10 章（实例层、管理层）等待事件接口是一种全新的研究性能问题的角度，它是基于时间因素的考量。利用 OWI 可以有效地观察 ORACLE 系统性能问题的瓶颈。第 11 章（管理层）讨论基于性能问题的内部统计手段及其性能问题的辅助诊断，它是基于统计意义的分析手段。

由于 ORACLE 系统的复杂性，面对调优问题时很多从业人员处于盲人摸象的状态。即使是某些细分领域的技术专家，在面对一些性能上的疑难杂症时，也会产生强烈的无力感。这就需要系统的研究与学习。作者希望本书是一座桥梁，通过向读者阐述 ORACLE 系统不同层次的调优技术，实现理论与实践的统一、原理和方法的贯通，使读者最终能够形成对 ORACLE 系统性能问题的全局性的认识，从而大大提升处理实际问题的能力。就像一位武林高手，游刃于各个门派和招式之间，融会贯通，出神入化，最后达到无招胜有招的高度。

本书的出版得到科学出版社的大力支持，在此一并致谢。衷心地希望读者能够通过本书，逐级跨越 ORACLE 性能调优的核心层、逻辑层、物理层、实例层、管理层的系列技术壁垒，迈上一个新台阶，达到一览众山小的境界。

由于作者水平有限、认识有限、时间有限，书中难免出现不妥或疏漏，希望读者给予批评指正。若能获得及时的反馈，作者深表谢意。联系邮箱：jiadp@oraclechina.org。

作　者

2017 年夏

目　录

第 1 章 ORACLE 性能优化思维概述

1.1 IT 系统的“功能”与“性能”

今天的企业和机构比以往任何时刻都更依赖于 IT 系统提供的各类服务，其中，数据服务是构建 IT 系统的核心价值。数据服务系统的“性能”关系着日常生活和工作的方方面面，关系着工作效率和生活质量。长期以来，IT 领域习惯于关注数据服务系统能够实现的功能，而一定程度上忽视了数据服务系统的“性能”。没有满足用户需求的“性能”表现，服务的“功能”往往失去意义，这在一些大中型数据服务系统中表现尤为突出。事实上，大多数的数据服务系统在实现了业务处理功能、投入运行的那一刻起，其“性能”表现就会直接影响用户体验和用户满意度。这里大家所熟知的典型案例就是中国铁路客户服务中心（www.12306.cn），该系统在 2011 年正式投入运营后，其性能表现广受用户诟病，其售票系统在第一个“春运”期间几度瘫痪，无法提供正常的购票服务。无独有偶，2009 年 6 月 26 日，美国 TMZ.com 网站率先报道了摇滚歌星迈克尔·杰克逊（Michael Jackson）病逝的消息，大量歌迷涌入网站，访问量骤增，导致该网站陷入瘫痪状态。类似的案例在国内外都曾出现过，概括地说都是由于“负荷”增加、“性能”减退，最终导致“功能”失效。

数据服务系统的“功能”问题是一个纯粹的技术问题，而数据服务系统的“性能”问题不仅仅是一个技术问题，也是一个 IT 工程问题。随着服务范围的扩大，用户数量的增加，“性能”问题几乎困扰着每个数据服务系统的运营方。从纯技术的角度，所有的数据服务系统都是高度相似的，这就给系统地研究数据服务系统的“性能”问题提供了科学依据。

本书讨论的对象是基于 ORACLE 的独立数据服务系统，它是一个逻辑上的概念，不仅仅是指物理上独立的系统，也可指工程中的集群服务系统、云端服务系统等。不论物理上的外在形式如何，从应用的角度，它是一个具有确定边界的数据服务系统，为用户服务的内在处理机制是完全一致的，因此关于此类系统的“性能”问题，完全可以用一种统一的观点去描述、研究。

1.2 服务性能与资源配置

1.2.1 性能问题的一般情形

在 IT 项目的工程实践中，服务资源的配置具有一定的盲目性，在这里，项目设计人员的工程实施经验起着主导作用，往往要等到项目上线后才能发现服务资源配置不足或资源配置不均衡（导致潜在的性能瓶颈）。研究性能问题的意义在于：根据“性能”要求预先配置服务资源，或根据已知的服务资源确定可以达到的“性能”目标，让“性能”指标在服务资源的配置中起决定性作用，避免服务资源配置的错位或浪费。

数据服务系统的“性能”问题起源于如下两个方面：一是随着系统负荷的增加，用户请求的响应时间开始缓慢增加，用户体验开始逐步变差，当超过某个临界值时，用户选择离开，这在面向市场的信息服务行业中是致命的；二是一旦系统负荷突破了某个所能承受的区域，数据服务的性能急剧下降，负荷的微小变化都会使“性能”变得敏感，最终系统不胜负荷，导致系统服务停滞或挂起（Hang）或“功能”的实现被无限拖延，最终导致数据服务系统瘫痪或死机。就目前国内外此领域的服务现实而言，第一种情形时刻都在发生，由于是正常的定量变化，大多数情况下无须特别关注；对于第二种情形，由于数据服务方尚没有系统的、完善的应对负荷急剧增加的策略，系统瘫痪的情形时有发生，在 IT 行业的新闻中常会看到此类报道。也就是说，在目前的条件下，完全杜绝这类事件的发生几乎是不可能的，所能做的是尽可能地降低此类事件发生的概率。

1.2.2 业界的服务水平管理

目前 IT 行业和学术界对性能问题的研究主要集中在计算机系统的服务水平管理（Service Level Management，SLM）上，它属于介于纯粹的技术学科和工程管理学科之间的交叉学科。SLM 涵盖如下五个方面的内容。

1）容量管理（Capacity Management）：在满足特定数据服务需求的情况下应有的容量支持，它依赖于性能指标、负荷监控、应用的业务处理等，包括存储容量、网络容量、计算容量、I/O 带宽等。

2）高可用性（High Availability）：现代数据服务系统一旦投入运行大多需要保持 7×24h 运行，高可用性研究的是保障数据服务系统持续运行的一系列解决方案，如构建数据服务集群（Cluster）、服务冗余（Service Redundancy）、主从冷热备用等。

3）系统保障与恢复（Data Guard and Recovery）：此方向关注的是数据服务系统的可靠性、可维护性、备份策略与可恢复性、应变能力，核心是通过各种技术手段确保业务数据的安全。

4）风险评估（Risk Assessment）：系统运行过程中，负荷和用户需求都是动态变化的，而服务资源则是相对静态的，系统需要以不变应万变，确保事先设定的服务标准，显然这充满不确定性。风险评估包含识别或预测潜在的风险和缓解风险两层含义，此处当然包含“性能”不满足用户需求的风险。

5）IT 投资分析（IT Investment Analysis）：此方向研究的是 IT 基础设施的成本效益核算（Cost Benefit Accounting，CBA）和投资回报率（Return On Investment，ROI），这是数

据服务“工程”区别于“技术”的显著方面。

显然，上述五个方面都与本书要研究数据服务的“性能”问题密切相关，国内外对这些分支的研究各有侧重。

首先，容量问题是制约数据服务“性能”的一个方面，然而单一的容量无法解决现实中遭遇的“性能”问题，甚至出现系统经过扩容后性能不但没有增加反而下降的案例。

其次，高可用性是数据服务系统提供连续不间断服务的保证，此方向的研究重点是避免系统出现各种极端的情况，防止意外事件的发生，如系统计划外停机、死机、崩溃等，另外某些高可用性方案也与容量密切相关。例如，集群数据服务系统既可增加系统的容量，也可提高系统的可用性。

再次，系统保障与恢复方向的重点在于“数据”本身，关注用户数据的安全，避免数据在意外情形下的不可访问和数据丢失。作者涉足此领域较早，曾就 ORACLE 系统撰写了一篇关于数据恢复的论文，讨论在各种意外情形下的完全数据恢复（零数据丢失）问题，被业界和同行广泛引用。

接着，风险评估方向探讨的是在满足服务需求方面可能面临的风险，本质上讨论的是未来的“性能”和未来的“负荷”之间的关系。在目前的学术研究和行业实践中，对于评估的结果，比较通行的做法是给出风险的显著性等级，显然这对于精细化的服务管理是远远不够的，需要将定性风险转换为数字化的“性能”与数据服务系统要素之间的定量关系。

最后，IT 投资分析研究的是在有限投资的约束条件下数据服务系统的实施情况，是一个典型的工程问题，目前基于数据服务系统的财务分析（包括 CBA 和 ROI）主要面向 IT 投资决策领域。

本书讨论的主题是基于 ORACLE 系统的性能优化问题，研究的是在资源限定的情况下如何改善或优化用户的性能体验。

1.3　性能研究关注的侧重点

1.3.1　关注数据访问的执行过程

用户数据访问的“性能”取决于用户请求在系统内部的执行过程。概括地说，用户的需求是 What to do，而这个过程是 How to do，如果能够详细探究这个过程，性能问题的细节信息就会展现在开发者面前。

对于性能问题，无论是由内部资源争用（Contention）还是由内部资源耗尽（Exhausted）导致的“性能”问题，最终都表现为用户请求响应时间的增加，即请求指令执行过程的延长。而这里的执行过程可以划分为计算过程和等待过程两个方面，其中计算过程反映了数据服务系统为响应用户请求实施的一系列内部操作及其资源消耗，根据现代资源调度和最优决策理论，可以将其转换为某些定量的指标，如计算路径（Computing Path）、计算成本或计算代价（Computing Cost）等。由于“性能”问题是一个多变量约束的问题，通过对计算过程中获取的路径和代价的分析，可以探索这个计算过程的最优路径和最小代价，将数据服务系统的“性能”问题转换为具有普遍意义的最优化问题。

探讨执行过程的另外一个方面就是分析等待事件（Wait Events）。前面的计算过程是一个资源消耗的过程，而这里的等待过程是一个资源需求不能满足的过程，即资源等待。在

之前的研究中发现，每当数据服务系统出现显著的“性能”问题时，其内部总是伴随着大量的等待过程（如果出现严重的等待，业界有一个直观但非正式的词——Hang）。在一个高度并发的数据服务系统中出现资源等待的情形不可避免，关键是首先要了解这些等待的情形，其次要了解出现这些内部等待发生的原因，在此基础上才有可能设法减少这些内部等待出现的次数，或缩短这些内部等待持续的时间。

这是从数据服务内在特性的角度研究“性能”问题的两个方面，即计算与等待。如果能够通过对资源的优化和调度，一方面减少资源的消耗，另一方面减少内部出现的等待，那就可以在性能优化方面做出改进。

1.3.2 关注负载和性能之间的动态关系

1.3.1 小节从数据服务系统的内部研究“性能”问题，另外一个角度是从数据服务系统的外部研究“性能”问题，即关注数据服务系统的输入/输出关系，这里的“输入”是指用户的各种数据处理请求给系统带来的“负载”，“输出”是指人们关心的“性能”。

对于一般意义上的计算机系统，“负载”和“性能”都是定性的概念，往往随着讨论问题的侧重点不同而有所不同。要研究数据服务系统的动态特性，首先要能够精确地计量“负载”和“性能”。对于本书要研究的数据服务系统，“负载”和“性能”的真正含义是什么？目前关于“负载”的说法众说纷纭：指某一时刻的并发会话（Session）数，单位时间内的事务处理的数据量，单位时间内的事务到达率，用户请求导致的逻辑读和物理读的数量，消耗的 CPU 时间，数据的查询量，数据的更新量，队列的长度或等待时间等。对于“性能”的描述同样如此：用户请求的响应时间，系统的 CPU 利用率、I/O 带宽利用率，缓存的命中率，事务处理的平均时间，吞吐量，单位数据块的读写时间等。

这就需要结合数据服务系统的特性，研究如何统一化、归一化地描述系统的“负载”和“性能”，并能适应于不同应用环境的需求；同时为了方便重复实验和数据采样，还有必要研究如何模拟一种用于“性能”测试目标的标准负载。与此同时，不论是“负载”还是“性能”，同样需要研究它们精确的计量方法。

通过对“负载”和“性能”的系列描述，可以将数据服务系统抽象为一个复杂的动态系统，而这里的“负载”和“性能”就是动态系统的输入和输出。如果把不同侧面的“负载”描述作为这个动态系统的输入向量，而把不同角度描述的“性能”计量作为输出向量，通过研究输出向量序列对输入向量序列的依赖关系，理论上就可以很好地把握数据服务系统的动态特性。这里，统计建模和序列计算理论（包括时间序列的理论与方法）都可以用来处理数据服务系统的输入向量序列、输出向量序列及其输入/输出关系。

1.4 负载—响应时间曲线

诚然，在一个复杂的数据服务系统里，实现性能优化需要关注的“点”有很多，而且这些“点”不是一成不变的，随着讨论问题的变化而变化，如讨论系统整体的性能、讨论系统某个局部的性能等。在讨论性能优化的一般性思维时，是否存在需要普遍关心的普适物理量呢？回答是肯定的。就性能表现来说，IT 系统无论是整体还是局部，其运行的机制是相似的，有输入、有资源消耗、有输出，而且任何输出都是在特定输入量（压力或负载）、限定资源消耗的情况下产生的。

直观地，在讨论性能优化时，大多数开发人员和用户关注的是指令执行的响应时间（Response Time，RT），但这个响应时间是有前提的，一是并发性，二是资源消耗。开发人员在测试“性能”时往往隐含着两个假设：并发性没有限制、资源是无限的，而这两项假设在实际上线的系统中都是不成立的。因此，考虑响应时间没有问题，性能优化需要考虑在一定前提条件下的响应时间。这个“前提条件”如何统一描述呢？笔者认为有一个很好的物理量来度量这个前提，即吞吐量（Throughput Capacity，TC）。

吞吐量是个通用的描述，可以把它用在任何服务系统或服务单元上。吞吐量是指网络、设备、端口、虚电路或其他设施单位时间内成功传送数据的数量（以比特、字节、分组等测量）。很显然，这个定义是对网络而言。对于性能优化，完全可以借用这个概念，推广到一般情形：单位时间内完成的“操作”数量都可以称为吞吐量，如机场的吞吐量指单位时间内发送的旅客数量，挖掘机的吞吐量指单位时间内开挖的土石方容量等。在 ORACLE 数据服务系统里，吞吐量指代的“操作”有事务处理（Transaction）、解析（Parse）、逻辑 I/O、物理 I/O、用户调用（User Call）等，它综合反映了系统的并发性和负载（Load）。

吞吐量与资源消耗的局部模型如图 1-1 所示。

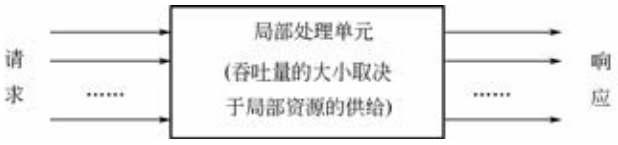

图 1-1　吞吐量与资源消耗的局部模型

根据图 1-1，如果把这里的“局部处理单元”看作一段管道，则请求（Request）是管道的输入，响应（Response）则是管道的输出，吞吐量则相当于管道能够通过的单位流量，它取决于局部处理单元的资源供给。在 ORACLE 数据库系统中，SQL*Net、Listener、SGA、PGA、Server Process、LGWR、DBWR、CPU、I/O 系统等都有各自的吞吐量限制，而这些处理单元相当于串联在一起，形成一个完整的数据处理回路。显然，其整体的吞吐量取决于回路中最窄的那段管道。如果这些管道提供的流量（吞吐量）比较均衡，则数据处理就不会出现肠梗阻现象，即系统性能的瓶颈。

图 1-1 中由请求发出到响应输出的时间间隔就是响应时间。一般来说，在单元遭遇资源瓶颈之前，响应时间与负载的大小几乎无关。随着负载的逐渐增大（单元提供更高的吞吐量），处理单元消耗的资源上升，当开始遭遇资源瓶颈时，并发访问的任务在单元内部出现各种资源访问的冲突，出现各种等待事件，等待时间随之增加，导致响应时间增加，此时负载-响应时间曲线开始出现拐点，如图 1-2 所示。

图 1-2 中的横坐标描述的是系统负载，此处用吞吐量（trx/ms）代表单位时间内的事务处理数量，由图可知，响应时间存在如下公式：

$$\text{Response Time} = \text{Service Time} + \text{Queue Time}$$

响应时间=服务时间+等待时间

这里的性能曲线（负载-响应时间曲线）给出了典型的 IT 系统性能表现，但没有给出资源消耗的信息。当单独考察某一个用户操作的响应时间时，很容易通过提高内部执行的并行度来降低响应时间，这是典型的通过增加资源消耗达到调优的目的的表现，但这一手段在大多数 OLTP 系统中是行不通的，因为系统资源是有限的，一旦用户操作的资源消耗

增加了，必然影响到其他操作的资源消耗，导致其他操作响应时间增加。因此在性能优化的实践中要结合具体的系统类别做不同的处理。

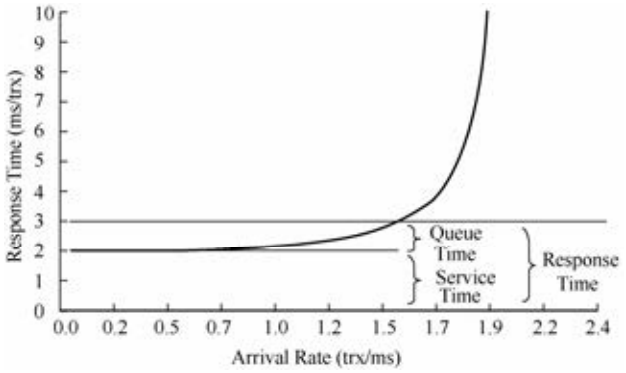

图 1-2　典型的负载-响应时间曲线

对于大多数的 OLTP 系统，需要追求的一般目标可以描述为，在满足特定吞吐量的前提下尽可能降低响应时间。这就要求响应时间在用户可接受的范围内尽可能减少资源的消耗，而不是通过增加资源的消耗来降低响应时间。对于 OLAP 或批处理任务，需要追求的一般目标可以描述为，在系统允许的范围内尽可能增加资源消耗来降低响应时间。由于此时系统没有吞吐量的要求，正在执行的任务总是希望以独占的方式利用系统的所有资源，所以其优化目标与 OLTP 有着显著的不同。

1.5　性能优化的一般方法

这里介绍一般性的方法。从系统架构和方法论的角度，ORACLE 的性能调优总体上可以划分为两大类：自下而上的优化方法和自上而下的优化方法。如果运用得当，两类方法都可以实现性能调优，只是它们各自侧重于不同的优化情境。

1.5.1　自下而上的优化方法

这类方法通常适用于已经部署完毕、投入运行的系统。此时，应用系统的架构已经定型，只能从局部的技术细节入手，发现并缓解性能瓶颈。

1）查看 ORACLE 系统的各类性能指标、利用率等。
2）检查系统内部的等待事件接口（Wait Event Interface）。
3）衡量系统的 I/O 特性与性能。
4）根据上面的检查综合判断性能问题的范围，如语句级、会话级或实例级等。
5）确定系统资源消耗的 Top N 项目，如 CPU 消耗、I/O 消耗等。
6）研究特定数据访问的执行计划（Execution Plan）。
7）研究相关数据库对象的特性与存储，表、索引等。
8）研究数据集（Result Set）之间的联接方法、访问方式等。
9）综合上面的研究内容分析产生性能问题的原因。
10）根据原因确定缓解或消除性能瓶颈的解决方案。

11）模拟测试，分析解决方案带来的性能变化和可能的影响。

12）方案实施，并记录完整负载的运行情况。

1.5.2　自上而下的优化方法

理想的情况是系统在架构设计（包括软硬件平台的选型）、项目开发、模块测试、系统集成、产品上线等环节及时应用性能优化的思想，考虑系统上线后在响应速度、吞吐量、高可用性、可扩展性、易维护性、备份与恢复、容灾能力等方面的综合要求，以提高系统的整体性能表现，而不仅仅局限于满足具体数据访问的性能要求。

对于 ORACLE 而言，将数据访问的性能问题划分为图 1-3 所示的几个层次，自上而下的优化方法可以根据每个层次的具体需求进一步细化优化目标。

1. *应用层*

该层的优化目标是以最小的应用负载实现用户的需求，在满足用户需求的前提下尽可能地要求数据库端“少做事”“更聪明地做事”，而不是向数据库端发送低效的数据处理请求。要实现这个目标，一是要优化数据库设计，包括根据业务需求优化数据库建模；二是要优化应用代码，提高代码请求的效率，包括查询重写、改进数据处理的方式等。

应用层是数据库负载之源，如果能够实现应用层的优化，那就是从源头降低数据库端出现性能问题的可能性。实践中，一个蹩脚的、低效的 SQL 程序拖累一个数据库系统、甚至拖垮一个数据库系统的案例时有发生。

2. *逻辑层*

应用层优化后，到达 DBMS 端的请求就是必不可少的“纯任务”，逻辑层的任务是如何高效地完成这些任务。这里的主要目标有两个：一是用户指令的有效执行，包括 SQL 语句的解析、执行计划的分析和优化、数据访问路径的选择、数据集之间的关联方法的优化等；二是要关注并发处理环境，降低多用户环境的资源访问问题，如最小化锁冲突、锁存器争用、减少内部等待等。

3. *实例层*

用户需求经过应用层、逻辑层优化处理后，最终的功能实现都是在 ORACLE Instance 中完成的。“实例”提供了指令的运行环境，实例层的目标就是要优化数据处理环境，包括两个方面的环境：一是内存环境，合理化配置 ORACLE DBMS 使用的各种内存空间，如共享池、数据缓存、日志缓存、排序区、缓存融合等；二是进程环境，Server Process 及各种后台进程的配置，关注进程的运行效率和吞吐量等。这部分内容就是传统的 ORACLE 数据库优化的内容。

4. *物理层*

绝大部分的用户请求都会带来或多或少的物理层操作，即针对磁盘的读写（物理 I/O）。一般来说，相对于逻辑的读写，物理 I/O 要慢得多。用前面在讨论吞吐量时的“管道”说，对于大多数系统而言，物理层的访问是整个数据访问路径中最“窄”的一段，因此提高 I/O 的效率，往往对数据库的性能优化有事半功倍的效果。这里面也涉及两个主要方面：一是避免物理 I/O 的带宽（物理 I/O 吞吐量）成为性能瓶颈，尤其是优化数据库物理读的操

作；二是优化 ORACLE 数据库对象（如数据段、索引段、临时段等）的物理存储，提高访问效率。

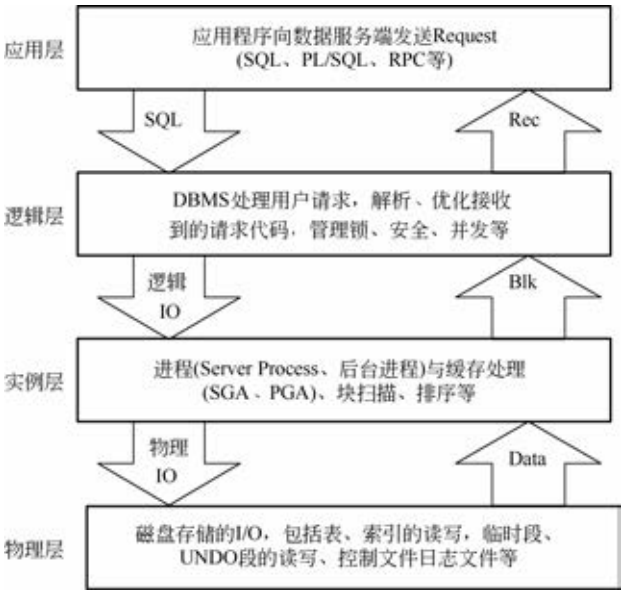

图 1-3　ORACLE 数据访问的纵向结构

1.6　服务性能的基本问题

数据库系统出现性能问题，其根本原因是系统资源出现瓶颈，这里的资源瓶颈包括硬件瓶颈和软件瓶颈。内存、CPU、I/O 是硬件资源，如果这些资源出现瓶颈，则系统出现硬件瓶颈。软件瓶颈则是由于软件设计的数据处理不合理导致性能问题。很多情况下，硬件瓶颈是由于应用软件的不合理导致的，这里说的性能调优其目标是在硬件资源有限的情况下，通过调整应用和配置，使应用软件配合硬件资源协调运行。

1.6.1　内存问题

现代计算机系统的内存操作要比磁盘操作快得多，因此一般认为 I/O 瓶颈是计算机系统常见的性能瓶颈之一。数据库系统性能调优的目标之一是将所有的数据处理置于内存中完成，因此一般认为增加内存能改善数据库系统的性能。但如果认为所有的数据处理都在内存中完成，数据库系统就不存在性能问题，是不妥当的。

充足的内存资源是保障数据库系统正常运行的必要条件，而非充分条件。数据库系统性能调优的目标是确保应用软件和数据库之间协调运行，提高性能的关键是数据处理在内存中如何运行，如果运行在内存中的数据处理出现大量等待、如果数据处理导致 CPU 利用率过高、如果数据处理在内存中大量无效地扫描数据，则一定会在某种程度上导致性能问题，此时再充足的内存也无济于事。

用户最终关心的是数据处理的结果，而非数据处理的过程。尽可能地提高数据处理过程的效率，从源头上减少不必要的数据访问，是提高性能的关键，这样既可降低对内存的

需求，又可提高数据处理的性能。

1.6.2　CPU 利用率

宏观上，CPU 利用率是数据库系统运行状况的表现之一，现代分时多任务操作系统对 CPU 都是分时间片使用的，如 A 进程占用 10ms，B 进程占用 30ms，然后空闲 60ms，再是 C 进程占用 20ms，D 进程占用 20ms，空闲 60ms。如果在一段时间内都是如此，那么这段时间内的 CPU 利用率就是 40%。

过高的 CPU 利用率通常会导致数据库系统性能的下降，但 CPU 利用率低并不是没有性能问题，所以并不是 CPU 利用率越低越好。CPU 利用率低只是表示 CPU 无所事事，所有的数据处理必须要借助于 CPU 才能完成。如果数据库系统出现性能问题，而与此同时 CPU 利用率又很低，这说明数据处理的方法或效率出现问题。要提高数据库系统的性能，一般需要有必要的 CPU 利用率做配合，否则数据库系统不可能有足够的吞吐量。

对于 OLTP 系统，存在大量的频繁的短时事务处理，正常情况下 CPU 利用率不应太高；而对于 OLAP 系统，往往存在长时间的批处理操作，此时应该充分利用 CPU 资源，高的 CPU 利用率是提高数据库系统处理效率的保障。

1.6.3　I/O 问题

磁盘的读写是 I/O，通过网络的输入与输出也是 I/O，数据库系统同时存在这两个方面的 I/O，一方面数据库存储于磁盘介质，系统在运行过程中要读写数据库的物理存储，另一方面，应用系统（包括应用服务器）通过网络访问数据库系统，两者的交互是通过网络 I/O 完成的。这两个方面的 I/O 都有可能成为数据库系统性能上的瓶颈。

ORACLE 数据库系统中，如果出现 I/O 瓶颈，大多数情况下是磁盘 I/O 瓶颈，或者说磁盘 I/O 是数据库性能调整的永恒话题，因为较慢的磁盘 I/O 或多或少是影响数据库系统性能的因素。如果系统出现明显的磁盘 I/O 瓶颈，在不改变磁盘和磁盘配置的情况下，主要有两种解决方法：一是分散数据库的物理存储和数据访问，以缓解磁盘 I/O 瓶颈；二是通过优化数据库系统配置和应用系统，减少磁盘 I/O 的访问量。

曾经在 IBM RS/6000（AIX）主机的 ORACLE 服务器上发生这类现象：磁盘 I/O 负载过重，超出 AIX 异步 I/O 请求量的上限，导致 ORACLE 数据库的数据文件脱机。为了保护磁盘，AIX 操作系统设置磁盘异步 I/O 请求量上限有它的合理性，处理这种故障的应急措施可以是关闭 AIX 异步 I/O 或分散数据库的物理存储，治本的方法还是要在 ORACLE 中寻找导致过量 I/O 的原因，优化数据访问，降低磁盘 I/O 的量。

至于网络 I/O 是数据库系统之外的话题，在服务器的范围内，网络 I/O 通常存在于数据库服务器与应用服务器之间，或数据库服务器与 Web 服务器之间，一般情况下，这些服务器之间都是采用高速网络连接，不大可能出现网络 I/O 瓶颈，但也有例外。曾经有这样一个案例，网络故障导致双向网络传输的速率明显不同，其中一个方向的传输速率过低，导致应用服务器在批量处理数据时出现严重的性能问题，此时 ORACLE 数据库端出现大量 SQL*Net more data from client 等待事件，原本仅需几分钟完成的任务延长到几个小时，影响后续业务处理，用户无法接受。这在检查网络 I/O 瓶颈时需要引起注意。

1.6.4 高资源消耗的 SQL

与用户执行 SQL 有关的动态视图有 V$SQL、V$SQLAREA、V$SQLTEXT、V$SQL_PLAN、V$SQLSTATS 等。V$SQL 中包含了所有用户执行的所有的 SQL 信息，不同用户、不同会话执行的相同 SQL 的语义、执行计划可能会不同，这些 SQL 字面值相同（具有相同的 SQL_ID），通过不同的 CHILD_NUMBER 来区分。V$SQLAREA 中仅包含 SQL 语句的字面信息，忽略了相同 SQL 语句在执行会话、语义、执行计划上的不同，相同的 SQL 语句在 V$SQLARA 中仅以一行显示。V$SQLTEXT 以多个 Piece 的形式给出了 SQL 语句的完整文本，它通过字段 ADDRESS 和 HASH_VALUE 的联合可以唯一地标识一条 SQL 语句。V$SQL_PLAN 包含了 SQL 语句的执行计划信息，它通过 ADDRESS、HASH_VALUE、CHILD_NUMBER 三个字段（可以和 V$SQL 关联）唯一地标识 SQL 语句的执行计划。V$SQLSTATS 包含 SQL 语句的性能统计信息，通过 SQL_ID 和 PLAN_HASH_VALUE 可以唯一地标识一条 SQL 语句的性能统计数据。V$SQLSTATS 视图的字段是 V$SQL 和 V$SQLAREA 字段的子集，即视图中的内容来自于它们，但 V$SQLSTATS 视图中的数据比 V$SQL 和 V$SQLAREA 保存更持久。

当需要根据各种资源消耗查找 Top SQL 时，可以根据不同的性能统计字段查询 V$SQL，该视图不仅包含用户提交给数据库的 SQL 语句及其文本，而且还包含这些语句的性能统计数据。表 1-1 所示为资源消耗与 V$SQL 视图相关字段的说明。

表 1-1 资源消耗与 V$SQL 视图相关字段的说明

资源消耗	字段及其说明
执行代价 COST	OPTIMIZER_COST 优化器给出的执行代价
CPU 消耗	CPU_TIME、ELAPSED_TIME 每次执行的 CPU 耗时= CPU_TIME/EXECUTIONS 每次执行消耗的时间= ELAPSED_TIME/EXECUTIONS
I/O 消耗	DISK_READS 每次执行的磁盘读次数= DISK_READS/EXECUTIONS
内存消耗	BUFFER_GETS、SHARABLE_MEM（消耗 Shared Pool） 每次执行的缓存消耗= BUFFER_GETS/EXECUTIONS
语句解析消耗	PARSE_CALLS 该语句（硬）解析的次数
排序消耗	SORTS
检索记录数	ROWS_PROCESSED、FETCHES

TOP SQL 查询示例：

1）查询最消耗 CPU 时间的前 10 个 SQL 语句。

```
SQL>select * from
  2  (select sql_id,child_number from V$SQL order by cpu_time desc)
  3  where rownum <=10;

SQL_ID          CHILD_NUMBER
------------- ------------
db78fxqxwxt7r          2
```

```
96g93hntrzjtr          0
aq1vwa734ww0j          0
04xtrk7uyhknh          0
4gb7r5dm6hnzs          0
cvn54b7yz0s8u          0
aq1vwa734ww0j          1
d92h3rjp0y217          0
30ndg3839521y          1
7vgmvmy8vvb9s          0

10 rows selected.
```

2）查询消耗磁盘读次数最多的 10 个 SQL 语句。

```
SQL>select * from
  2  (select sql_id,child_number,
  3  round(disk_reads/executions) dsk_rds
  4  from V$SQL where executions >0 order by dsk_rds desc)
  5  where rownum <=10;

SQL_ID          CHILD_NUMBER    DSK_RDS
------------- ------------ ----------
b7jn4mf49n569       0              953
24hc2470c87up       0              474
24b3xmp4wd3tu       0              297
cfz686a6qp0kg       0              207
c2p32r5mzv8hb       0              108
bsvpwnht1m9bm       0              103
bxywuzvtp6wjg       0              90
73z18fnnbb7dw       0              88
gfjvxb25b773h       0              82
dyd4b36t1ppph       0              76

10 rows selected.
```

另外，ORACLE 提供了一个视图 V$SESSION_LONGOPS，记录了所有超过 6s 的会话操作，如执行备份与恢复操作、统计信息的收集等，包括执行超过 6s 的 SQL 语句（需要设置 TIMED_STATISTICS 为 True 并需要统计信息的支持）。可以通过该视图获得响应时间较长的 SQL 语句信息，通过字段 SQL_ADDRESS 和 SQL_HASH_VALUE 与视图 V$SQL（或 V$SQLTXT）联接可以获得完整的 SQL 语句的文本。

1.6.5　引发性能瓶颈的应用问题

数据库系统的性能优化是一项复杂的系统工程。定性地说，一个高质量的数据库应用系统涉及系统硬件、网络、存储、平台软件（操作系统、DBMS、APP Server）、系统架构、应用软件等各个层面，涵盖系统需求分析、设计、开发、产品上线测试等各个环节，其综合目标是满足用户业务的响应速度和并发吞吐量，满足系统管理的高可用性、数据安全

性、数据容灾性、可扩展性、易管理性等。

然而，当一个应用系统投入运行后，系统运行的绝大部分因素和环节都已相对固定，需要面对的是一个在特性环境下的应用功能发挥问题，面对性能问题，根据实践经验，需要有下面几个基本的认识：

1）不要将系统性能的优化与提升全部寄托在系统硬件升级和系统层面。事实上国内大多数 ORACLE 系统的硬件部署和系统平台软件已经和国际先进水平同步了，整个系统能否发挥应有的效能，更多地取决于应用层面。就像演出剧目，华丽的舞台掩盖不了内容的苍白。

2）再好的硬件环境也解决不了数据库应用软件上存在的问题。在数据库应用软件开发过程中，开发人员往往注重功能问题，而不注意效能问题。系统投运后，效能问题成为主要矛盾。面对业务数据量的扩大、并发性的增加，效能问题往往呈现几何级数的恶变，这种情形下任何硬件的提升和扩容都很难满足效能的提升，只能是杯水车薪，收效甚微。

3）当出现性能问题时，需要注意区分性能问题的表象和根源。ORACLE 数据库系统出现性能问题时，往往伴随着 CPU 利用率高、内存吃紧、I/O 负载重、网络拥堵、大量等待事件、锁存器和缓存命中率等统计指标偏离正常范围等，这些都仅仅是现象，究其根源大多存在于应用层面，如果应用层面的问题解决了，这些现象就会随之得到缓解甚至消失，往往起到立竿见影的效果。

4）经济领域有一个大家公认的 20/80 现象，如 20%的富人占有 80%的财富，80%的利润往往来源于 20%的投资等，而且这种现象几乎无处不在，被称为 20/80 法则。研究数据库系统性能问题，这一规则也同样适用，主要表现在：80%的性能问题由 20%的应用导致；20%的优化技术能够解决 80%的性能问题。

根据大量工程实践中的应用总结，导致 ORACLE 数据性能瓶颈的应用层问题主要来自于以下几个方面。

1）查询的选择性（Selectivity of Query）：全表扫描与索引访问。避免对大表的全表扫描是防止 SQL 语句执行效率低下的主要措施之一，不仅包括查询，也包括 DML（如 Update、Merge 等）。很多原因导致 ORACLE 会执行全表扫描，如缺乏索引或索引失效、索引被抑制、不正确的索引类型与结构、不合理的多表联接顺序与联接方法、索引与联接的配合问题、不当的子查询（Sub-query）、事务设计与事务逻辑问题、优化器问题等。高效的数据选择性和有效的索引访问是提高 SQL 语句执行效率的关键。

2）游标的共享性（Sharing of Cursor）：解析（Hard Parse/Soft Parse）与绑定变量（Bind Variable）。在 OLTP 系统中，存在大量即时而频繁的查询和 DML 操作，如果用户游标没有很好的共享性，会导致频繁的 SQL 硬解析，消耗大量的共享内存（Shared Pool）和 CPU 资源，这种情形与系统性能和可扩展性紧密相关。如果系统存在大量 SQL 语句的重复解析，随着用户访问量的增加和应用规模的扩大，应用性能就会急剧下降，特别是 CPU 资源将很快枯竭。避免 SQL 频繁硬解析的主要手段是使用绑定变量。

3）数据的排序性（Sorting of Rows）：排序操作是一项典型的消耗计算机资源的操作，特别需要足够的内存空间。排序算法的复杂度介于 O(n)到 $O(n^2)$之间，ORACLE 对记录的排序也不例外，应尽可能减少对数据的排序，特别是要避免对大数据量的排序、基于磁盘的排序。另外，还要注意一些隐性的排序操作，如 distinct、union、group by、merge join 等。注意，大多数情况下，union 可以考虑改为 union all；当排序变得不可避免时，要提高

排序性能，一是要保障足够的 PGA 内存空间，二是可以充分利用索引对索引键值的排序性。

4）多表的联接性（Join of Tables）：有效地处理多表之间的联接关系是现代 DBMS 的精髓之一。ORACLE 提供多种处理多表联接的方法，不同的方法都有其适用范围，不适当地使用表与表之间的联接方法是导致 SQL 效率低下的常见原因之一。影响执行效率的主要因素有驱动表的选择（将限制性最强的表作为驱动表，必要时可以使用优化器提示）、多表联接的顺序（两两联接的顺序）、联接方法与索引的配合等。高效地处理多表联接是提高复合查询效率的关键。

1.6.6　OLTP 与 OLAP

数据库系统性能优化的目标与应用系统的类别密切相关。从应用的角度，各种数据库应用系统大致可以分成两大类：联机事务处理（Online Transaction Processing，OLTP）和联机分析处理（Online Analysis Processing，OLAP）。OLTP 是传统数据库系统的主要形式，主要是大量、频繁的事务处理，实时性要求高。OLAP 是数据仓库系统的主要应用，支持复杂的查询、分析与统计操作（如统计报表、批量数据加载与卸载等），常存在大批量处理业务。表 1-2 所示为两类应用系统的主要特点比较。

表 1-2　两类应用系统的主要特点比较

比较项目	OLTP 系统	OLAP 系统
最终用户	前台业务人员、业务管理人员	企业决策层管理人员
业务数据	当前业务数据	历史业务数据、多维视角数据
事务类型	持续时间较短、并发度高、DML	持续时间较长、只读或批量 DML
业务处理要求	要求快速响应，实时性要求高	数据吞吐量大，实时性要求不高
内部访问方式	根据索引访问数据为主	常需全表所描
索引类型	以 B-tree 索引为主	较多使用 Bitmap 索引
表联接方式	以 Nested Loop 为主	较多使用 Hash Join
绑定变量	很有必要使用绑定变量	没有必要使用绑定变量
Parallel 技术	较少使用并行技术	较多使用并行技术
物化视图	较少使用 Materialized View	较多使用 Materialized View
分区 Partition	以提高访问速度为主要目的	以提高访问并发度为主要目的

值得注意的是，目前市场上很多信息系统多属于混合类型，典型的工作时段主要是面向联机业务处理，并发度高，响应时间敏感；非工作时段（八小时之外或周六周日）执行业务数据的综合与统计或必要的批处理。因此，数据库系统的性能优化需要结合用户的应用目标有针对性地实施优化策略。

第 2 章 性能度量的主要途径

2.1 性能调优的度量概述

子曰：“工欲善其事，必先利其器。”随着 DBMS 版本的升级，ORACLE 提供的工具也越来越丰富，包括一系列的管理工具和开发工具。单就性能优化方面，常被提到的工具就有 AUTO Trace、SQL Trace、TKPROF、Explain、Oradebug、SQL Tuning Advisor、Stackpack、AWR、ASH、ADDM、OEM 等，虽然这些工具不断演化、层出不穷，但其核心的功能并没有变化。操纵数据库的核心是 SQL，提高应用系统性能的核心当然也是 SQL，所以数据库系统性能优化的核心就落实在 SQL 上，SQL 的效率提高了，应用系统的性能也就上去了。通过 ORACLE 提供的工具，可以方便地了解 SQL 的执行效率（执行路径、执行计划、统计数据等），这是使用调优工具的核心。

2.2 EXPLAIN 解释 SQL

ORACLE RDBMS 执行每一条 SQL 语句，都必须经过 ORACLE 优化器的评估。所以，了解优化器如何选择路径、表如何联接及索引如何被使用，对优化 SQL 语句至关重要。利用 ORACLE 提供的 EXPLAIN 可以迅速方便地查出给定 SQL 语句中的查询数据是如何得到的，即搜索路径（通常称为 Access Path），从而选择最优的查询方式，达到最大的优化效果。

EXPLAIN 工具的主要用途在于快速了解 SQL 语句的执行过程，该工具的特点是并不会真正地去执行该 SQL 语句，而仅仅是通过优化器给出它的执行计划，这在某些场合下是非常实用的。需要注意的是，实际的执行过程可能和 EXPLAIN 的输出有所不同，特别是对于数据量发生显著改变、统计数据发生变化、索引变更等情况。

2.2.1 配置 EXPLAIN

使用该工具需要访问一个特殊的表 PLAN_TABLE，该表用来存储执行计划，为此需要执行脚本 utlxplan.sql 创建它。

```
SQL>connect / as sysdba
Connected.

SQL>@$ORACLE_HOME/rdbms/admin/utlxplan.sql
Table created.
```

```
SQL>create public synonym plan_table for plan_table;
Synonym created.

SQL>grant select ,insert, update ,delete on plan_table
  2  to public;
Grant succeeded.
```

EXPLAIN 的基本语法：

```
SQL>EXPLAIN PLAN
             [SET STATEMENT_ID = 'stmt_id']
             FOR sql_statement;
```

2.2.2　获得执行计划

EXPLAIN 指令的执行结果存储于表 PLAN_TABLE，有两种方式获得执行计划的详细信息。

1）直接运行脚本 utlxpls.sql 获得最近一次 EXPLAIN 的执行计划。

2）使用 PL/SQL 程序包 DBMS_XPLAN 获得指定 SQL 语句的执行计划。

```
SQL>EXPLAIN PLAN for
  2  select * from emp e, dept d
  3  where e.deptno = d.deptno AND e.empno = 7934;

Explained.

SQL>@$ORACLE_HOME/rdbms/admin/utlxpls.sql

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 2385808155

--------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |     1 |    58 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |         |     1 |    58 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP     |     1 |    37 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN          | PK_EMP  |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     7 |   147 |     1   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN          | PK_DEPT |     1 |       |     0   (0)| 00:00:01 |
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------


   3 - access("E"."EMPNO"=7934)
   5 - access("E"."DEPTNO"="D"."DEPTNO")

18 rows selected.
```

ORACLE 提供了一个专门的程序包 DBMS_XPLAN，用以方便地输出 PLAN_TABLE

中的内容，还可以根据 statement_id 检索历史 SQL 语句的执行计划。

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

SQL>select * from
  2  table(dbms_xplan.display('PLAN_TABLE','stmt_id'));
```

DBMS_XPLAN 程序包的两个常用函数，参数 filter_preds 是 filter predicates 的缩写，用来过滤函数的输出。

```
DBMS_XPLAN.DISPLAY(
   table_name    IN  VARCHAR2  DEFAULT    'PLAN_TABLE',
   statement_id      IN  VARCHAR2  DEFAULT    NULL,
   format        IN  VARCHAR2  DEFAULT    'TYPICAL',
   filter_preds      IN  VARCHAR2  DEFAULT    NULL);

DBMS_XPLAN.DISPLAY_CURSOR(
   sql_id        IN  VARCHAR2  DEFAULT    NULL,
   child_number      IN  NUMBER      DEFAULT     NULL,
   format        IN  VARCHAR2    DEFAULT     'TYPICAL');
```

函数 DISPLAY_CURSOR 中的参数 sql_id 和 child_number 可以通过视图 V$SQL 获得，为方便起见可在 SQL 中加入特殊注释，如下面程序段。

```
SQL>explain plan for
  2  select /* JIADP */ ename, dname
  3  from dept d join emp e USING (deptno);

Explained.

SQL>select sql_id, child_number
  2  fromV$SQLwhere sql_text LIKE '%JIADP%';

SQL_ID       CHILD_NUMBER
------------- ------------
d7h07tp86m5gj          0

SQL>select * from
  2  table(DBMS_XPLAN.DISPLAY_CURSOR('d7h07tp86m5gj',0));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------
SQL_ID  d7h07tp86m5gj, child number 0
-------------------------------------
select /* JIADP */ ename, dname from dept d join emp e USING (deptno)
……
```

2.3　语句级跟踪 AUTOTRACE

AUTOTRACE 是一项 SQL*Plus 功能开关，打开后 ORACLE 自动跟踪执行的每一条 SQL 语句，为其生成一个执行计划并且提供与该语句执行过程相关的统计数据。

使用 AUTOTRACE 的优点在于不必关心跟踪文件（Trace File），可以立刻看到 SQL 语句的执行过程及其资源消耗情况。与 EXPLAIN PLAN 不同的是，AUTOTRACE 可以分析和执行语句；而 EXPLAIN PLAN 仅分析语句。

在 SQL*Plus 中使用 AUTOTRACE 功能的用户需要能够访问 PLAN_TABLE 表（参见 2.2 节），并且需要有必要的权限。sys 用户需要运行 plustrce.sql 脚本以创建 plustrace 角色，使用 AUTOTRACE 功能的用户需要具有该角色权限。

```
SQL>set autotrace on;
SP2-0618:   Cannot find the Session Identifier.
Check PLUSTRACE role is enabled.
SP2-0611: Error enabling STATISTICS report.

SQL>@$ORACLE_HOME/SQLPLUS/ADMIN/plustrce.sql
```

通过执行 plustrce.sql 脚本创建 plustrace 角色，将 V$视图上的选择权限授予该角色，也将 plustrace 角色授予 DBA 角色。该脚本的主要内容如下：

```
drop role plustrace;
create role plustrace;
grant select on v_$sesstat to plustrace;
grant select on v_$statname to plustrace;
grant select on v_$session to plustrace;
grant plustrace to dba with admin option;

SQL>grant plustrace to scott;

Grant succeeded.

SQL>set autotrace
Usage: SET AUTOT[RACE]
{OFF | ON | TRACE[ONLY]} [EXP[LAIN]] [STAT[ISTICS]]
```

选择“AUTOTRACE”选项后，默认情况下每条 SQL 语句的执行会输出三部分内容：语句执行结果、SQL 执行计划和执行过程中的统计信息。其中，后两项内容正是“跟踪”的内容，如果仅关心跟踪内容，可以选择“set autotrace traceonly”选项，进一步通过关键字 EXPLAIN 或 STATISTICS 选择输出跟踪内容。

AUTOTRACE 跟踪输出的执行计划同 EXPLAIN 部分，下面给出一次 SQL 语句执行过程的统计信息示例。

```
SQL>set autotrace traceonly statistics;
```

```
SQL>
SQL>select * from emp e, dept d
  2  where e.deptno = d.deptno;
14 rows selected.

Statistics
----------------------------------------------------
         1  recursive calls
         0  db block gets
        24  consistent gets
         0  physical reads
         0  redo size
      1782  bytes sent via SQL*Net to client
       384  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
        14  rows processed
```

通过上面的统计信息，可以清楚地了解到 SQL 语句执行过程中的内存消耗（db block gets 和 consistent gets，单位是 db_block_size）、I/O 情况（包括磁盘 I/O 和网络 I/O）、排序情况。

2.4 会话级跟踪 SQL_TRACE

2.4.1 设置 SQL 跟踪

会话级 SQL 跟踪是通过设置实例初始化参数 SQL_TRACE 实现的，虽然该参数可以在实例级设置，但在工程上极少这样做，因为这样跟踪所有的会话，一是显著增加实例负担，二是会产生过量的跟踪文件。

启动会话级跟踪后，ORACLE 会将该会话执行的所有 SQL 语句的跟踪内容（主要指执行计划及其统计信息）输出到跟踪文件，此时的跟踪信息要比 AUTOTRACE 提供的跟踪更详细。通过此项跟踪可以得到 SQL 执行时实际的执行计划，可以得到 SQL 执行时所花时间的具体分布（如 CPU 消耗的时间、逻辑读和物理读消耗的时间等），可以得到 SQL 执行时的各种与性能相关的统计数据，如逻辑读、物理读、fetch 次数、parse 次数等。所以，该项跟踪功能不仅能够用于性能测试，还能够用于诊断正在执行的 SQL 或存储过程的性能。

虽然跟踪文件是文本性质，但直接阅读比较困难（专业性太强），通常需要借助另一工具 TKPROF 对其进行整理和转换，以增加可读性。除了 ORACLE 自带的 TKPROF 外，还有多种工具用于格式化这里的跟踪文件，如 Metalink Note 224270.1 Trace Analyzer、第三方免费工具（如 orasrp、TVD$XTAT 等）、商业化的软件 Hotsos Profiler 等。

启用跟踪当前会话：

```
SQL>alter session set sql_trace = true;
Session altered.

SQL>exec dbms_session.set_sql_trace(true);
PL/SQL procedure successfully completed.
```

跟踪指定会话：

```
SQL>exec dbms_system.
set_sql_trace_in_session(sid,serial#,true|false);
```

这里的参数 sid 和 serial#是会话标识，唯一地标识一个会话，需要通过查询视图 V$SESSION 获得。

跟踪文件的默认命名规则是<SID>_ora_<SPID>.trc，这里的 SID 是实例名，SPID 是会话对应的 Server Process ID，可通过下列查询获得：

```
SQL>select sid from v$mystat where rownum = 1;
      SID
----------
       39

SQL>select s.username, p.spid
  2  from V$SESSION s, v$process p
  3  where s.paddr = p.addr
  4  and s.sid = &sid;
Enter value for sid: 39
old   4: and s.sid = &sid
new   4: and s.sid = 39

USERNAME                       SPID
------------------------------ ------------
SYS                                740
SQL>oradebug setmypid;
Statement processed.
SQL>oradebug tracefile_name
/ORACLE/product/10.2.0/db_1/rdbms/trace/jiadp_ora_740.trc
```

2.4.2　TKPROF 格式化跟踪文件

Tkprof 是一个用于分析 ORACLE 跟踪文件并且产生一个更加清晰可读的输出结果的操作系统命令行工具。如果数据库系统出现性能问题，一个可取的方法是通过跟踪用户的会话，使用 Tkprof 工具的排序功能格式化输出，从而找出有问题的 SQL 语句。

利用 TKPROF 的格式化输出，容易找出下列语句：

1）占用过多的 CPU 资源。

2）在 Parse、Execute、Fetch 耗费很长时间。

3）较多的磁盘访问、较大的 SGA 内存访问。

4）读取大量的数据块，却返回少量记录……

在操作系统环境下，直接输入 tkprof 指令可获得指令使用的帮助信息：

```
$ tkprof
Usage: tkprof tracefile outputfile [explain= ] [table= ]
            [print= ] [insert= ] [sys= ] [sort= ]
```

这里有两个参数比较实用，一是排序参数（具体排序选项参见指令帮助），可根据各种 SQL 的时间因素进行排序，如选项 fchela，即按照 elapsed time fetching 来对分析的结果排序（需要设置初始化参数 time_statistics=true），转换输出的文件将把最消耗时间的 SQL 放在最前面显示。另外一个有用的参数就是 sys，这个参数设置为 no，可以阻止所有以 sys 用户执行的递归 SQL 被显示出来，这样可以减少转换输出文件的内容，便于进一步分析。

下面转换输出文件中典型的一段示例：

```
select * from test where object_id=:x

call     count   cpu    elapsed    disk   query   current   rows
------- ------  -----  ---------- ------- ------- --------  ------
Parse       13   0.14     0.15       0       1        0        0
Execute  10025   0.12     0.08       0       1        0        0
Fetch       38   0.01     0.01       1     117        0       18
------- ------  -----  ---------- ------- ------- --------  ------
total    10076   0.28     0.25       1     119        0       18

Misses in library cache during parse: 13
Misses in library cache during execute: 12
Optimizer mode: ALL_ROWS
Parsing user id: 83     (recursive depth: 1)

......
```

输出文件中各数据项的含义：

```
Count   = number of times OCI procedure was executed
Cpu     = cpu time in seconds executing
Elapsed = elapsed time in seconds executing
Disk        = number of physical reads of buffers from disk
query   = number of buffers gotten for consistent read
current = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
```

1）CALL：被跟踪的 SQL 语句的处理过程分成三个阶段，即解析、执行和提取数据。

2）Parse：将 SQL 语句转换成执行计划，包括语法检查及语义分析。

3）Execute：真正地由 ORACLE 来执行语句。对于 insert、update、delete 操作，该语句会修改数据；对于 select 操作，该语句就只是确定选择的记录。

4）Fetch：返回查询语句中所获得的记录，只有 select 语句会被执行。

5）COUNT：记录被 parse、execute、fetch 的次数。

6）CPU：所有的 parse、execute、fetch 所消耗的 CPU 的时间，以秒为单位。

7）ELAPSED：所有消耗在 parse、execute、fetch 的总的时间。

8）DISK：从磁盘上的数据文件中物理读取的数据块的数量。

9）QUERY：在一致读模式下，所有 parse、execute、fetch 所获得的 buffer 的数量。一致读模式的 buffer 是并发事务环境下为用户提供一个一致读的快照。

10）CURRENT：在 current 模式下所获得的 buffer 的数量。执行 insert、update、delete 操作涉及在 current 模式下获取的 buffer 数据。

11）ROWS: 所有 SQL 语句返回的记录数目，但是不包括子查询中返回的记录数目。对于 select 语句，返回记录是在 fetch 这步；对于 insert、update、delete 操作，返回记录则是在 Execute 这步。

SQL_Trace 和 TKPROF 的联合应用：首先在操作系统层面上利用 top 命令找到当前占用 CPU 资源最高的 ORACLE 进程的 PID 号，然后在数据库中根据 PID 号找到相应的 sid 号和 serial#，示例如下：

```
# top
......
PIDUSERNAME THR PR NCE  SIZE   RES  STATE  TIME  FLTS  CPUCOMMAND
17781 ORACLE  1   49  0   8.8G8.8G run    91:11  8    10.24%  ORACLE
......

SQL>select s.sid,s.serial# from V$SESSION s, v$process p
  2  where s.paddr=p.addr and p.spid='17781';
     SID    SERIAL#
---------- ----------
     1273392
```

有了 sid 和 serial#，就可以对该会话实施 SQL 跟踪，使用 PL/SQL 包 dbms_system 跟踪该会话：

```
SQL>execute
  2  DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION(127,3392,true);
PL/SQL procedure successfully completed.
```

最后通过 TKPROF 对跟踪文件进行转换输出，从而可以诊断分析该会话执行的操作。

2.5　扩展的 SQL 跟踪

为了进一步跟踪 SQL 的行为，可以利用“诊断事件”来扩展这里的 SQL Trace。诊断事件是 ORACLE 官方未公开的性能诊断手段，此类事件主要用来产生额外的数据库跟踪信息。适当利用必要的诊断事件，可以追踪 ORACLE 的内部工作机制，窥视 ORACLE 行为，帮助分析数据库故障，诊断性能问题。

每一个诊断事件都有一个事件代码，它和错误代码是统一编号的，这里要介绍的是 10046 诊断事件，它是 SQL_Trace 的功能扩展，可以提供更丰富的跟踪信息，主要体现在对绑定变量和等待事件的跟踪上。

使用 10046 事件根据需要可以设置以下四个级别：

1）Level 1——启用标准的 SQL 跟踪功能，等价于 SQL_Trace。

2）Level 4——Level 1 + 绑定变量信息（bind varialble）。

3）Level 8——Level 1 + 等待事件信息。

4）Level 12——Level 1 + Level 4 + Level 8。

类似于 sql_trace，10046 事件通常是通过 alter session set events 指令在会话级设置，当然也可以通过初始化参数 event 在实例级设置（强烈建议不要这样做，理由同 SQL_Trace）。

设置对当前会话的跟踪（这里的 forever 表示在整个会话期间该跟踪设置都有效）：

```
SQL>alter session set events '10046 trace name context forever, level 12';
Session altered.
......
SQL>alter session set events '10046 trace name context off';
Session altered.
```

设置对指定会话的跟踪，在获得会话的 sid 和 serial#标识后可以通过 PL/SQL 程序包 DBMS_MONITOR 或 DBMS_SYSTEM 进行。

```
DBMS_MONITOR.SESSION_TRACE_ENABLE(
   session_id    IN  BINARY_INTEGER DEFAULT NULL,
   serial_num    IN  BINARY_INTEGER DEFAULT NULL,
   waits         IN  BOOLEAN DEFAULT TRUE,
binds       IN  BOOLEAN DEFAULT FALSE);

DBMS_MONITOR.SESSION_TRACE_DISABLE(
   session_id        IN    BINARY_INTEGER DEFAULT NULL,
   serial_num        IN    BINARY_INTEGER DEFAULT NULL);

SQL>exec DBMS_MONITOR.SESSION_TRACE_ENABLE(
<sid>, <serial#>, waits=>true, binds=>true);
......
SQL>exec DBMS_MONITOR.SESSION_TRACE_DISABLE (<sid>, <serial#>);

DBMS_SYSTEM.SET_EV(
session_id      IN      BINARY_INTEGER DEFAULT NULL,
serial_num  IN      BINARY_INTEGER DEFAULT NULL,
event   IN      BINARY_INTEGER DEFAULT NULL,
level   IN      BINARY_INTEGER DEFAULT NULL,
name    IN      VARCHAR2 DEFAULT NULL );

SQL>exec
SYS.DBMS_SYSTEM.SET_EV (<sid>, <serial#>, 10046, 12, '') ;
......
SQL>exec SYS.DBMS_SYSTEM.SET_EV (<sid>, <serial#>, 10046, 0, '' ) ;
```

以上介绍了在性能优化过程中最常用的三种途径，这些途径定量、精致、随手可得，熟练使用这类手段可以诊断大部分的数据库性能问题，特别是局部的性能问题。数据库是一个复杂的软件系统，在性能方面，除了局部症状，还可能有全局、整体性的问题，性能问题表现在多个方面，无从下手，这时可以借助于 Statspack 报告或 AWR 报告对数据库做

一个综合性的体检（贾代平　等，2015），以寻找性能问题背后的原因。对数据库系统由整体体检走向局部诊断是性能优化过程中的基本路径。

2.6　度量的阈值与告警

为了有效地诊断 ORACLE 的性能问题，数据库内部的统计数据必须是可用的。当启用 Statistics 功能后，ORACLE 会计算多种会话级、系统级甚至是单个 SQL 语句上的累计统计量，也会跟踪在存储段和服务上的累计统计量。其中，一些累计统计量上的变化率被 ORACLE 定义为性能上的测度或度量（Metric），这类被计算的物理量被存储在 AWR（Automatic Workload Repository，参见 11.1 节）中。

例如，关于数据库内部空间使用的度量，就有 Temp Space Used、Tablespace Space Usage 和 Tablespace Bytes Space Usage 三种，它们分别对应临时表空间的使用、永久（Permernent）表空间的使用、以字节为单位的空间使用。

```
SQL>select metric_id,metric_name from v$metricname
  2  where metric_name like'%Space%';
 METRIC_ID METRIC_NAME
---------- ----------------------------------------
      2157 Temp Space Used
      9001 Tablespace Bytes Space Usage
      9000 Tablespace Space Usage
```

大多数的 ORACLE 度量可以设置阈值（Thresholds），如果这个度量的计算值在设置的阈值之下，说明此度量反映的数据库状况是正常的，一旦度量值超过事先设置的阈值，说明度量反映的数据库状况进入某种异常状态。为了反映这个“异常”的紧急程度，阈值设置分为两级，Warning 级和 Critical 级。度量阈值超过 Warning 级，说明度量的状况进入警告状态；超过 Critical 级，说明度量的状况进入严重警告状态。下面的查询结果显示当前系统对于度量 Tablespace Space Usage 的设置，其 Warning 级和 Critical 级的设置分别是 85% 和 97%，代表当表空间的使用率超过 85%时，表空间的使用状态进入警告状态，当表空间的使用率达到 97%时，表空间的使用状态进入严重警告状态。

```
SQL>select metrics_name,warning_value,critical_value
  2  from dba_thresholds
  3  where metrics_name like '%Space%';
METRICS_NAME             WARNING_VALUE   CRITICAL_VALUE
------------------------ --------------- ---------------
Tablespace Space Usage    85              97
```

当度量的计算值超过阈值时，不论是 Warning 级还是 Critical 级，ORACLE 都会生成一条告警信息，DBA 通过视图 DBA_OUTSTANDING_ALERTS 可查看其中的告警信息。在 ORACLE 系统的性能维护过程中，DBA 应该及时查看此视图，第一时间获取关于数据库的各类告警信息。DBA 可以根据系统维护和性能管理的需要，使用 DBMS_SERVER_ALERT 包来设置度量的阈值。

```
SQL>BEGIN
  2   DBMS_SERVER_ALERT.SET_THRESHOLD(
  3     metrics_id => DBMS_SERVER_ALERT.TABLESPACE_PCT_FULL,
  4     warning_operator => DBMS_SERVER_ALERT.OPERATOR_GE,
  5     warning_value => '80',
  6     critical_operator => DBMS_SERVER_ALERT.OPERATOR_GE,
  7     critical_value => '90',
  8     observation_period => 30,
  9     consecutive_occurrences => 3,
 10     instance_name => NULL,
 11     object_type => DBMS_SERVER_ALERT.OBJECT_TYPE_TABLESPACE,
 12     object_name => NULL);
 13  END;
 14  /
PL/SQL procedure successfully completed.

SQL>select metrics_name,warning_value,critical_value
  2  from dba_thresholds
  3  where metrics_name like '%Space%';
METRICS_NAME             WARNING_VALUE   CRITICAL_VALUE
------------------------- ------------- ---------------
Tablespace Space Usage    80        90
```

常见的与性能有关的度量如表 2-1 所示。

表 2-1　常见的与性能有关的度量

度量名称（Internal）	度量名称（External）	度量单位
AVG_FILE_READ_TIME	Average File Read Time	微秒/μs
AVG_USERS_WAITING	Average Number of Users Waiting on a Class of Wait Events	会话数
BUFFER_CACHE_HIT	Buffer Cache Hit/%	百分比/%
CONTENTION_WAIT_TIME	Internal Contention Wait(by time)	微秒/μs
CPU_TIME_PER_CALL	CPU time for each user call for each service	微秒/μs
CURSOR_CACHE_HIT	Cursor Cache Hit/%	百分比/%
DATABASE_CPU_TIME	Database CPU Time/%	占全部数据库时间的百分比/%
DATA_DICT_HIT	Data Dictionary Hit/%	百分比/%
DB_BLKGETS_SEC	DB Block Gets (for each second)	数据块逻辑读的次数
DB_TIME_WAITING	Percent of Database Time Spent Waiting on a Class of Wait Events	百分比/%
DBWR_CKPT_SEC	DBWR Checkpoints (for each second)	每秒执行检查点的次数
DISK_IO	Disk I/O	毫秒/ms
ELAPSED_TIME_PER_CALL	Elapsed time for each user call for each service	微秒/μs
EXECUTE_WITHOUT_PARSE	Executes Performed Without Parsing	占所有执行次数的百分比/%
HARD_PARSES_SEC	Hard Parses (for each second)	硬解析次数
LEAF_NODE_SPLITS_SEC	Leaf Node Splits (for each second)	叶子节点分裂每秒分裂的次数

续表

度量名称（Internal）	度量名称（External）	度量单位
LIBRARY_CACHE_HIT	Library Cache Hit/%	百分比/%
LOG_SWITCH_SEC	Background Checkpoints (for each second)	每秒执行检查点的次数
LONG_TABLE_SCANS_SEC	Scans on Long Tables (for each second)	每秒表扫描的次数
OPEN_CURSORS_SEC	Cumulative Open Cursors (for each second)	每秒打开的游标数
MEMORY_SORTS_PCT	Sorts in Memory/%	百分比/%
OPEN_CURSORS_CURRENT	Current Number of Cursors	打开的游标数
PGA_CACHE_HIT	PGA Cache Hit/%	百分比/%
PHYSICAL_READS_SEC	Physical Reads (for each second)	每秒物理读的次数
PHYSICAL_WRITES_SEC	Physical Writes (for each second)	每秒物理写的次数
REDO_ALLOCATION_HIT	Redo Log Allocation Hit	百分比/%
REDO_WRITES_SEC	Redo Writes (for each second)	每秒写日志的次数
RECURSIVE_CALLS_SEC	Recursive Calls (for each second)	每秒递归调用的次数
RESPONSE_TXN	Response (for each transaction)	平均每个事务的响应时间/ms
ROWS_PER_SORT	Rows Processed for each Sort	平均每次排序涉及的行数
SHARED_POOL_FREE_PCT	Shared Pool Free/%	百分比/%
SOFT_PARSE_PCT	Soft Parse/%	占所有解析次数的百分比/%
TABLESPACE_PCT_FULL	Tablespace space usage	占表空间的百分比/%
TABLESPACE_BYT_FREE	Tablespace bytes space usage	KB（空闲空间）
TOTAL_PARSES_SEC	Total Parses (for each second)	每秒解析的次数
TRANSACTION_RATE	Number of Transactions (for each second)	平均每秒收到的事务数量
USER_COMMITS_SEC	User Commits (for each second)	平均每秒收到的用户提交的数量

表 2-1 精选了部分 ORACLE 与性能密切相关的度量，通过这些度量可以监测性能相关的大多方面，这里涉及的大多概念在本书的后续章节都有描述。表中的列“度量名称（Internal）”的内容是用于内部的，主要用于 DBMS_SERVER_ALERT 包的函数调用（如设置阈值等）；列“度量名称（External）”的内容是用于对外查询的，主要用于与度量有关的数据字典视图（如 V$SYSMETRIC）中的 METRIC_NAME 字段的填充内容。

第 3 章 优化器及其决策环境

应用系统向 ORACLE 系统发送的数据处理请求如何被执行？这是由 ORACLE 的决策机构决定的，这个决策机构就是优化器（Optimizer），优化器的决策结果就是用户指令的具体执行方案，即执行计划（Execution Plan）。一般来说，由用户指令形成执行计划的决策过程与当时的数据处理环境有关，不同的数据处理环境可能形成不同的执行计划。

3.1 游标及其处理过程

游标（Cursor）是一块特定的内存区域，又称为上下文区域（Context Area），ORACLE 使用它来保存特定的 SQL 语句及其相关的元数据（Meta Data）。游标也是一块私有的 SQL 内存区域，默认情况下，分配在 PGA 中。游标名称是 ORACLE 系统内部为处理 SQL 语句而创建的句柄。

ORACLE 系统在接收到用户请求的数据处理指令后，一般来说，需要执行如下一系列的处理过程。

1）在 PGA 中创建游标（Open Cursor）。

2）检索共享池中的库缓存（Library Cache），确认是否存在此条 SQL 指令的副本。

3）解析（Parse），包含的步骤有语法检查（Syntax Check）、语义分析（Semantic Analysis）、生成解析树（Parse Tree）及执行计划。

4）绑定变量的实例化，即绑定变量赋值。

5）执行 SQL 语句。

6）提取（Fetch）数据。对于查询来说，此步形成结果集（Record Set）。

7）关闭游标（Close Cursor）。

3.2 优化器的成本核算

在 3.1 节 SQL 指令处理过程中，形成执行计划是关键的一环，执行计划中的每一步数据处理活动都是需要消耗服务器资源的，而这种资源消耗就是数据处理的代价或成本，在执行计划里用 COST 来描述，它是一种数值型的标识，用来反映资源消耗的程度。

用户使用的 SQL 是一种非过程化的语言，这给开发人员和用户都带来极大方便。因为只需要告诉系统“做什么”，至于“怎么做”，就要交给 ORACLE 系统，而实现由“做什么”到“怎么做”的转换，就是优化器的任务，它是由一系列复杂算法来决定如何处理 SQL 语句的软件组件，它的输出就是“怎么做”的执行方案，即执行计划。基于代

价的优化器（Cost-based Optimizer，CBO）就是尝试根据 COST 的大小来选择 SQL 语句的执行方案。

3.2.1　ORACLE 成本估算模型

虽然在执行用户指令的过程中有各种不同的资源消耗，但为了评估数据处理活动优劣，这些不同类型的资源消耗都需要被统一量化、累计，这样才能有一个统一的标准去衡量不同的数据处理活动。

在 ORACLE 系统中，用户指令的执行成本被抽象归纳为两大类，消耗 CPU 的成本和产生 I/O 的成本，即存在如下公式：

$$COST = CPU\_COST + IO\_COST$$

这里的 CPU_COST 可以理解为指令执行过程中消耗的 CPU 周期数（CPU Cycles）、执行的 CPU 底层指令的数量等，而 IO_COST 可以理解为指令执行过程中需要扫描的数据块数、产生数据读请求的数量等，当然这里产生的数据 I/O 包括逻辑的和物理的。显然，这两者不是决然分开的，在一定条件下，CPU_COST 和 IO_COST 是可以相互转换的，因此从应用的角度，需要关心的是 COST 消耗及其两类代价所占的比例。在相同的执行环境下，优化器估算出的 COST 可以看作系统执行用户指令所消耗的时间指标。图 3-1 所示为优化器成本核算的主要输入信息。

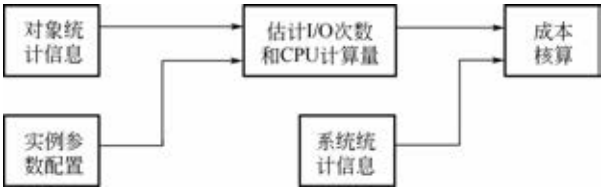

图 3-1　优化器成本核算的主要输入信息

为了估算出一种执行方案的成本，优化器需要考虑非常多的因素，试举例如下：

1）要访问的每个表的记录数。

2）对于每个表估计表中记录数的平均长度。

3）表的缓存信息及要访问的数据块数量。

4）估计可能需要的物理 I/O 的次数。

5）计算记录排序、散列链接等操作涉及的工作量。

6）相关索引的深度、索引键值数量、聚簇因子（Clustering Factor）。

7）表中行链接和行迁移的数量。

概括地说，优化器的成本核算依赖于以下三个主要方面的信息输入，优化器最终给出的是这些综合因素所决定的加权成本。

1）对象统计信息。关于操作对象（表、索引、分区等）的统计信息，包括数据方面的统计、段存储方面的统计、字段的直方图（Column Histogram）等。

2）实例参数配置。部分数据库实例的初始化参数会影响成本核算，如单个 I/O 操作能够读取的数据块数、排序操作能够使用的内存空间、并行处理的设置、优化器默认行为的配置等。

3）系统统计信息。这里的统计信息主要涉及优化器所处的操作系统环境所能提供的能

力，包括 CPU 的计算能力、I/O 速度和带宽、磁盘性能等。

3.2.2 执行计划中的相关概念

生成执行计划、解读执行计划，是优化 SQL 执行过程的基本手段，前提是需要理解执行计划中涉及的主要概念。

1）Rowid（行地址）：记录的物理存储地址。ORACLE 中的每个表都有一个 rowid 的伪列（Pseudo Column），可以像使用其他列那样使用它，但是不能对该列应用 DDL，也不能对该列执行 DML 操作。一旦一行数据插入数据库，则 rowid 在该行的生命周期内是唯一的，即使该行产生行迁移，行的 rowid 也不会改变。

2）Recursive Call（递归调用）：在执行用户的 SQL 语句时，ORACLE 必须执行一些额外的操作（如访问数据字典中的 Metadata），这些额外的操作被称为 Recursive calls 或 Recursive SQL statements，例如，当一个 DDL 语句发出后，ORACLE 总是隐含地发出一些 recursive SQL 语句，来修改数据字典信息，以便用户可以成功地执行该 DDL 语句。当需要的数据字典信息没有在共享内存中时，经常会发生 Recursive calls，这些 Recursive calls 会将数据字典信息从硬盘读入内存。

3）Row Source（行源）：用在查询中，由上一操作返回的符合条件的记录的集合，既可以是表的全部行数据的集合，又可以是表的部分行数据的集合，还可以为对上两个 Row source 进行联接操作（如 Join 联接）后得到的行数据集合。

4）Cardinality（基数）：此值表示 CBO 预期从一个行源（Row Source）返回的记录数，这个行源可能是一个表、一个索引，也可能是一个子查询。在 ORACLE 早期版本的执行计划中，Cardinality 缩写成 Card，在新的执行计划中，Card 值被 rows 替换。Cardinality 的值对于 CBO 做出正确的执行计划来说至关重要。如果 CBO 获得的 Cardinality 值不够准确（通常由没有做分析或者分析数据过旧造成），在执行计划成本计算上就会出现偏差，从而导致 CBO 输出错误（非最优）的执行计划。

5）Predicate（谓词）：在查询中的 Where 字句中使用的限制条件。

6）Driving Table（驱动表）：这个概念用于嵌套与 Hash 联接中，该表又称为外层表（Outer Table）。一般来说，当多个表（或多个行源）之间执行联接操作时，返回较少记录的表作为驱动表。所以，如果一个大表在 Where 条件有严格的限制条件（如等值限制），则该大表作为驱动表也是合适的。所以，并不是只有较小的表可以作为驱动表，正确说法应该为应用查询的限制条件后，返回较少行源的表作为驱动表。如果驱动表所代表的 Row source 返回较多的行数据，则会加重所有后续操作的负担。注意，此处虽然翻译为驱动表，但实际上翻译为驱动行源（Driving Row Source）更为确切。

7）Probed Table（被探查表）：该表又称为内层表（Inner Table）。从驱动表中得到具体一行的数据后，在该表中寻找符合联接条件的行。所以，该表应当为大表（实际上应该为返回较大 row source 的表），且在联接的字段上应该有索引。

8）Concatenated Index（组合索引）：由多个列构成的索引，如 Create index idx_emp on emp（col1，col2，col3，…），则称 idx_emp 索引为组合索引。在组合索引中有一个重要的概念——引导列（Leading Column），在上面的例子中，col1 列为引导列。当进行查询时，可以使用“where col1=？”，也可以使用“where col1=？and col2=？”，这样的限制条件都会使用索引，但是“where col2=？”的条件查询一般就不会使用该索引。

9）Selectivity（选择性）：反映过滤条件对记录的筛选程度，指应用谓词条件后符合条件的记录数占总记录数的一个比率，比率越小，选择性越强。对于字段来说，通过比较字段中唯一键的数量和表中的行数，就可以判断该字段的选择性。如果该字段中的唯一键的数量和表中的行数的比值越接近 1，则该字段的选择性越高。在选择性高的字段上执行条件查询时，比较适合使用索引。

3.3　数据访问的路径

ORACLE 中存在三种不同的访问路径可以获取用户需要的数据：一是直接扫描表的数据存储来获得数据；二是无需访问表而直接访问索引获得数据；三是先访问索引获得 Rowid，再通过 Rowid 访问表来获得数据。

3.3.1　表的访问方法

（1）全表扫描

顾名思义，执行全表扫描（Full Table Scans，FTS）时，ORACLE 读取表中所有的行，并检查每一行是否满足检索条件来过滤记录。

为了提高全表扫描操作的效率，ORACLE 使用多块读（Multi-block）方法获取数据，即在一次 I/O 操作中能够读取多个数据块的数据，至于能够读取的数据块的最大数量，由 db_block_multiblock_read_count 初始化参数确定，这极大地减少了 I/O 总次数，提高了系统的吞吐量。一般来说，利用多块读的方法可以十分高效地实现全表扫描。

这里要注意的是，全表扫描要消耗的资源取决于 ORACLE 需要扫描的数据块数（而不是记录数），这就涉及一个存储的概念——高水位线（High Water Mark，HWM），即全表扫描时 ORACLE 要读取高水位线下的所有数据块。因此，在实践中经常会看到，对一个小表进行全表扫描，却需要读取大量的数据块，其原因就在于这个小表曾经“大”过。关于优化物理存储的内容将在 5.6 节里介绍。

```
SQL>explain plan for select * from emp;
Execution Plan
----------------------------------------------------------
Plan hash value: 3956160932
--------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | select STATEMENT  |      |    14 |  1218 |    3   (0)|
|   1 |  TABLE ACCESS FULL| EMP  |    14 |  1218 |    3   (0)|
--------------------------------------------------------------------
```

（2）通过 ROWID 的表存取（Table Access by ROWID 或 rowid lookup）

行的 ROWID 指出了该行所在的数据文件、数据块及行在该块中的位置，所以通过 ROWID 来存取数据可以快速定位到目标数据，这是 ORACLE 访问单行数据的最快方法。

这种存取方法不会用到多块读操作，一次 I/O 只能读取一个数据块。执行计划中可以看到该存取方法，如访问索引后的后续步骤往往是 TABLE ACCESS BY ROWID，这就是

通过 ROWID 直接访问表中的数据。

ROWID 里包含记录存储的丰富信息，如数据库对象的编号、所在文件编号、数据块编号、记录在数据块中的编号等，ORACLE 专门提供了一个 PL/SQL 包 DBMS_ROWID 来处理 ROWID，需要时可以使用。

使用 ROWID 存取的方法：

```
SQL>explain plan forselect * from emp
where rowid='AAAMqdAAEAAAACXAAI';
Execution Plan
----------------------------------------------------------
Plan hash value: 1116584662
-------------------------------------------------------------
| Id  | Operation                   | Name | Rows  | Bytes |
-------------------------------------------------------------
|   0 | select STATEMENT            |      |    1 |   99 |
|   1 |  TABLE ACCESS BY USER ROWID | EMP  |    1 |   99 |
-------------------------------------------------------------
```

3.3.2 索引的访问方法

索引扫描（Index Scan）是通过 Index 查找到记录对应的 Rowid 值（对于非唯一索引可能返回多个 Rowid 值），然后根据 Rowid 直接从表中得到具体的数据，这种查找方式称为索引扫描或索引查找（Index Lookup）。一个 Rowid 唯一地标识一条记录，该记录所在的数据块是通过一次 I/O 得到的，在此情况下该次 I/O 只会读取一个数据库块。

在索引中，除了存储被索引字段的值外，索引还存储了记录对应的 Rowid 值。通常索引扫描由两个步骤组成：

1）扫描索引得到对应的 Rowid 值。

2）通过找到的 Rowid 从表中读出具体的数据。

每步都是单独的一次 I/O，但是对于索引，由于经常使用，绝大多数都已经缓存 CACHE 到内存中，所以第一步的 I/O 经常是逻辑 I/O，即数据可以从内存中得到。但是对于第二步来说，如果表比较大，则其数据不可能全在内存中，所以其 I/O 很有可能是物理 I/O，相对于逻辑 I/O 来说，这类操作要低效得多。

注意，下面两个查询对应的执行计划，区别在于前者仅通过索引就可直接获得数据，而后者在访问索引后需要根据获得的 Rowid 信息再去访问表中的数据。

执行计划一：

```
SQL>explain plan forselect empno from emp where empno=7839;
Execution Plan
----------------------------------------------------------
Plan hash value: 56244932
--------------------------------------------------------------------
| Id  | Operation        | Name   | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | select STATEMENT |        |    1 |   13 |    1   (0)|
```

```
|*  1 |  INDEX UNIQUE SCAN| PK_EMP   |    1 |   13 |    1   (0)|
---------------------------------------------------------------
```

执行计划二：

```
SQL>explain plan forselect empno,ename from emp where empno=7839;
Execution Plan
----------------------------------------------------------
Plan hash value: 2949544139
-------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes |
-------------------------------------------------------------------------
|   0 | select STATEMENT             |               |    1 |  20    |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP            |    1 |  20    |
|* 2 |   INDEX UNIQUE SCAN          | PK_EMP         |    1 |        |
-------------------------------------------------------------------------
```

在上面的例子中已经见到了索引唯一性扫描（INDEX UNIQUE SCAN），这是针对唯一性索引（Unique Index）的索引访问。根据被索引字段的等值条件查询，索引唯一性扫描可以快速获得对应记录的 Rowid 信息。

下面介绍另外几种常用的索引访问方法。

（1）索引范围扫描

索引里包含被索引字段的数据，而且与被索引字段在表中的数据不同，被索引字段在索引里的数据是经过排序的，因此，利用这一特性根据取值范围的条件查询数据就很容易。使用一个索引存取多行数据，在索引上使用索引范围扫描（Index Range Scan）的典型情况是在谓词（where 限制条件）中使用了范围操作符（如>、<、<>、>=、<=、between）。

```
SQL>explain plan forselect empno from emp where empno>7839;
Execution Plan
----------------------------------------------------------
Plan hash value: 1567865628
-----------------------------------------------------------------
| Id  | Operation       | Name   | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------
|   0 | select STATEMENT |       |      5 |   65 |    2   (0)|
|*  1 |  INDEX RANGE SCAN| PK_EMP |      5 |   65 |    2   (0)|
-----------------------------------------------------------------
```

在非唯一索引上，等值条件的查询可能返回多行记录，所以在非唯一索引上都是使用索引范围扫描。使用 Index Rang Scan 的三种情况：

1）在唯一索引列上使用了 range 操作符（>、<、<>、>=、<=、between）。

2）在组合索引上，只使用部分列进行查询，导致查询出多行。

3）对非唯一索引列上进行的任何查询。

（2）索引全扫描

在逻辑的使用上，可以把索引比作一个特殊的、具有两个字段的表，这两个字段分别是被索引字段、记录物理地址 Rowid，并且与普通的堆表（Heap Table）不同的是，索引中

的数据是根据被索引字段排序的。因此，才有索引范围扫描。

这里的索引全扫描（Index Full Scan），相当于把索引看作“索引表”，对“索引表”的全表扫描（但读取的方式是单块读），此时 ORACLE 会访问索引树的所有叶子（Leaf）节点，从叶子层的一端到叶子层的另外一端，这种访问带来的一种自然好处是获得的数据是有序的。

```
SQL>select empno from emp order by empno;
Execution Plan
----------------------------------------------------------
Plan hash value: 179099197
-------------------------------------------------------------------
| Id  | Operation        | Name   | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | select STATEMENT |          |    14 |   182 |    2   (0)|
|   1 |  INDEX FULL SCAN | PK_EMP   |    14 |   182 |    2   (0)|
-------------------------------------------------------------------
```

这里的索引全扫描操作，ORACLE 并不需要访问索引树的所有分支节点，此时 ORACLE 只是从根节点访问有限的分支节点定位到最左端（或最右端）的叶子节点，再通过叶子节点之间的双向链接，即可访问所有的叶子节点。值得注意的是，正是这种访问方式决定了索引全扫描不能使用并行处理。

（3）索引快速全扫描

与上面的索引全扫描相比，这里的索引快速全扫描（Index Fast Full Scan）多了“Fast”，主要区别是在读取索引块的方式上。与 Index Full Scan 类似，这种索引访问方式不仅需要扫描索引中的所有叶子节点，同时也扫描所有的分支节点，这是由其采用多块读的方式决定的。另外，多块读的方式也决定了这种扫描方式获得的数据具有随机性，即数据不确定以排序顺序被返回。在这种索引扫描方式中，由于使用多块读功能，所以也可以使用并行读入，以便获得最大吞吐量与缩短执行时间。

```
SQL>explain plan for select empno from emp;
Execution Plan
----------------------------------------------------------
Plan hash value: 366039554
-------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | select STATEMENT    |       |    14 |   182 |    2   (0)|
|   1 |  INDEX FAST FULL SCAN| PK_EMP | 14 |   182 |    2   (0)|
-------------------------------------------------------------------
```

（4）索引跃式扫描

索引跃式扫描（Index Skip Scan）是针对复合 B 树索引的扫描，它充分发挥了复合索引的价值。一般来说，针对复合索引中的所有索引字段的复合条件查询，或针对前导字段的条件查询 ORACLE 可以利用对应的复合索引，Index Skip Scan 扩展了复合索引的使用环境。

如果复合索引的前导索引字段的 Cardinality 相对较小，当根据非前导字段的条件查询时，ORACLE 试图对前导字段做遍历处理，将查询条件分解为多个针对所有被索引字段查询的多个查询条件的联合，这样优化器就可以正常地利用复合索引。故这里的 Skip 是 ORACLE 针对复合索引的前导字段的特殊处理。

```
SQL>create table employee(ename varchar2(10),gender char(1));
SQL>create index idx_ge on employee(gender,ename);

begin
for i in 1..10000 loop
insert into employee
values(dbms_random.string('a',10),
decode(round(dbms_random.value(0,1)),0,'F',1,'M'));
end loop;
Commit;
end;
/
SQL>exec
dbms_stats.gather_table_stats('SCOTT','EMPLOYEE',
cascade=>true);

SQL>select /*+ index(e,idx_ge) */ * from employee e
  2  where ename='IpBOdkCIMj';
Execution Plan
----------------------------------------------------------
Plan hash value: 2937972234
-------------------------------------------------------------------
| Id  | Operation        | Name   | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | select STATEMENT |        |     1 |    13 |     3   (0)|
|*  1 |  INDEX SKIP SCAN | IDX_GE |     1 |    13 |     3   (0)|
-------------------------------------------------------------------
```

在上面的这个例子中，gender 字段只有两个值'F'、'M'，Index Skip Scan 的意思是 ORACLE 将查询条件 ename='IpBOdkCIMj'分解为：

```
(gender='F') and (ename='IpBOdkCIMj') or
(gender='M') and (ename='IpBOdkCIMj')
```

3.4　行源的联接关系

这里的行源（Row Source）是指包括表、视图、子查询、中间查询在内的任意查询结果集（Result Set），在优化器形成执行计划的过程中，除了要考虑数据的访问路径之外，还要考虑两个或多个结果集之间的联接关系。

如果存在多个结果集需要关联，优化器总是将其转换为两两关联，在这个过程中需要确定两个方面的内容：一是联接顺序；二是联接方法。一般来说，要优先处理选择性强的过滤条件，使后续的结果集尽可能地包含更少的记录。至于联接方法，选择的依据取决于两个结果集的特征及其关联的条件。

3.4.1 内联接和外联接

（1）内联接

内联接（Inner Join）是典型的联接运算，使用等号或不等号之类的比较运算符，包括等值联接和自然联接。内联接使用比较运算符根据联接字段值匹配两个行源中的记录。例如，根据部门编号检索 employees 和 departments 表中所有 department_id 相同的行。

内联接是使用较普遍的行源联接方式，标准 SQL 中使用 Inner Join、Join、Natural Join、Join Using…描述的联接均为内联接。

（2）外联接

外联接（Outer Join）是对传统内联接的一种扩展，该类联接除返回内联接的结果外，还额外包含驱动行源中所有不满足联接条件的记录。

外联结可以是左外联接、右外联接或全外联接。指定外联接时，可以由下列几组关键字中的一组指定。

1）LEFT JOIN 或 LEFT OUTER JOIN（左向外联接）：该联接的结果集包括 LEFT OUTER 子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值。

2）RIGHT JOIN 或 RIGHT OUTER JOIN（右向外联接）：左向外联接的反向联接，将返回右表的所有行。如果右表的某行在左表中没有匹配行，则将为左表返回空值。

3）FULL JOIN 或 FULL OUTER JOIN（完整外部联接）：返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

（3）交叉联接

交叉联接（Cross Join）返回两个行源中任意两条记录的所有行组合，当在两个联接行源中不设置任何关联条件时，返回此结果集。交叉联接也称作笛卡儿积（Cartesian Product）。

```
SQL>explain plan for select * from emp,dept;
Execution Plan
----------------------------------------------------------
Plan hash value: 2034389985
-----------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------------
|   0 | select STATEMENT        |      |    56 |  6720 |  10  (0)|
|   1 |  MERGE JOIN CARTESIAN   |      |    56 |  6720 |  10  (0)|
|   2 |   TABLE ACCESS FULL     | DEPT |     4 |   120 |   3  (0)|
|   3 |   BUFFER SORT           |      |    14 |  1260 |   7  (0)|
|   4 |    TABLE ACCESS FULL    | EMP  |    14 |  1260 |   2  (0)|
-----------------------------------------------------------------------
```

上面的执行计划中显示，步骤 2 和步骤 3 对应的两个行源之间实施笛卡儿积操作（MERGE JOIN CARTESIAN），显然笛卡儿积的操作属于融合联接（参见 4.3.3 节）大类中的一种。

（4）半联接和反联接

当两个行源之间执行联接时，只需返回其中一个行源的数据行，而另外一个行源是用来实施过滤标准的，这类情形优化器可以使用半联接或反联接（Semi/Anti Join）。返回行源 1 的数据，前提是在行源 2 中找到匹配的行，此为半联接。返回行源 1 的数据，前提是在行源 2 中不存在匹配的行，此为反联接。反联接则是一种特殊的半联接。

在执行计划中，半联接和反联接作为其他常规联接的一种操作选项出现。行源 1 中的数据行是否出现在结果集中需要根据行源 2 中是否出现至少一个相匹配的数据行来判断，这种情况就会发生半联接；而反联接是半联结的补集。

半联接多用于 In、Exists 等子查询，对于行源 1，查找行源 2（子查询），匹配第一行之后就返回，不再往下查找。

```
SQL>explain plan for select D.* from dept D
  2  where D.deptno in (select deptno from emp E);
Execution Plan
----------------------------------------------------------
Plan hash value: 1754319153
-------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | select STATEMENT   |      |      3   |   129 |       7   (15)|
|*  1 |  HASH JOIN SEMI    |      |      3   |   129 |       7   (15)|
|   2 |   TABLE ACCESS FULL| DEPT |      4   |   120 |       3    (0)|
|   3 |   TABLE ACCESS FULL| EMP  |     14   |   182 |       3    (0)|
-------------------------------------------------------------------------
```

隐含参数_always_semi_join 的设置值决定优化器是否启用半联接及如果启用默认采用哪种联接类型（上例中为散列联接）。该参数的取值有 CHOOSE、HASH、MERGE、NESTED_LOOPS、OFF，当设置为 CHOOSE 时，默认的联接类型由优化器确定，当设置为 OFF 时，禁用半联接。

在使用和禁用半联接的操作中，前者往往带来性能上的优势。以 NESTED LOOPS 为例，在 NESTED LOOPS 联接中，驱动表被读取后需要逐个地进入内层循环来进行匹配工作，并且只有当外层循环的数据行与内层循环中的每一行数据匹配运算完成后才会结束一个结果集的获取；相对而言，半联接的优势在于行源 1 中的每一条记录只返回一次，而不论行源 2 中有几条匹配的记录，因此，半联接会在找到行源 2 中匹配到的第一条数据后立即结束处理。

反联接和半联接相反，只有行源 1 中的行仅在行源 2 中不能匹配的时候才返回, 多用于!=、Not in 等子查询。

```
SQL>explain plan for select D.* from dept D
2  where not exists (select 1 from emp E
3  where E.deptno = D.deptno);
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 474461924
-------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | select STATEMENT   |      |      4 |     172 |    7  (15)|
|*  1 |  HASH JOIN ANTI    |      |      4 |     172 |    7  (15)|
|   2 |   TABLE ACCESS FULL| DEPT |      4 |     120 |    3   (0)|
|   3 |   TABLE ACCESS FULL| EMP  |      14 |    182 |    3   (0)|
-------------------------------------------------------------------
```

3.4.2 嵌套循环联接

嵌套循环联接（Nested Loops Join）是将联接的两个行源分别作为外循环和内循环处理，并在内循环中根据谓词条件返回符合条件的数据行的联接方法。其中，用作外循环的行源称为驱动行源（或驱动表），用作内循环的行源称为被探查行源（Probed Table 或被驱动表）。

```
SQL>explain plan for select /*+ ordered use_nl(d,e) */ e.*,d.*
  2  from dept d,emp ewhere e.deptno=d.deptno;
Execution Plan
----------------------------------------------------------
Plan hash value: 4192419542
-------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | select STATEMENT   |         |   14  |  840  |   10   (0)|
|   1 |  NESTED LOOPS      |         |   14  |  840  |   10   (0)|
|   2 |   TABLE ACCESS FULL| DEPT    |    4  |   80  |    3   (0)|
|*  3 |   TABLE ACCESS FULL| EMP     |   14  |  160  |    2   (0)|
-------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("E"."DEPTNO"="D"."DEPTNO")
```

在上面例子演示的嵌套循环中，dept 是驱动表，emp 是被驱动表。一般来说，通过下面两种方法可以优化嵌套循环联接的效率：①选择记录数较少的行源作为外循环；②在内循环的联接字段建立索引。如上例所示，选择记录数较少的 dept 作为驱动表是明智的选择，如果能够在被驱动表的联接字段 deptno 上建立索引，那么在处理内循环时就可避免在 emp 表上的全表扫描，联接效率会大大提高。

嵌套循环在联接过程中能够快速输出符合条件的第一条（第 n 条）记录，这在应用中是非常好的特性。

3.4.3 排序融合联接

排序融合联接（Sort Merge Join）是先将联接的两个行源根据联接字段执行排序（Sort），再将两个排好序的行源根据联接字段进行融合（Merge），从而找出符合条件的记录。这两

个过程决定了排序融合联接的特点：

1）如果原始的两个行源的数据都是无序的，一般情况下应用这种联接方法往往不是最优的，其原因是联接过程中要分别对两个行源执行排序操作，这在 OLTP 系统中是典型的消耗资源的操作。Sort 操作要消耗额外的 CPU 和内存资源，CPU 资源主要消耗在算法排序和结果集合整合上，而内存资源的消耗更加需要关注，排序操作要在专门的 PGA 排序区内完成。如果 PGA 中指定的排序大小（sort_area_size）不足以支持排序操作，ORACLE 需要调用临时表空间的容量来进行磁盘操作。

2）此类联接不存在驱动行源和被驱动行源的问题，两边的数据行源没有顺序区别，都要进行排序操作；由于是经过排序后执行两个行源的融合，它不仅适用于等值连接，同样还适用于不等联接，如大于（>）、小于（<）、大于等于（>=）、小于等于（<=）等。

3）排序融合联接的优势是可以在一定程度上减少随机读的情况，它的最大特征是在一次扫描的同时，就可判断行源之间的联接关系。不会像 Nest Loops Join 那样频繁地进行数据读取。使用这种方式的前提，就是联接的两个行源必须按照关联字段的顺序进行排序。如果两个需要联接的行源事先就是已排序的（如利用已有索引），执行此类联接是优先考虑的选项。

```
SQL>explain plan forselect /*+ use_merge(d,e) */ e.*,d.*
  2  from dept d,emp e where e.deptno=d.deptno;
Execution Plan
----------------------------------------------------------
Plan hash value: 844388907
-------------------------------------------------------------------
| Id  | Operation                  | Name    | Rows  | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | select STATEMENT          |       |   14  |   6  (17)|
|   1 | MERGE JOIN                |       |   14  |   6  (17)|
|   2 | TABLE ACCESS BY INDEX ROWID| DEPT | 4     |   2   (0)|
|   3 | INDEX FULL SCAN           | PK_DEPT | 4     |   1   (0)|
|* 4 | SORT JOIN                 |       |   14  |   4  (25)|
|   5 | TABLE ACCESS FULL         | EMP   |   14  |   3   (0)|
-------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("E"."DEPTNO"="D"."DEPTNO")
       filter("E"."DEPTNO"="D"."DEPTNO")
```

3.4.4　散列联接

散列联接（Hash Join）是将两个行源依据散列运算来确定对应数据行之间的联接关系，可以这样理解：ORACLE 通过对关联字段实施散列函数的运算来确定对应数据行之间的关联。对于排序融合联接，如果两个待联接的行源其结果集很大而且需要排序，则排序融合联接的执行效率一定不高；而对于嵌套循环联接，如果驱动行源所对应的驱动结果集的记录数很大，即便在被驱动表的联接列上存在索引，此时使用嵌套循环联接的执行效率也会同样存在问题。散列联接可避免上述问题。

根据散列运算的特点，可以确定散列联接具有以下特点：

1）散列联接对行源没有排序的要求，但它只能用于等值联接的条件，即使是散列反联接，ORACLE 实际上也是将其换成等值联接。

2）散列联接很适用大小行源之间的联接，且联接后产生的结果集的记录数较多的情形，特别是在小行源的关联字段选择性非常好的情况下，散列联接的执行代价与单独扫描大行源几乎相当。

3）和排序一样，散列操作需要在 PGA 的内存空间（hash_area_size）完成。散列操作形成的散列表（Hash Table）的大小与行源大小和关联字段有关，如果散列表能够完全存在于内存中，散列联接操作会获得很高的效率。

ORACLE 优化器在解析目标 SQL 的时候是否考虑散列联接受限于隐含参数_HASH_JOIN_ENABLED，默认值是 TRUE。

```
SQL>explain plan forselect /*+ use_hash(e,d) */ e.*,d.*
  2  from emp e,dept d where e.deptno=d.deptno;
Execution Plan
----------------------------------------------------------
Plan hash value: 615168685
--------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | select STATEMENT   |      |    14 |   840 |    7  (15)|
|*  1 |  HASH JOIN         |      |    14 |   840 |    7  (15)|
|   2 |   TABLE ACCESS FULL| DEPT |     4 |    80 |    3   (0)|
|   3 |   TABLE ACCESS FULL| EMP  |    14 |   560 |    3   (0)|
--------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("E"."DEPTNO"="D"."DEPTNO")
```

3.4.5 星形转换

星形转换（Star Transformation）又称星形模式，它包括一个大的用来存储详细业务数据的事实表（Fact Table），表中存在一系列外键，这些外键关联到一个相对较小的更静态的维度表（Dimenstion Table）。在维度表中记录了用来描述事实表中业务数据的一些分类，如客户、时间、产品等信息。星形模式的一种特殊形式称为雪花模式。雪花模式是指维度表本身也包括外键，用以关联其他更高一级的维度表或是其他数据表。在 OLAP 系统中，星形模式是一个能够实现行源之间高效联接的设计模式，能够合理地满足数据仓库中的数据处理需求。

在 ORACLE 的案例中，SH 模式下有一套关于销售记录的表，如图 3-2 所示，SALES 是个事实表，它记录了销售量（QUANTITY_SOLD）、销售金额（AMOUNT_SOLD）等，同时记录还关联了各个时间段、产品、客户、销售渠道和促销等信息。通过关联时间维度表（TIMES）、产品维度表（PRODUCTS）、客户维度表（CUSTOMERS）、渠道维度表

（CHANNELS）和促销活动维度表（PROMOTIONS），可以得到事实表中每个维度的详细描述。

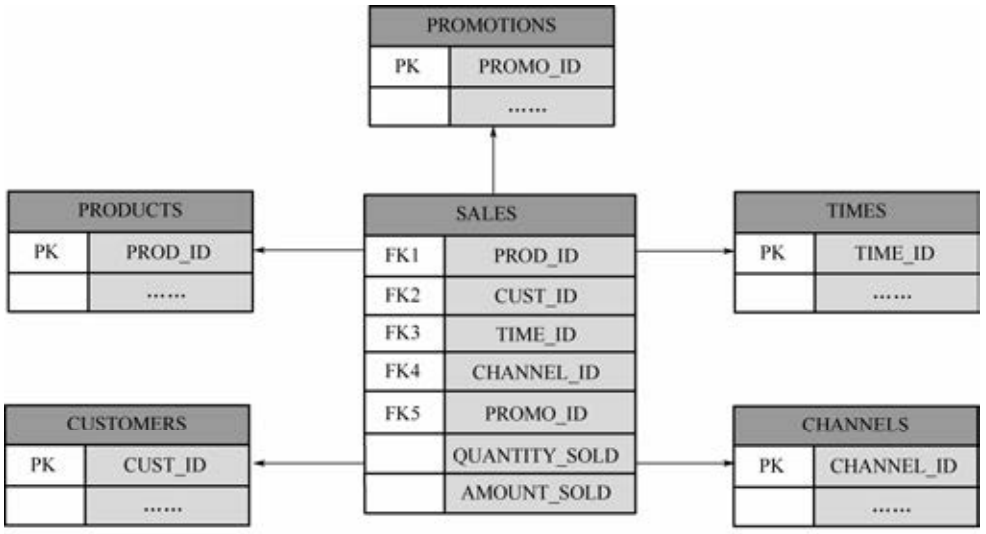

图 3-2　星形模式示例

典型的事实表针对每一个维度表会存在一个外键（Foreign Key），指向对应的维度表（主码）。事实表与维度表间的联接操作一般都会发生在事实表上的外键和与之对应的维度表的主联间。在上述模式的典型查询中，事实表总是会和多个与之相比小得多的维度表存在联接(Join)操作，同时也会在维度表的其他字段上存在严格的过滤谓词，类似下面的查询：

```
SQL>explain plan forselect quantity_sold, amount_sold
  2   from sales s JOIN products p USING (prod_id)
  3       JOIN times USING (time_id)
  4       JOIN customers c USING (cust_id)
  5  where week_ending_day = to_date('2008-11-29','YYYY-MM-DD')
  6    AND prod_name = 'CD-ROM Diskette'
  7    AND cust_first_name = 'William'
  8    AND cust_last_name = 'Smith'
  9    and cust_year_of_birth=1966;

Execution Plan
----------------------------------------------------------
Plan hash value: 3972590324
-----------------------------------------------------------
| Id  | Operation                    | Name     |
-----------------------------------------------------------
|   0 | select STATEMENT             |          |
|   1 |  NESTED LOOPS                |          |
|   2 |   NESTED LOOPS               |          |
|   3 |    NESTED LOOPS              |          |
|   4 |     NESTED LOOPS             |          |
```

```
|   5 |     TABLE ACCESS FULL            | SALES            |
|*  6 |     TABLE ACCESS BY INDEX ROWID| PRODUCTS           |
|*  7 |      INDEX UNIQUE SCAN           | PRODUCTS_PK       |
|*  8 |    TABLE ACCESS BY INDEX ROWID  | CUSTOMERS         |
|*  9 |     INDEX UNIQUE SCAN            | CUSTOMERS_PK      |
|* 10 |   INDEX UNIQUE SCAN              | TIMES_PK          |
|* 11 |  TABLE ACCESS BY INDEX ROWID    | TIMES             |
-------------------------------------------------------------
```

在上述演示的执行计划中，显然在步骤 5 中对事实表执行了全表扫描。

星形转换的目的是尽量避免直接去扫描星形模式中的事实表，因为这些事实表往往存在海量数据，对这些表的全表扫描会引起大量物理 I/O，这在 OLTP 和 OLAP 系统中都是应该极力避免的。星形转换的思路是将原星形联接查询中的针对各个维度表的谓词条件通过查询重写的方式转换为子查询应用到事实表上。

针对上面的查询，ORACLE 实施星形转换，改写的查询可能类似如下：

```
select quantity_sold, amount_sold
from sales s
where s.prod_id IN
(select prod_idfrom products
where prod_name = 'CD-ROM Diskette')
AND s.time_id IN
(select time_idfrom times
where week_ending_day = to_date('2008-11-29','YYYY-MM-DD'))
AND s.cust_id IN
(select cust_idfrom customers
where cust_first_name = 'William'
AND cust_last_name = 'Smith'
AND cust_year_of_birth=1966);

SQL>explain plan forselect
  2  /*+ OPT_PARAM('star_transformation_enabled' 'true') */
  3  quantity_sold, amount_sold
  4  from sales s JOIN products p USING (prod_id)
  5        JOIN times USING (time_id)
  6        JOIN customers c USING (cust_id)
  7  where week_ending_day = to_date('2008-11-29','YYYY-MM-DD')
  8     AND prod_name = 'CD-ROM Diskette'
  9     AND cust_first_name = 'William'
 10     AND cust_last_name = 'Smith'
 11     and cust_year_of_birth=1966;

Execution Plan
----------------------------------------------------------
Plan hash value: 3428561791
-------------------------------------------------------------------
| Id  | Operation                     | Name                      |
```

```
-------------------------------------------------------------
|   0 | select STATEMENT                |                       |
|   1 |  NESTED LOOPS                   |                       |
|   2 |   NESTED LOOPS                  |                       |
|   3 |    MERGE JOIN CARTESIAN         |                       |
|   4 |     MERGE JOIN CARTESIAN        |                       |
|   5 |      TABLE ACCESS BY INDEX ROWID | CUSTOMERS             |
|*  6 |       INDEX RANGE SCAN          |   CUST_NAMEDOB_IDX    |
|   7 |      BUFFER SORT                |                       |
|   8 |       TABLE ACCESS BY INDEX ROWID| TIMES                 |
|*  9 |        INDEX RANGE SCAN         |   TIMES_WEND_IDX      |
|  10 |     BUFFER SORT                 |                       |
|  11 |      TABLE ACCESS BY INDEX ROWID | PRODUCTS              |
|* 12 |       INDEX RANGE SCAN          |   PROD_NAME_IDX       |
|* 13 |    INDEX RANGE SCAN             |   SALES_CONCAT_IDX    |
|  14 |   TABLE ACCESS BY INDEX ROWID   |   SALES               |
-------------------------------------------------------------
```

上面的实例演示了默认的 ORACLE 星形转换行为。事实上，针对此类查询，还可以充分利用位图索引对其实施人为的星形转换，如针对联接条件，创建如下位图联接索引：

```
CREATE BITMAP INDEX sales_cust_bjix
ON sales(customers.cust_first_name,customers.cust_last_name,
         customers.cust_year_of_birth)
from sales, customers
where sales.cust_id = customers.cust_id;
```

上面创建的位图索引，将 SALES 表和 CUSTOMERS 表根据查询条件进行了预先的位图联接，使它能够在查询过程中直接提取与维度表中的值相匹配的事实表记录。对应的执行计划转换如下（注意此处的 4、5、6 三个步骤）：

```
-------------------------------------------------------------
| Id  | Operation                    | Name          |
-------------------------------------------------------------
|   0 | select STATEMENT             |               |
|   1 |  NESTED LOOPS                |               |
|   2 |   NESTED LOOPS               |               |
|   3 |    NESTED LOOPS              |               |
|   4 |     TABLE ACCESS BY INDEX ROWID | SALES       |
|   5 |      BITMAP CONVERSION TO ROWIDS|             |
|*  6 |       BITMAP INDEX SINGLE VALUE | SALES_CUST_BJIX |
|*  7 |     TABLE ACCESS BY INDEX ROWID  | PRODUCTS    |
|*  8 |      INDEX RANGE SCAN            | PROD_NAME_IDX |
|*  9 |    INDEX RANGE SCAN              | TIMES_WEND_IDX |
|* 10 |   TABLE ACCESS BY INDEX ROWID    | TIMES       |
-------------------------------------------------------------
```

在这个执行计划里，步骤 BITMAP INDEX SINGLE VALUE→BITMAP CONVERSION TO ROWIDS→TABLE ACCESS BY INDEX ROWID 利用位图联接索引 sales_cust_bjix，完全避免了访问维度表 Customers，这种处理方式可以提供很好的星形联接性能。本例中其他针对维度表的查询条件也可做类似处理。

3.5 优化器的决策环境

现代 DBMS 的核心技术之一就是基于“代价”估算的决策算法——查询优化器，它将开发人员通过 SQL 表达的用户需求转换为数据服务系统的一系列可执行步骤。对绝大多数软件技术人员来说，优化器是神秘的，但它的决策是有据可循的，可以通过执行计划观测它的决策，还可以通过调整优化器的运行环境来影响它的决策。在我们看来，这个优化器的决策环境包括如下三个方面：影响优化器的主要参数、数据库对象的统计信息、系统统计信息。

3.5.1 影响优化器的主要参数

虽然 ORACLE 系统提供了多个参数来控制优化器的运行，每个参数又会有不同的设置值（包括默认值），但需要注意的是，没有一个参数的设置值适应所有的运行环境，需要根据应用系统的负荷情况和特定 SQL 的运行需求动态地调整。

（1）优化模式参数 optimizer_mode

此参数确定优化器的总体行为，即优化模式。ORACLE 提供两类优化模式，涉及对 SQL 运行的优化是以响应时间为目标还是以吞吐量为目标，前者以最短的时间返回符合条件的 Top n 条记录（n 可取 1、10、100 或 1000），后者以返回所有符合条件的记录为目标。

1）目标 1，响应时间，参数设置为 first_rows_n。

2）目标 2，吞吐量，参数设置为 all_rows。

（2）多块读参数 db_file_multiblock_read_count

在执行全表扫描或索引快速全扫描的过程中，ORACLE 执行“多块读”方式以最大化磁盘 I/O 效率。“多块读”对 ORACLE 系统的 I/O 性能有明显的影响，参数 db_file_multiblock_read_count 决定最大一次物理读能够读取的数据块（db_block_size）的数量。

默认情况下，数据库引擎对该参数的默认设置是取如下两个表达式的小者：

```
(1048576/db_block_size, db_cache_size/(sessions*db_block_size))
```

这个表达式的内在含义是，默认在大多数系统中，一次物理读能够获取 1MB 大小的数据为最佳。显然此值不能适用所有的系统，但需要注意的是，参数 db_file_multiblock_read_count 的大小设置与磁盘读的吞吐率（MB/s）成正比，而与 CPU 使用率成反比。

（3）优化器特征参数 optimizer_features_enable

每个 DBMS 版本也都对应着某种优化器的版本，随着 DBMS 版本的升级，优化器的特性也在更新，参数 optimizer_features_enable 指定优化器行为对应的版本，执行下面的查询可以了解参数的有效设置值：

```
select value from v$parameter_valid_values
where name = 'optimizer_features_enable';
```

建议将此参数设置为与当前 DBMS 软件相匹配的版本号。

（4）动态采样参数 optimizer_dynamic_samppling

对 ORACLE 而言，基于成本优化器 CBO 工作的基础是系统环境、对象统计量和成本计算。然而大多数情况下，收集统计量是一个异步单独的过程。

为了应对统计量缺失的情况，ORACLE 推出了 Dynamic Sampling（12c 之后称为 Dynamic Statistic）动态采样收集的技术策略。如果数据表等对象没有统计量存在，ORACLE 又需要统计量生成执行计划，数据库会进行一次临时性的数据收集动作。根据不同的采用比例，ORACLE 实时地采集部分数据块，应用查询条件进行小规模试算。最后根据统计规则，将结果集合放大后，作为结果集合统计量和 row source 纳入 CBO 体系里面。

参数 optimizer_dynamic_samppling 确定动态采样的比例（实际采样的数据块数），其设置值为 0～10 的整数，0 表示禁用动态采样，10 表示采样所有的数据块，对于 1～9 的设置值，实际采样的数据块数由表达式确定：

```
32*power(2, optimizer_dynamic_samppling-1)
```

显然此参数的设置值影响到优化器对数据库对象统计信息的估算精度。

（5）索引代价调节参数 optimizer_index_cost_adj

该参数设置优化器估算通过索引访问代价的一个额外比例。优化器计算通过索引扫描访问表数据的 cost 开销，可以通过这个参数进行调整。参数可用值的范围为 1～10000。默认值为 100，超过 100 后越大，则会使索引扫描的 COST 开销越高，从而导致查询优化器更加倾向于使用全表扫描。相反，值越小于 100，计算出来的索引扫描的开销就越低。

（6）索引缓冲调节参数 optimizer_index_caching

该参数用于设置在执行对索引的 in-list 迭代和嵌套循环联接时，优化器评估已经存在于 Buffer Cache 中的索引块的数量（以百分比的方式）。

参数的取值范围是 0～100，默认值为 0，取值越大，假设索引被缓存的数据块越多（也许与实际情形不符），优化器在评估 In-list 迭代和嵌套循环联接的索引扫描的开销 COST 就会越小，这样会越有利于索引的使用。

（7）星形转换参数 star_transformation_enabled

星形转换是通过对原来的 SQL 语句的隐式的改写来实现的，它能够很大程度地减少 I/O，开发人员并不需要知道有关星形转换的任何细节。数据库优化器会在合适的时候进行星形转换。要获得星形转换的最大性能，需要遵循以下三个基本条件：

1）将参数 star_transformation_enabled 设置为 true。

2）事实表上的维度列上存在外键。

3）事实表的每个外键上存在 BITMAP 索引。

参数 star_transformation_enabled 系统默认值 False。如果能够满足这三个条件，则星形模式的查询会使用 star_transformation，这是提高基于事实表的查询效率非常有力的优化技术。在 OLAP 系统中，星形转换被广泛地使用。

3.5.2 数据库对象的统计信息

优化器在决定如何访问表、索引或分区（Partition）时，为了能够做出正确的决策，需要对这类数据库对象有详细的了解，这就是关于数据库对象的统计数据。

对象统计信息是描述数据是如何在数据库中存储的，如一张表里面的记录数量、某一字段数据的分布、索引的高度和叶子节点数量等。这些信息有助于查询优化器找到正确高效的执行计划。假设有这样一个场景，有人要从 A 地去 B 地，哪种交通工具最快捷？汽车、火车还是飞机？是否可以骑自行车或步行？如果不知道 A 地和 B 地的详细信息，不知道两地之间的道路状况，就很难得到正确答案。同样，如果没有对象统计信息，查询优化器也找不到正确高效的执行计划。

有三种类型的对象统计信息可用：表统计、字段统计和索引统计，包括表分区和索引分区。这里的表、索引、分区都是用于数据存储的段（Segment），关于它们的统计信息实际上是关于“段”的统计数据。

（1）表的统计信息

数据字典视图 user_tab_statistics 给出了关于表的一系列统计信息，如数据的行数、涉及的数据块数、出现行链接和行迁移的记录数量等。

```
SQL>select num_rows,blocks,empty_blocks,avg_space,
2chain_cnt,avg_row_len
3 from user_tab_statistics where table_name='EMPLOYEE';
NUM_ROWSBLOCKS EMPTY_BLOCKS  AVG_SPACE  CHAIN_CNT AVG_ROW_LEN
-------------------------------------------------------------
10000       28          311        0        13
```

其中，NUM_ROWS：表中的数据行数；

BLOCKS：高水位线下的数据块数量；

EMPTY_BLOCKS：高水位线以上的数据块数量；

AVG_SPACE：数据块中的平均空闲空间（单位字节）；

CHAIN_CNT：出现行链接和行迁移的记录总数；

AVG_ROW_LEN：平均的记录长度（单位字节）。

（2）字段的统计信息

数据字典视图 user_tab_col_statistics 给出了关于用户表字段的统计信息，如 NUM_DISTINCT（非重复值数量）、LOW_VALUE（最小值）、HIGH_VALUE（最大值）、DENSITY（表示列的密度，这是一个判断列选择性的衡量指标，如果值为 0，表示非常有选择性，如果为 1，表示没有选择性）、NUM_NULLS（空值数量）、NUM_BUCKETS（直方图中的分段数量）、AVG_COL_LEN（平均长度）、HISTOGRAM（直方图类型）等。

直方图（Histogram）是一种关于列的统计信息，是对列中的数据分布状况进行描述的一类统计信息，它会按照某一列不同值出现数量的多少，以及出现的频率高低来绘制数据的分布情况，以便能够指导优化器根据数据的分布做出正确的选择。在某些情况下，表的列中的数值分布将会影响优化器使用索引还是执行全表扫描的决策。当 Where 子句的值具有不成比例数量的数值时，将出现这种情况，使全表扫描比索引访问的成本更低。如果 Where 子句的过滤谓词列上有一个合理的正确的直方图，将会对优化器做出正确的选择发

挥巨大的作用，使 SQL 语句执行成本最低从而提升性能。

从直方图中可以获得关于列的选择性。从一个行源中评估返回行数所占的比例就是选择率，选择率在 CBO 的查询优化中起着重要作用。选择率的取值范围是 0～1。粗略地讲，如果满足谓词条件的只有少量的行记录，那么 CBO 将更喜欢使用索引扫描；如果谓词条件要从表中获取大量数据，那么 CBO 将更喜欢使用全表扫描。

对于 ORACLE 而言，CBO 优化器可以根据直方图收集的列值分布信息，让选择性高（返回数据行比例少）的列值使用索引，而选择性低（返回数据行比例多）的列值不使用索引。尤其对于存在数据倾斜严重的列而言，直方图很重要。优化器对于字段数据的假设是平均分布的，对于那些非均匀分布的字段，需要收集直方图数据。

如何收集字段的直方图信息？在调用 dbms_stats.gather_table_stats 函数时，method_opt 确定是否收集关于列的统计信息及其直方图有三种模式：

1）为所有列收集统计信息及其直方图：

```
for all columns size n|skewonly|auto|repeat
```

其中，n：直方图桶数，取值范围为（1,254］，1 不收集直方图；

repeat：更新现有的直方图信息；

auto：由 ORACLE 自行决定 n 的大小；

skewonly：只收集非均匀分布列的直方图，系统决定桶数。

2）为所有列收集统计信息及在部分列上收集直方图：

```
for all columns size 1 for colmns size n col1,col2,……
```

3）只为部分列收集统计信息与直方图：

```
for columns size n col1,col2,……
```

（3）索引的统计信息

数据字典视图 user_ind_statistics 给出了关于用户索引的详细统计信息，它包含了索引的层级（blevel）、叶子块数量（leaf_blocks）、聚簇因子（clustering_factor）等，下面是视图主要字段的释义。

BLEVEL：索引的高度；

LEAF_BLOCKS：叶子节点的数量；

DISTINCT_KEYS：非重复值的数量；

AVG_LEAF_BLOCKS_PER_KEY：平均每个键值涉及的叶子节点数量；

AVG_DATA_BLOCKS_PER_KEY：平均每个键值引用的数据块的数量；

CLUSTERING_FACTOR：聚簇因子。

层级（Level）：表示从根节点到叶子块的深度。层级被 CBO 用于计算访问索引叶子块的成本，层级越大，表示从根节点到叶子块所需要访问的数据块的数量就越多，耗费的 I/O 就会越多，索引访问的成本就会越大。在数据库里，如果需要降低索引的层级，需要 rebuild 才可以。

聚簇因子：按照索引键值排序的索引行和存储于对应表中的数据行的存储顺序的相似程度。ORACLE 数据库按照以下算法计算聚簇因子：

1）聚簇因子初始值为 1。

2）ORACLE 首先定位到目标索引处于最左边的叶子块。

3）从最左边叶子块的第一个索引键值所在的索引行开始顺序扫描，在顺序扫描的过程中，ORACLE 会比较当前索引行的 rowid 和之前索引行的 rowid，如果这两个 rowid 并不是指向同一个表块，那么 ORACLE 就将聚簇因子的当前值递增 1；如果这两个 rowid 是指向同一个表块，ORACLE 就不改变聚簇因子的值。ORACLE 在比对 rowid 时并不会回表去访问相应的表块。

4）比对过程会持续下去，直到扫描完目标索引的所有索引块的所有索引行。

5）上述顺序扫描完成后，聚簇因子的当前值就是索引统计信息中的 clustering_factor，ORACLE 将其存储在数据字典里。

聚簇因子虽然是索引的统计数据，但它的取值与表的存储密切相关，它反映记录在表中的存储顺序和索引项顺序的相似程度。因此，能够降低聚簇因子的唯一方法就是对表中数据按照目标索引的索引键值排序后重新存储。

除了调用 dbms_stats.gather_index_stats 函数直接收集索引统计信息外，在调用 dbms_stats.gather_table_stats 函数时，将 cascade 参数指定为 True 也会一并收集表上所有索引的统计信息。

（4）对象统计信息的动态采样

在段对象（表、索引、分区）没有统计数据的情况下，ORACLE 采用“动态采样”技术来获取必要的统计信息，供 CBO 使用。动态采样（Dynamic Sampling）采样有限的数据块，估计的统计信息一般存在偏差，因此动态采样是 CBO 在段对象缺乏统计信息时的补救措施。至于具体的采样比例或数据块数，请参考 optimizer_dynamic_sampling 参数。

动态采样仅获得有限度的段对象统计数据，其信息的来源是部分采样的数据块和数据字典中的元数据。一般来说，全局临时表是没有统计信息的，针对临时表的访问必须使用动态采样技术来获得统计信息。

动态采样需要消耗额外数据库资源，在 OLTP 系统中，一些频繁执行的 SQL 语句绝不应该使用动态采样实现 SQL 解析。在 OLAP 系统中，通常 SQL 执行消耗的资源远大于 SQL 解析，动态采样的消耗可以忽略不计。

动态采样的特殊用途：在涉及相关联表或关联字段的查询中，各表及其列的统计信息是相互独立的，通过动态采样，CBO 能够获得不同字段之间的相关性统计信息。

3.5.3 系统统计信息

系统统计信息是关于 CPU 性能、磁盘 I/O 子系统性能的统计数据。在 CBO 环境中，ORACLE 依赖于对象的统计估算成本，以选择正确的 SQL 执行计划。随着版本的升级，ORACLE 逐步强化了 CPU 性能、I/O 性能对成本估算的影响。

ORACLE 提供了 dbms_stats.gather_system_stats 来收集系统统计信息。系统统计信息让优化器考虑服务器的 I/O 与 CPU 性能及其利用率，作为计算成本的依据；为每一个可选的执行计划估算 I/O 与 CPU 成本。因而对于 CBO 来说，获得准确的系统统计信息对于正确估计成本是不可或缺的。

系统统计信息存储在 sys.aux_stats$表中，其主要内容说明如下：

1）Cpuspeednw：表示非负载情况下的 CPU 速度，在系统启动时自动搜集。

2）Ioseektim：I/O 查找时间，以毫秒（ms）表示，默认为 10ms，非负载模式或可以手动设置。

3）Iotfrspeed：I/O 传输速度，表示 ORACLE 数据库单次读数据的传输速率，单位为 bytes/ms，在系统启动时自动收集，默认为 4096 bytes/ms。

4）Cpuspeed：表示负载情况下的 CPU 速度，以平均每秒可提供的 CPU 周期表示。

5）Maxthr：最大 I/O 吞吐量，单位为 bytes/s。

6）Slavethr：从属 I/O 吞吐量，表示并行进程时，从属进程的 I/O 吞吐量，单位为 bytes/s。

7）Sreadtim：单块读时间（如索引读取），表示随机读一个 ORACLE 数据块的时间，以 ms 计算。

8）Mreadtim：多块读时间（主要是指全表扫描），表示连续读取多个 ORACLE 数据库的平均时间，以 ms 计算。

9）Mbrc：多块读计数，表示一次多块读的读取的 ORACLE 数据块数量。

系统统计信息有工作负载与无工作负载两种类型。ioseektim、iotrfspeed、cpuspeednw 是无负载的统计信息，即系统不需要有工作负载，可以空闲时进行收集。ORACLE 为在系统启动时间重新设置，或重置为默认值。要手动收集非工作负载统计信息，使用 dbms_stats.gather_system_stats(gathering_mode => 'NOWORKLOAD')。当使用 dbms_stats.delete_system_stats()删除系统统计信息时间时，将只保留非负载时的统计信息:

```
SQL>exec dbms_stats.delete_system_stats( );
```

一般来说，需要采集系统高峰时段或典型负荷时段的系统统计信息。收集负载情况下的统计信息有两种方式，一种是手工指定收集时段的开始与结束。

```
exec dbms_stats.gather_system_stats(gathering_mode=>'START');
exec dbms_stats.gather_system_stats(gathering_mode=>'STOP');
```

另一种是使用间隔模式，指定一个间隔时段，ORACLE 自动开始与结束信息收集，如下面的指令以未来 30min 的系统负载做背景，收集系统统计信息。

```
exec dbms_stats.gather_system_stats(
gathering_mode => 'INTERVAL',interval =>30);
```

3.5.4　统计信息的维护与管理

统计信息的维护与管理主要依赖于 DBMS_STATS 包，该包提供一组函数和过程用于针对统计信息的各类需求，如统计信息的收集、删除、锁定、复制、导出、还原、设置（修改）等。

（1）DBMS_STATS 包

DBMS_STATS 包就被广泛用于统计信息的收集，用 DBMS_STATS 包收集统计信息也是 ORACLE 官方推荐的方式。在收集 CBO 所需要的统计信息方面，可以简单地将 DBMS_STATS 包理解成 analyze 命令的增强版。

DBMS_STATS 包最常见的四个存储过程：

1）dbms_stats.gather_table_stats：用于收集目标表、目标表上列及目标表上索引的统计

信息。

2）dbms_stats.gather_index_stats：用于收集指定索引的统计信息。

3）dbms_stats.gather_schema_stats：用于收集 schema 下所有对象的统计信息。

4）dbms_stats.gather_database_stats：用于收集整个数据库统计对象的统计信息。

（2）收集统计信息的调度

ORACLE 数据库统计信息的收集已经被整合到自动维护任务中，默认情况下能够满足大多数情形下的运行需求。对于具体的系统而言，在维护过程中可能需要调整其执行时间或者自行指定收集窗口。此处以 11g 版本为例。

```
SQL>select task_name, status
  2  from dba_autotask_task
  3  where client_name = 'auto optimizer stats collection';

TASK_NAME                    STATUS
-------------------------------------------------
gather_stats_prog             ENABLED

SQL>select program_action, number_of_arguments, enabled
  2  from dba_scheduler_programs
  3  where owner = 'SYS'
  4  AND program_name = 'GATHER_STATS_PROG';

PROGRAM_ACTION                          NUMBER_OF_ARGUMENTS ENABL
-------------------------------------------------------------------
dbms_stats.gather_database_stats_job_proc         0         TRUE

SQL>select window_group
  2  from dba_autotask_client
  3  where client_name = 'auto optimizer stats collection';

WINDOW_GROUP
-------------------------------------------------------------
ORA$AT_WGRP_OS

SQL>select w.window_name, w.repeat_interval,
  2  w.duration, w.enabled
  3  from dba_autotask_window_clients c, dba_scheduler_windows w
  4  where c.window_name = w.window_name
  5  AND c.optimizer_stats = 'ENABLED';

SQL>select * from dba_autotask_client_history
  2  where client_name LIKE '%stats%';
CLIENT_NAME: auto optimizer stats collection
WINDOW_NAME: SUNDAY_WINDOW
WINDOW_START_TIME: 16-JUL-17 07.39.44.429000 AM +08:00
WINDOW_DURATION: +000000001 00:02:51.571000
```

```
WINDOW_END_TIME: 17-JUL-17 07.42.36.000000 AM +08:00
```

（3）维护统计信息的调度方案

1）统计信息调度的启用与禁用。

```
EXEC DBMS_AUTO_TASK_ADMIN.ENABLE(
client_name=>'auto optimizer stats collection',
operation=>NULL,
window_name=>NULL);
EXEC DBMS_AUTO_TASK_IMMEDIATE.GATHER_OPTIMIZER_STATS();
```

2）单个时间调度窗口的设置。

```
EXEC DBMS_AUTO_TASK_ADMIN.DISABLE(
client_name=>'auto optimizer stats collection',
operation=>NULL,
window_name=>'MONDAY_WINDOW');
```

3）修改指定的时间调度窗口。

```
EXECDBMS_SCHEDULER.DISABLE(
name=>'SYS.FRIDAY_WINDOW',
force=>TRUE);
BEGIN
    DBMS_SCHEDULER.SET_ATTRIBUTE (
        name=> 'SYS.FRIDAY_WINDOW',
      attribute=>'REPEAT_INTERVAL',value=> 'FREQ=WEEKLY;BYDAY=FRI; BYHOUR=23;
BYMINUTE=30;BYSECOND=0');
END;
EXEC DBMS_SCHEDULER.ENABLE (name=>'SYS.FRIDAY_WINDOW');
```

第 4 章 执行计划的分析与干预

用户指令的执行最终是通过执行计划来体现的，执行计划决定用户指令执行的效率，这是毋容置疑的。但优化器决策最终形成特定执行计划是由特定的优化环境和对数据库对象的理解决定的。就目前的技术水平而言，优化器已日臻成熟，但绝算不上完美。因此，需要理解执行计划、关注执行计划，必要的时候适当干预执行计划。

4.1 观测执行计划

就一条特定的 SQL 指令而言，形成并查看执行计划有多种方式，本书主要使用其中的两种，一种是 SQL 跟踪，另外一种是 Explain plan，这两种方式各有特点，相互补充。前者除形成执行计划外，还提供一系列执行过程中的统计信息；后者的方便性在于指令并不需要得到执行就可单独形成执行计划（某些情况下，执行过程本身既耗时间又耗资源），并可利用 DBMS_XPLAN 实现定制化的输出。

4.1.1 查看执行计划

最直接的方式是在 SQL*Plus 里启用 SQL 跟踪，这样在执行 SQL 指令的过程中可以输出跟踪信息。这里的跟踪信息包含执行计划和执行过程中的数据统计，而这些统计信息对研究执行计划具有很好的参考价值，特别是执行过程中的 I/O 信息。

```
SQL>set autotrace traceonly
SQL>select e.*, d.*
  2  from employees e, departments d
  3  where e.department_id=d.department_id
  4  and employee_id=188;

Execution Plan
----------------------------------------------------------
Plan hash value: 2782876085

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    90 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |               |     1 |    90 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    69 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS   |    27 |   567 |     1   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN         | DEPT_ID_PK    |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("EMPLOYEE_ID"=101)
   5 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")

Statistics
----------------------------------------------------------
        33  recursive calls
         0  db block gets
         8  consistent gets
         1  physical reads
         0  redo size
      1621  bytes sent via SQL*Net to client
       520  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

1）recursive call：递归调用次数；ORACLE 在执行 SQL 的时候，会生成额外的对数据字典的访问，这被称为递归调用。

2）db block gets：从 buffer cache 中读取的 block 的数量。

3）consistent gets：从 buffer cache 中读取的 undo 数据的 block 的数量。

4）physical reads：从磁盘读取的 block 的数量。

5）redo size：生成的 redo 的大小。

6）sorts (memory)：在内存执行的排序量。

7）sorts (disk)：在磁盘上执行的排序量。

8）bytes sent via SQL*Net to client，bytes received via SQL*Net from client：客户端与数据库之间通过 sql*net 传递的数据量。

4.1.2　定制执行计划的输出

ORACLE 的执行计划包含非常丰富的信息，详细内容可以查看数据字典视图 V$SQL_PLAN 或当前用户下的表 PLAN_TABLE（用于存储由 Explain Plan 指令形成的执行计划信息）。如果没有此表，可在当前用户下运行 ORACLE 预置 utlxplan.sql 脚本创建此表。

事实上没有必要直接查看执行计划的原始信息，借助于 ORACLE 提供的程序包 DBMS_XPLAN 可以实现定制化的输出。表 4-1 所示为执行计划的主要信息。

表 4-1　执行计划的主要信息

类别	ALL	TYPICAL	SERIAL	说明
Basic	√	√	√	包含步骤 Id、Operation、Name
Alias	√			对象别名
Rows	√	√	√	估计的记录数

续表

类别	ALL	TYPICAL	SERIAL	说明
Bytes	√	√	√	估计的字节数
Cost	√	√	√	估计的优化代价
Partition	√	√	√	分区及分区剪裁信息
Parallel	√	√		输出与并行有关处理有关的信息
Predicate	√	√	√	谓词条件
Projection	√			结果集的投影信息
Remote	√			分布式 SQL 的有关信息
Note	√	√	√	备注解释部分

```
EXPLAIN PLAN[SET STATEMENT_ID = 'stmt_id']
FOR sql_statement;
SQL>explain plan for select e.*, d.*
  2  from employees e, departments d
  3  where e.department_id=d.department_id
  4  and employee_id=101;
Explained.

SQL>select * from table(dbms_xplan.display(null,null,'basic'));
PLAN_TABLE_OUTPUT
------------------------------------------------------------
Plan hash value: 2782876085
-----------------------------------------------------
| Id  | Operation                     | Name          |
-----------------------------------------------------
|   0 | select STATEMENT              |               |
|   1 |  NESTED LOOPS                 |               |
|   2 |   TABLE ACCESS BY INDEX ROWID | EMPLOYEES     |
|   3 |    INDEX UNIQUE SCAN          | EMP_EMP_ID_PK |
|   4 |   TABLE ACCESS BY INDEX ROWID | DEPARTMENTS   |
|   5 |    INDEX UNIQUE SCAN          | DEPT_ID_PK    |
-----------------------------------------------------

FUNCTION DISPLAY RETURNS DBMS_XPLAN_TYPE_TABLE
 Argument Name          Type            In/Out Default?
 ------------------------------------------------------------
 TABLE_NAME             VARCHAR2           IN    DEFAULT
 STATEMENT_ID           VARCHAR2           IN    DEFAULT
 FORMAT                 VARCHAR2           IN    DEFAULT
 FILTER_PREDS           VARCHAR2           IN    DEFAULT
```

在函数 dbms_xplan.display 的参数中，table_name 默认指当前用户下的 plan_table，statement_id 是由 explain plan 指令指定的语句标识符，format 限定执行计划输出的部分（参见表 4-1），filter_preds 指定过滤谓词。通过下面的查询定制当前执行计划的输出：

```
select * from table(dbms_xplan.display(null,null,'basic'));
select * from table(dbms_xplan.display(null,null,'basic +cost'));
select * from table(
   dbms_xplan.display(null,null,'typical -rows -bytes'));
```

另外，dbms_xplan 包中还有一个函数 display_cursor( )可以用来查看共享池中已有的执行计划，使用方法与 dbms_xplan.display 类似，不同的是调用此函数需要提供 SQL_ID（从视图 V$SQL 中可获得）。其中，参数 CURSOR_CHILD_NO 是指同一 SQL 语句在共享池中可能存在不同版本的执行计划，默认是 0（子游标编号）。同一 SQL 语句存在多个执行计划是由优化环境或统计数据的变化导致的。

```
FUNCTION DISPLAY_CURSOR RETURNS DBMS_XPLAN_TYPE_TABLE

 Argument Name          Type                In/Out      Default?
 ------------------------------------------------------------------
 SQL_ID                 VARCHAR2            IN          DEFAULT
 CURSOR_CHILD_NO        NUMBER(38)          IN          DEFAULT
 FORMAT                 VARCHAR2            IN          DEFAULT
```

4.2　认识执行计划

下面来研究一个查询案例。

```
SQL>EXPLAIN PLAN FOR
  2  select
  3  e.employee_id, j.job_title, e.salary, d.department_name
  4  from employees e, jobs j, departments d
  5  where e.employee_id < 103
  6  AND e.job_id = j.job_id
  7  AND e.department_id = d.department_id;
Explained.

SQL>select * from table(
  2  dbms_xplan.display(null,null,'basic +cost'));
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------
Plan hash value: 2963623819
-------------------------------------------------------------------
-------------------------------------------------------------------
| Id  | Operation              | Name     | Cost (%CPU) |
-------------------------------------------------------------------
|   0 | select STATEMENT                  |             |8(13)  |
|   1 |  NESTED LOOPS                     |             |       |
|   2 |   NESTED LOOPS                    |             |8(13)  |
|   3 |    MERGE JOIN                     |             |5(20)  |
|   4 |     TABLE ACCESS BY INDEX ROWID   | JOBS        |2(0)   |
```

```
|   5 |     INDEX FULL SCAN                  | JOB_ID_PK |1(0)   |
|   6 |    SORT JOIN                         |           |3(34)  |
|   7 |     TABLE ACCESS BY INDEX ROWID |EMPLOYEES      |2(0)   |
|   8 |      INDEX RANGE SCAN          | EMP_EMP_ID_PK    | 1(0)  |
|   9 |   INDEX UNIQUE SCAN            | DEPT_ID_PK       |0(0)   |
|  10 |  TABLE ACCESS BY INDEX ROWID   | DEPARTMENTS|1(0)       |
-------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   6 - access("E"."JOB_ID"="J"."JOB_ID")
       filter("E"."JOB_ID"="J"."JOB_ID")
   8 - access("E"."EMPLOYEE_ID"<103)
   9 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

在这个具有 10 个步骤的执行计划里，可以这样理解这些步骤之间的先后关系，如图 4-1 所示，注意步骤之间的缩进关系，缩进最多的步骤最先得到执行，其基本的执行流程是由右向左、从上至下的执行顺序：

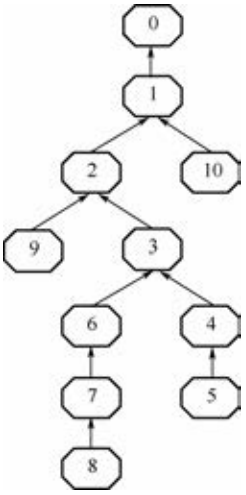

图 4-1　执行计划中的步骤及其关系

1）在主码索引 EMP_EMP_ID_PK 上执行索引范围扫描（步骤 8）。

2）由步骤 8 获得的 ROWID 访问表 EMPLOYEES（步骤 7）。

3）在主码索引 JOB_ID_PK 上执行索引范围扫描（步骤 5）。

4）由步骤 5 获得的 ROWID 访问表 JOBS（步骤 4）。

5）根据条件"E"."JOB_ID"="J"."JOB_ID"执行过滤、排序（步骤 6）。

6）将步骤 4 和步骤 6 对应的行源进行融合关联（步骤 3）。

7）在主码索引 DEPT_ID_PK 上执行索引唯一性扫描（步骤 9）。

8）将步骤 3 和步骤 9 对应的行源执行嵌套循环（步骤 2）vb

9）由于在步骤 9 中已经扫描了索引 DEPT_ID_PK，根据获得的 ROWID 直接访问表 DEPARTMENTS（步骤 10）。

10）将步骤 2 和步骤 10 对应的行源执行嵌套循环（步骤 1）。

11）获得最终的查询结果集（步骤 0）。

一般，当查看一份执行计划时，可以从以下角度去分析其中的执行步骤及其先后关系：

1）首先查看缩进量最多的行，这些步骤会最先得到执行。

2）从上到下，如果多行缩进量相同，则按照从上到下的顺序去查看。

3）从右向左，根据缩进量由多到少的原则查看缩进量相对减少的步骤，向左推进。

4）广义上，执行计划中的每一步骤都会产生一个行源（结果集），缩进相同的两个步骤之间的行源一般需要根据某种关联条件执行联接（Join）。

5）重复上面的查看思路直到执行计划的最上层（缩进最小的行），执行完毕。

在上面的步骤查看的过程中，可以带着以下问题分析其中的每一个步骤：

1）每一步骤执行的访问路径（表扫描还是索引扫描、扫描方式）。

2）每一步骤涉及的数据库对象、记录数、字节数、对应的局部代价。

3）根据前面并行步骤获得的行源信息，考虑行源之间的联接关系及其操作选项。

4）注意每一步骤对行源数据的筛选，关注筛选条件及其相关索引。

5）如果计划的某一步骤存在异常（如访问路径异常、估计的记录数异常、估计的代价异常、估算的时间异常、联接方式异常等），请注意检查统计信息的时效性、索引的类别状态及有效性，以及对象的物理存储信息（段结构）等，从而分析执行计划的合理性。

4.3　对多表联接的分析

4.3.1　多表联接概述

从多个相关联的表中提取出用户需要的数据，是任何数据库应用的核心功能之一，有效地处理多表联接也是现代数据库系统的精髓之一。执行多表查询的 SQL 语句时，ORACLE 优化器不仅要确定每个单表的访问路径，而且需要确定这些表的联接顺序和联接方法。

1）定性的目标之一是在执行过程中应尽早过滤掉不需要的数据，减少后续步骤需要处理的数据量。这与多表之间的联接顺序（包括驱动表）有关。

2）虽然在 SQL 语句中可以处理多个表中的数据，但在具体的联接过程中，每一次只能处理两个表，即多表联接一定被分解为两两联接。

3）两个表之间的联接方法与数据量有密切的关系，随着数据量的增加，不同联接方法之间的效率就会显现出来。

4）排序是比较消耗资源的操作，特别是对大数据量的排序，因此应尽可能避免对数据的排序，即能不排序就不排序。另外充分利用索引可以减少某些排序的必要性，因为索引中的数据（叶块中索引条目的索引键值）是排序的，因此通过索引获得的数据是有序的。

4.3.2　联接条件和类型

在一个多表操作的 SQL 语句中，同时存在过滤条件和联接条件是非常普遍的。这里要区分一下两者。过滤条件（Filtering Condition）又称限制条件或约束条件（Restriction Condition），是用来指定从表中筛选数据的标准，而联接条件（Join Condition）是用来设置两表联接的标准。在传统的 SQL 语法中，往往将两者混为一谈，不区分它们，这两类条件通常都写在 Where 子句中。最近发布的几个 SQL 标准（从 ANSI/ISO SQL_92 开始）都有意识地将两者区别开来，过滤条件写在 Where 子句中，而联接条件则使用关键字 ON 引导写在 From 子句中。下面结合联接条件的说明，对新旧 SQL 的联接语法做一简要比较。

（1）笛卡儿乘积

笛卡儿乘积是集合的概念，从数据库系统多表联接的角度看，它又被称为交叉联接，两个表（分别由 M、N 条记录）的交叉联接的结果将产生 M×N 条记录，如果两个表分别有 1 万条记录，交叉联接将产生 1 亿条记录。因此，这里的交叉联接在实际应用中很少直

接使用，但两个表各自记录的子集之间产生交叉联接形成需要的结果是非常常用的，这也是考察多表联接的基础。

当进行多表查询而没有指定联接条件时（可以对表指定约束条件），将产生笛卡儿乘积的结果集。下面两种 SQL 的写法都有如下相同的执行计划。

```
SQL>select e.ename, d.dname
  2  from emp e cross join dept d
  3  where e.ename like 'M%' and d.deptno>30;

SQL>select e.ename, d.dname
  2  from emp e, dept d
  3  where e.ename like 'M%' and d.deptno>30;

Execution Plan
----------------------------------------------------------
Plan hash value: 2034389985
-------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |    98 |  1666 |    11   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN|      |    98 |  1666 |    11   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL  | DEPT |     7 |    77 |     3   (0)| 00:00:01 |
|   3 |   BUFFER SORT        |      |    14 |    84 |     8   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL | EMP  |    14 |    84 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

（2）多表的条件联接

两表之间的联接分为交叉联接、内联接、外联接，其中外联接又分为左外联接、右外联接和全外联接。ANSI/ISO SQL_92 之后的标准，两表之间的联接查询的基本语法为

```
SQL>select ……
from table1 [cross | inner | outer ] join table2 on { join condition }
where { filtering condition }
```

其中，内联接是默认的联接方式，就是通常意义上的条件联接，关键字 inner 可以省略。外联接中有保留表（preserved table）的概念，对于保留表，那些不符合联接条件的记录也会出现在结果集中。根据指定保留表的位置，外联接使用的联接关键字有 left outer join、right outer join 和 full outer join 三种形式，对应地分别指定 join 的左表、右表、两边的表为保留表。传统的 SQL 语法中，在两表联接条件的一边或两边使用（+）表示外联接。还有一种联接称为自然联接（Natural Join），它与内联接相似，区别在于使用该联接时，ORACLE 会自动使用两个表中的相同列（通常两表中的相同列具有主外键关系）进行联接，而不需要用户显式地指定联接条件。

下面的两个查询语句使用了新旧不同的 SQL 语法（注意语法上的区别），功能相同，执行计划也相同（Plan hash value 不同），它们不仅给出了职工和部门的对应关系信息，同时也给出了那些没有职工的部门信息（结果集中对应于 emp 职工表的字段以 null 填充）。

```
SQL>select e.empno,e.ename,e.sal,d.deptno,d.dname
  2  from emp e right outer join dept d
```

```
  3  on e.deptno = d.deptno;

SQL>select e.empno,e.ename,e.dal,d.deptno,d.dname
  2  from emp e, dept d
  3  where e.deptno(+) = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: .........
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    14 |   378 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN OUTER   |      |    14 |   378 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPT |     7 |    98 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

下面给出一个多表联接查询的多版本示例（注意使用了其他不同的联接条件），都是查询出暂时还没有职工信息的部门。

```
SQL>select deptno from dept
  2  minus (select distinct deptno from emp);

Execution Plan
----------------------------------------------------------
Plan hash value: 2288528972
-------------------------------------------------------------------------------
| Id  | Operation           | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |         |     7 |    30 |     6  (84)| 00:00:01 |
|   1 |  MINUS              |         |       |       |            |          |
|   2 |   SORT UNIQUE NOSORT|         |     7 |    21 |     2  (50)| 00:00:01 |
|   3 |    INDEX FULL SCAN  | PK_DEPT |     7 |    21 |     1   (0)| 00:00:01 |
|   4 |   SORT UNIQUE       |         |     3 |     9 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| EMP     |    14 |    42 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

SQL>select deptno from dept d
  2  where not exists
  3  (select * from emp e where e.deptno = d.deptno);

Execution Plan
----------------------------------------------------------
Plan hash value: 4049784039
-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     5 |    30 |     5  (20)| 00:00:01 |
|*  1 |  HASH JOIN ANTI    |         |     5 |    30 |     5  (20)| 00:00:01 |
|   2 |   INDEX FULL SCAN  | PK_DEPT |     7 |    21 |     1   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP     |    14 |    42 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

```
SQL>select deptno from dept
  2  where deptno not in (select distinct deptno from emp);

Execution Plan
----------------------------------------------------------
Plan hash value: 304125360

--------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     6 |    18 |     8   (0)| 00:00:01 |
|*  1 |  INDEX FULL SCAN   | PK_DEPT |     1 |     3 |     1   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| EMP     |     2 |     6 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------


SQL>select d.deptno from emp e, dept d
  2  where e.deptno(+)=d.deptno and e.deptno is null;

Execution Plan
----------------------------------------------------------
Plan hash value: 4049784039

--------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     5 |    30 |     5  (20)| 00:00:01 |
|*  1 |  HASH JOIN ANTI    |         |     5 |    30 |     5  (20)| 00:00:01 |
|   2 |   INDEX FULL SCAN  | PK_DEPT |     7 |    21 |     1   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP     |    14 |    42 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

上面的查询在联接条件中用到 IN 和 Exists 关键字，两者都是从两个相关联的结果集中搜寻数据，从一个结果集中参照另一个结果集找到匹配数据，使用的是子查询，这类联接方式比较松散，称为半联接；相反，还有另一类联接在联接条件中用到 NOT IN 和 NOT Exists 关键字，它们是参照一个结果集从一个结果集中排除数据，同样使用的子查询，这种用法正好和半联接相反，被称为反联接。

（3）避免使用子查询

一般来说，使用子查询（Sub-query）的复合查询的效率要低于多表之间的直接联接（Join），因此使用子查询被称为半联接。一方面，ORACLE CBO 优化器在 SQL 语句的解析（Parse）阶段，会尽可能地将子查询转化为多表联接操作，但对于复杂的 SQL 语句（特别是逻辑混乱的 SQL 语句），优化器则不能实现这种自动转化；另一方面，子查询往往会导致优化器对整个 SQL 语句选择不合理的执行计划，使整个 SQL 语句的执行效率显著下降。因此，优化器应尽可能避免使用子查询。

4.3.3 两两联接的方法

根据现代数据库建模的设计规范，表由实体（Entity）转化而来，表中的数据则是实体的表示，实体与实体之间的关联关系由表结构及其中的数据表现出来。从多个表中提取出相关联的数据则是数据库应用必须处理的基本问题之一。正因为如此，高效地处理表与表

之间的联接是现代关系数据库系统的精髓之一。

在一个复杂的 SQL 语句中，不管有多少个表进行关联，但 ORACLE 一定是将它们转换为多个两两之间的联接，这里就存在两个问题，一是两表（或两个结果集 Records Set）之间的联接方法，二是两两之间的联接顺序。正确地处理这两个问题，多表联接问题就会迎刃而解。

对于 OLTP 系统，对于多表联接中的联接顺序问题，有一个非常确定的原则，那就是从选择性、限制性强的数据开始，尽可能将较小的结果集传递到下一个环节，并且在过滤数据的过程中充分利用索引去选择性地访问数据。

对于复杂的、多步骤的数据关联，当然可以在系统的应用层处理，但在绝大多数情况下这种处理方式的效率一定会远低于把这种关联关系交给 DBMS 去处理的效率，所以要充分利用 DBMS 提供的数据集之间的联接方法。虽然具体的联接方法多种多样，但它们都是从三种基本的联接方法中派生出来的。因此，如果掌握了这三种联接方法的联接机制及其适用环境，就可以很好地理解和处理其他衍生的情形。这三种基本的联接方法是嵌套循环（Nested Loop）、融合联接（Merge Join）和散列联接（Hash Join）。

（1）嵌套循环

当两个表或两个数据集进行关联时，分别使用循环的方式对数据集进行遍历，相当于手工编程时使用两套循环遍历一个二维数组一样，嵌套循环同时使用两套游标循环访问数据集中的行记录，并进行有条件的匹配，其中的外循环（Outer Loop）被称为驱动循环，涉及的数据集被称为驱动数据源（Driving Row Source）或驱动表（Driving Table）。对应地，内循环（Inner Loop）涉及的数据集则是被驱动的（Drived）。两表或两个数据集的这种联接方法是较常用的、也是较经典的联接方法。

首先来单纯地看两个表的嵌套循环。请看下面的执行计划。

```
SQL>select /*+ ordered use_nl(e d) no_index(d) */
  2  e.empno,e.ename,d.deptno,d.dname
  3  from  emp e, dept d
  4  where e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 1189851940

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    14 |   378 |    18   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |      |    14 |   378 |    18   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| DEPT |     1 |    14 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

这个执行计划的嵌套循环按照表的书写顺序，ORACLE 选择了 EMP 作为驱动表。这个执行计划的执行步骤非常类似于使用如下的 PL/SQL 语言编程，使用双重游标循环来获得匹配的数据。

```
SQL>declare
```

```
 2  cursor c_emp is select empno, ename, deptno from emp;
 3  cursor c_dept is select deptno, dname from dept;
 4  r_emp c_emp%rowtype;
 5  r_dept c_dept%rowtype;
 6  begin
 7  open c_emp;
 8  loop
 9  fetch c_emp into r_emp;
10  exit when c_emp%notfound;
11   open c_dept;
12   loop
13    fetch c_dept into r_dept;
14    exit when c_dept%notfound;
15    if r_dept.deptno = r_emp.deptno then
16     dbms_output.put_line(
17     r_emp.empno||r_emp.ename||r_dept.deptno||r_dept.dname);
18    end if;
19   end loop;
20   close c_dept;
21  end loop;
22  close c_emp;
23  end;
24  /
```

如果选择 DEPT 作为驱动表，再来检查 ORACLE 的执行情况。

```
SQL>select /*+ ordered use_nl(d e) no_index(e) */
  2  e.empno,e.ename,d.deptno,d.dname
  3  from  dept d, emp e
  4  where e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 4192419542

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    14 |   378 |    11   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |      |    14 |   378 |    11   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPT |     7 |    98 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMP  |     2 |    26 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

从上面的两个执行计划中可以看出，同样是嵌套循环，同样都是全表扫描，两个执行计划估计耗费的代价却明显不同，由此可见，驱动表的选择对于执行计划的效率有明显的影响。EMP 表中有 14 条记录，DEPT 表中有 7 条记录，选择 DEPT 作驱动表，内循环 EMP 表上的扫描效率要高于前者，这也正是效率提高的原因。一般来说，应该选择选择性强、记录数少的表作为驱动表。

下面结合索引的使用来考查嵌套循环的效率。

```
SQL>select /*+ ordered use_nl(e d) */
  2  e.empno,e.ename,d.deptno,d.dname
  3  from  emp e, dept d
  4  where e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 351108634
--------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |    14 |   378 |     5   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |         |    14 |   378 |     5   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL          | EMP     |    14 |   182 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     1 |    14 |     1   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | PK_DEPT |     1 |       |     0   (0)| 00:00:01 |
--------------------------------------------------------------------------------------
```

优化器根据联接字段的索引情况选择 EMP 表作为驱动表，在表 EMP 中每选择一行，在 DEPT 表中执行 INDEX UNIQUE SCAN 找到匹配的行，这里驱动表 EMP 执行的是全表扫描，而内循环的被驱动表执行的是索引扫描。

根据嵌套循环的一般性原则，应该选择 DEPT 作为驱动表才是正确的选择，然而 ORACLE 为什么没有选择 DEPT 作驱动表而选择 EMP 呢？这里的原因在于索引的使用，如果选择 DEPT 作为驱动表，由于内循环的 EMP 表在联接字段没有索引，那只能是全表扫描，这会导致估计的该执行计划耗费的代价更大。如果在内循环 EMP 表的联接字段 deptno 上建立索引，情况就会不同，执行计划耗费的代价应该能够下降。

```
SQL>create index i_edeptno on emp(deptno);
Index created.

SQL>select /*+ ordered use_nl(d e) */
  2  e.empno,e.ename,d.deptno,d.dname
  3  from  dept d, emp e
  4  where e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 3789599609
--------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |    14 |   378 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP       |     2 |    26 |     1   (0)| 00:00:01 |
|   2 |   NESTED LOOPS               |           |    14 |   378 |     4   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL         | DEPT      |     7 |    98 |     3   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | I_EDEPTNO |     5 |       |     0   (0)| 00:00:01 |
--------------------------------------------------------------------------------------
```

由此可以看到，此时内循环 EMP 表上执行的是索引范围扫描，执行计划耗费的代价进一步地降低。这个例子表明，嵌套循环需要和索引配合使用，才能取得很好的效果。

上面的查询演示表明，在被驱动表（内循环）的联接字段建立索引是必要的，有时根据多字段的联接条件，还需要建立必要的复合索引。因此，嵌套循环和索引相辅相成，嵌套循环离不开索引的配合。

（2）融合联接

多表联接的最终目的是根据应用的需要将多个表中的数据根据一定的条件融合到一起，关键在于融合的方法。前面介绍的嵌套循环是一种经典的方法，融合联接（Merge Join）采用另外一种思路，它首先将两个需要联接的表或数据集按照联接字段进行排序，然后将排序后的数据关联到一起，这种联接方法又称为排序融合联接（Sort Merge Join）。由于这类联接的过程需要经过至少两次的排序，所以一般情况下它的联接效率要低于其他两种联接方法。

在执行融合联接的过程中，两个数据集都要被完全读出并进行排序，所以这种联接方法没有驱动表的概念。下面使用融合联接来考查上面的复合查询。

```
SQL>select /*+ use_merge(e d) no_index(e) no_index(d) */
  2  e.empno,e.ename,d.deptno,d.dname
  3  from  emp e, dept d
  4  where e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 1407029907
-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    14 |   378 |     8  (25)| 00:00:01 |
|   1 |  MERGE JOIN         |      |    14 |   378 |     8  (25)| 00:00:01 |
|   2 |   SORT JOIN         |      |     7 |    98 |     4  (25)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| DEPT |     7 |    98 |     3   (0)| 00:00:01 |
|*  4 |   SORT JOIN         |      |    14 |   182 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

从执行计划可以看出，两表通过全表扫描后完全读出，之后分别进行排序，最后执行融合操作。如果两个数据集在排序过程中能够充分利用索引，效率会得到提高。

```
SQL>select /*+ use_merge(e d) */
  2  e.empno,e.ename,d.deptno,d.dname
  3  from  emp e, dept d
  4  where e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 710178378
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |    14 |   378 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                  |           |    14 |   378 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP       |    14 |   182 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | I_EDEPTNO |    14 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |           |     7 |    98 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL         | DEPT      |     7 |    98 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

这里 ORACLE 对 EMP 表利用了之前在 EMP 表 deptno 字段上创建的索引，使得该执行计划的资源消耗由原来的 8 下降到 6。注意，这里 DEPT 表在联接字段 deptno 上是有索引（主码）的，但 ORACLE 并没有利用索引，而是选择了全表扫描，这是由于 DEPT 表中的数据较少，如果选择索引扫描，估计的资源消耗与全表扫描相当。

```
SQL>select /*+ use_merge(e d) index(d) */
  2  e.empno,e.ename,d.deptno,d.dname
  3  from  emp e, dept d
  4  where e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 1808643156
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |    14 |   378 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                  |           |    14 |   378 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID | EMP       |    14 |   182 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | I_EDEPTNO |    14 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |           |     7 |    98 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS BY INDEX ROWID| DEPT      |     7 |    98 |     3   (0)| 00:00:01 |
|   6 |     INDEX FULL SCAN          | PK_DEPT   |     7 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

从上面的实例可以看出，融合联接同样需要利用索引，因为 B 树索引天生就有排序的特性，因此充分利用索引对索引键值排序的特性是降低融合联接资源消耗的关键。

另外需要注意的是，在比较复杂的多表查询中，多重限制条件有可能形成多重的融合联接，而多重融合联接又是根据不同的联接字段关联的，这就有可能在一次复合查询中对同一部分数据形成多重排序，从而导致资源消耗的显著增加和执行效率的显著下降，在工程上需要防止这种情况的发生。

（3）散列联接

多表之间的散列联接（Hash Join）与 ORACLE 的散列算法（Hash Algorithm）密切相关。散列算法是一种与索引并列、用于快速查找并定位数据的技术。对于一个数据集来说，ORACLE 通过对某个字段实施散列运算来构建散列表。对数据项执行散列运算的结果可以理解为数据项存储的序号，称为散列键值，散列表中包含数据项和散列键值的对应关系。反之，若要根据该字段查询数据集中的数据，利用散列表，可以实现对数据项的快速查找与定位，从而避免对全体数据项的扫描。因此，散列表（Hash Table）是一种以存储空间换取访问时间的数据结构。

当两个数据集进行散列联接时，ORACLE 首先选择其中的一个数据集（通常是较小的数据集），在联接字段上实施散列运算构建散列表。然后选择另外一个数据集按行对联接字段同样实施散列运算，对照散列表，获得对应的匹配记录，从而实现两个数据集的散列联接。散列联接的过程中，用于构造散列表的数据集称为构造输入（Build Input），用于在散列表中查找匹配项的数据集称为探测输入（Probe Input）。

```
SQL>select /*+ use_hash(d) no_index(e) */
  2  e.empno,e.ename,d.deptno,d.dname
  3  from  emp e, dept d
```

```
  4  where e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 615168685
--------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    14 |   378 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN         |      |    14 |   378 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPT |     7 |    98 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP  |    14 |   182 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

上面的散列联接中，选择 DEPT 表作为构造输入，EMP 表则为探测输入。当缺少可利用的索引时，散列联接比嵌套循环联接更加有效。散列联接也可能比（排序）融合联接更快，因为当散列表构造完毕时，探测输入获得第一条匹配记录就可返回输出，而融合联接要等到两个数据集分别排序完成后才能进行融合。

散列联接在构造散列表、进行散列匹配时需要足够的用户工作区 PGA 内存空间（hash_area_size），当联接的数据集较大而又没有足够的内存空间时，数据就会溢出到临时段，这会显著增加临时表空间的 I/O，极大地降低散列联接的效率。

散列联接的有效性取决于散列算法，散列算法的优势在于它的算法复杂度为 O(1)，也就是说，一旦散列表构造完毕，利用散列算法查找数据项需要的时间与数据量的大小无关，是个常量，这也正是散列联接通常更适合数据量大的情况的原因。另外，散列联接只适合等值联接的情形。

从前面的描述可以看出，散列联接并不需要索引，索引访问和散列访问是访问数据的两种方式。但散列联接中有一种特殊的情况，就是索引联接，它是发生在同一张表上不同索引之间的联接。如果一个查询存在多个过滤条件，每个过滤条件都有对应的索引，且只需要通过访问不同的索引就可获得查询需要的数据，那么此时使用索引联接最为有效。索引联接只能发生在散列联接中，即索引之间发生的散列联接称为索引联接。

```
SQL>select /*+ index_join(e pk_emp i_edeptno) */
  2  e.empno,e.deptno
  3  from  emp e
  4  where empno between 7500 and 8000
  5  and e.deptno > 20;

Execution Plan
----------------------------------------------------------
Plan hash value: 442549435
--------------------------------------------------------------------------------
| Id  | Operation          | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                  |     5 |    35 |     3  (34)| 00:00:01 |
|*  1 |  VIEW              | index$_join$_001 |     5 |    35 |     3  (34)| 00:00:01 |
|*  2 |   HASH JOIN        |                  |       |       |            |          |
|*  3 |    INDEX RANGE SCAN| I_EDEPTNO        |     5 |    35 |     2  (50)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN| PK_EMP           |     5 |    35 |     2  (50)| 00:00:01 |
--------------------------------------------------------------------------------
```

上面的查询虽然只是从单表中搜索数据，联接却发生在 I_EDEPTNO 和 PK_EMP 两个

索引之间，该查询并没有访问表中的数据。工程上如果能够充分利用索引联接，查询效率就会得到极大的提高。

（4）三种联接方式比较

上面介绍的三种多表联接方式是 ORACLE 进行多表查询的主要联接方式，它们在使用过程中各有其适应范围，表 4-2 所示为三种联接方式的比较。

表 4-2　三种联接方式的比较

比较项	嵌套循环	融合联接	散列联接
优化器 hint	use_nl	use_merge	use_hash
联接条件	所有联接可用	所有联接可用	仅用于等值联接
消耗资源	CPU、磁盘 I/O	PGA、临时空间	PGA、临时空间
优点	特别适合选择性强的联接条件，与索引配合使用，联接效率很高	两数据集的排序可以并行执行，当数据集有预排序时，效果更好	无须索引的支持，特别适合大数据量，联接过程中就可以返回记录
缺点	无索引利用时，效率低	排序效率限制其效率	构建大散列表耗内存
适用系统	主要用于 OLTP	OLTP 和 OLAP	大数据量，OLAP

4.4　干预执行计划

在数据库性能调优过程中，经常面临单个 SQL 语句的问题。如果在应用常规的调优手段后（如更新统计信息、重建索引、改善表存储等）仍然不能优化其性能，依然可以针对单个 SQL 语句做出调整，这个手段就是优化提示（Hint）。优化提示不应该是优化 SQL 语句性能优先考虑的选项，而应该是最后的选项。

顾名思义，Hint 是一种用户提示或者一种用户建议，让优化器在做决策的过程中增加一种选择路径。但这种提示不是绝对的，当优化器认为当前 Hint 不是合理的选项时它会完全忽略，因此优化提示 Hint 可以看作镶嵌在 SQL 语句里的特殊注释。

值得注意的是，虽然在实践中使用的大部分 Hint 都是针对优化器的，但并不是所有的 Hint 都是针对优化器的，典型的例外，如 Append、Cache 等优化提示，它们分别与 I/O、缓存有关。

4.4.1　优化提示的使用

ORACLE 的优化提示写在带加号（+）的注释里，格式如下：

```
/*+ hint1 hint2 … */
--+ hint1 hint2 …
```

其位置必须写在 SQL 语句第一个关键字之后，这里的符号“/*+”或“--+”是一体的，中间不能存在空格。一般要求这里的加号（+）与第一个 Hint 之间添加一个空格（非强制），多个 Hint 之间以空格隔开（强制）。

由于第二种优化提示的写法并没有像第一种写法那样存在明确的结束符（*/），故此种格式要求 SQL 语句的第一个关键字和优化提示单独作为一行处理。

可以应用优化提示的语句有 Select 语句和 DML 语句（Insert、Delete、Update、Merge），DDL 语句无法应用优化提示。如果在这些 SQL 语句中引用的表使用了别名（Alias），则在

优化提示中必须引用别名。

优化提示是一种辅助性的 SQL 调优手段，ORACLE 有个布尔型的隐含初始化参数 _optimizer_ignore_hints 可以控制优化提示的启用和禁用，其默认值为 False。该参数可以在实例级或会话级设置：

```
SQL>alter session set "_optimizer_ignore_hints" = true;
```

优化提示不应该被滥用。在实践中，如果要取消某些系统里大量使用 Hint 而造成的执行计划混乱，可以在实例级将此参数设置为 true，即可消除滥用优化提示给系统带来的负面影响。

```
SQL>select * from emp where empno=7788;
Execution Plan
----------------------------------------------------------
Plan hash value: 2949544139
-------------------------------------------------------------------
| Id  | Operation                    | Name   | Rows  | Bytes |
-------------------------------------------------------------------
|   0 | select STATEMENT             |        |    1 |    40  |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |    1 |    40  |
|*  2 |   INDEX UNIQUE SCAN          | PK_EMP  |    1 |        |
-------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO"=7788)

SQL>select /*+ full(emp) */ * from emp where empno=7788;
Execution Plan
----------------------------------------------------------
Plan hash value: 3956160932
--------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | select STATEMENT  |      |    1 |    40 |    3   (0)|
|*  1 |  TABLE ACCESS FULL| EMP  |    1 |    40 |    3   (0)|
--------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("EMPNO"=7788)
```

4.4.2 与优化模式有关的 Hint

1）指示优化器以吞吐量为目标：

```
/*+ all_rows */
```

2）指示优化器以响应时间为目标：

```
/*+ first_rows(n) */
```

3）应用基于规则的优化器 RBO：

```
/*+ rule */
```

这里一个有趣的现象是，与将初始化参数 optimizer_mode 设置为 first_rows_n（n 只能取 1、10、100、1000）不同的是，优化提示 first_rows(n)中的 n 可以取任意的整数。

```
SQL>select /*+ first_rows(7) */ * from emp;
Execution Plan
----------------------------------------------------------
Plan hash value: 3956160932
--------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | select STATEMENT  |      |    7 |   280 |    3   (0)|
|   1 |  TABLE ACCESS FULL| EMP  |    7 |   280 |    3   (0)|
--------------------------------------------------------------------

SQL>select /*+ rule */ * from emp where empno=7788;
Execution Plan
----------------------------------------------------------
Plan hash value: 2949544139
-----------------------------------------------
| Id  | Operation                     | Name  |
-----------------------------------------------
|   0 | select STATEMENT              |       |
|   1 |  TABLE ACCESS BY INDEX ROWID | EMP       |
|* 2 |   INDEX UNIQUE SCAN           | PK_EMP|
-----------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO"=7788)
Note
-----
   - rule based optimizer used (consider using cbo)
```

4.4.3　与表有关的 Hint

1）指示优化器执行全表扫描：

```
/*+ full(table_name) */
```

2）按照表在语句中的顺序执行联接操作：

```
/*+ ordered */
```

3）按照提示中的顺序执行联接操作：

```
/*+ leading(table1 table2) */
```

```
SQL>select /*+ ordered */ e.*,d.*
  2  from dept d,emp e
  3  where e.deptno=d.deptno;
Execution Plan
----------------------------------------------------------
Plan hash value: 844388907
-------------------------------------------------------------------
| Id  | Operation                    | Name    | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | select STATEMENT             |         |    6  (17)|
|   1 |  MERGE JOIN                  |         |    6  (17)|
|   2 |   TABLE ACCESS BY INDEX ROWID | DEPT    |    2   (0)|
|   3 |    INDEX FULL SCAN           | PK_DEPT |    1   (0)|
|*  4 |   SORT JOIN                  |         |    4  (25)|
|   5 |    TABLE ACCESS FULL         | EMP     |    3   (0)|
-------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  4 - access("E"."DEPTNO"="D"."DEPTNO")
      filter("E"."DEPTNO"="D"."DEPTNO")

SQL>select /*+ leading(e d) */ e.*,d.*
  2  from dept d,emp e
  3  where e.deptno=d.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 1123238657
-------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | select STATEMENT  |      |    14 |   840 |    7  (15)|
|*  1 |  HASH JOIN        |      |    14 |   840 |    7  (15)|
|   2 |   TABLE ACCESS FULL| EMP  |    14 |   560 |    3   (0)|
|   3 |   TABLE ACCESS FULL| DEPT |     4 |    80 |    3   (0)|
-------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("E"."DEPTNO"="D"."DEPTNO")
```

4.4.4 与索引有关的 Hint

```
/*+ index(table index) */
/*+ index(table index1 index2 ...) */
/*+ index(table_name) */
```