前言

本文档主要介绍与 SQL 调整有关的内容，内容涉及多个方面：SQL 语句执行的过程、ORACLE 优化器，表之间的关联，如何得到 SQL 执行计划，如何分析执行计划等内容，从而由浅到深的方式了解 SQL 优化的过程，使大家逐步步入 SQL 调整之门，然后你将发现……。

该文档的不当之处，敬请指出，以便进一步改正。请将其发往我的信箱：xu_yu_jin2000@sina.com。

如果引用本文的内容，请著名出处！

作者：徐玉金
MSN：sunny_xyj@hotmial.com
Email: xu_yu_jin2000@sina.com
日期：2005.12.12
活跃于:www.cnoug.org　SunnyXu

目录

第 1 章 性能调整综述

Oracle 数据库是高度可调的数据库产品。本章描述调整的过程和那些人员应与 Oracle 服务器的调整有关，以及与调整相关联的操作系统硬件和软件。本章包括以下方面:

- 谁来调整系统?
- 什么时候调整?
- 建立有效调整的目标
- 在设计和开发时的调整
- 调整产品系统
- 监控产品系统

谁来调整系统:

为了有效地调整系统，若干类人员必须交换信息并牵涉到系统调整中，例如:

- 应用设计人员必须传达应用系统的设计，使得每个人都清楚应用中的数据流动.
- 应用开发人员必须传达他们选择的实现策略，使得语句调整的过程中能快速、容易地识别有问题的应用模块和可疑的 SQL 语句.
- 数据库管理人员必须仔细地监控系统活动并提供它们的资料，使得异常的系统性能可被快速得识别和纠正.
- 硬件/软件管理人员必须传达系统的硬件、软件配置并提供它们的资料，使得相关人员能有效地设计和管理系统。

简而言之，与系统涉及的每个人都在调整过程中起某些作用，当上面提及的那些人员传达了系统的特性并提供了它们的资料，调整就能相对的容易和更快一些。

不幸的是，事实上的结果是：数据库管理员对调整负有全部或主要的责任。但是，数据库管理员很少有合适的系统方面的资料，而且，在很多情况下，数据库管理员往往是在实施阶段才介入数据库，这就给调整工作带来许多负面的影响，因为在设计阶段的缺陷是不能通过 DBA 的调整而得以解决，而设计阶段的缺陷往往对数据库性能造成极大的影响。

其实，在真正成熟的开发环境下，开发人员作为纯代码编写人员时，对性能的影响最小，此时大部分的工作应由应用设计人员完成，而且数据库管理员往往在前期的需求管理阶段就介入，为设计人员提供必要的技术支持。

调整并不是数据库管理员的专利，相反大部分应该是设计人员和开发人员的工作，这就需要设计人员和开发人员具体必要的数据库知识，这样才能组成一个高效的团队，然而事实上往往并非如此。

什么时候作调整？

多数人认为当用户感觉性能差时才进行调整，这对调整过程中使用某些最有效的调整策略来说往往是太迟了。此时，如果你不愿意重新设计应用的话，你只能通过重新分配内存(调整 SGA)和调整 I/O 的办法或多或少地提高性能。Oracle 提供了许多特性，这些特性只有应用到正确地设计的系统中时才能够很大地提高性能。

应用设计人员需要在设计阶段设置应用的性能期望值。然后在设计和开发期间，应用设计人员应考虑哪些 Oracle 特性可以对系统有好处，并使用这些特性。

通过良好的系统设计，你就可以在应用的生命周期中消除性能调整的代价和挫折。图 1-1 图 1-2 说明在应用的生命周期中调整的相对代价和收益，正如你见到的，最有效的调整

时间是在设计阶段。在设计期间的调整能以最低的代价给你最大的收益。

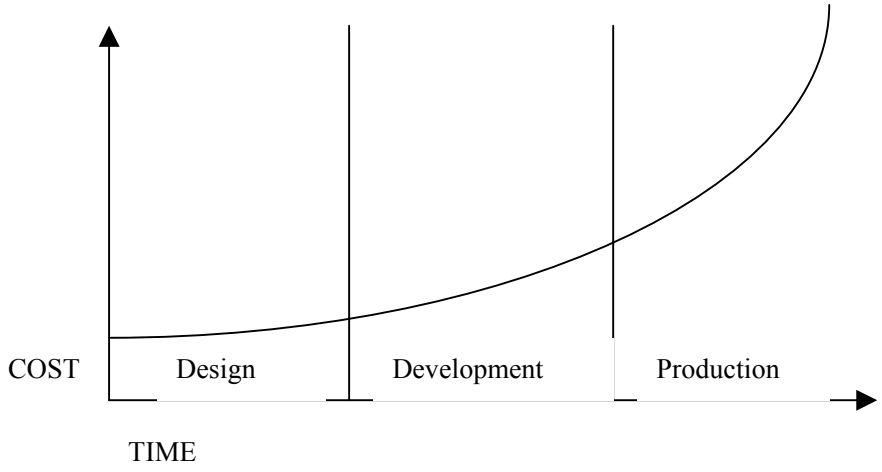


图 1-1　在应用生命周期中调整的代价

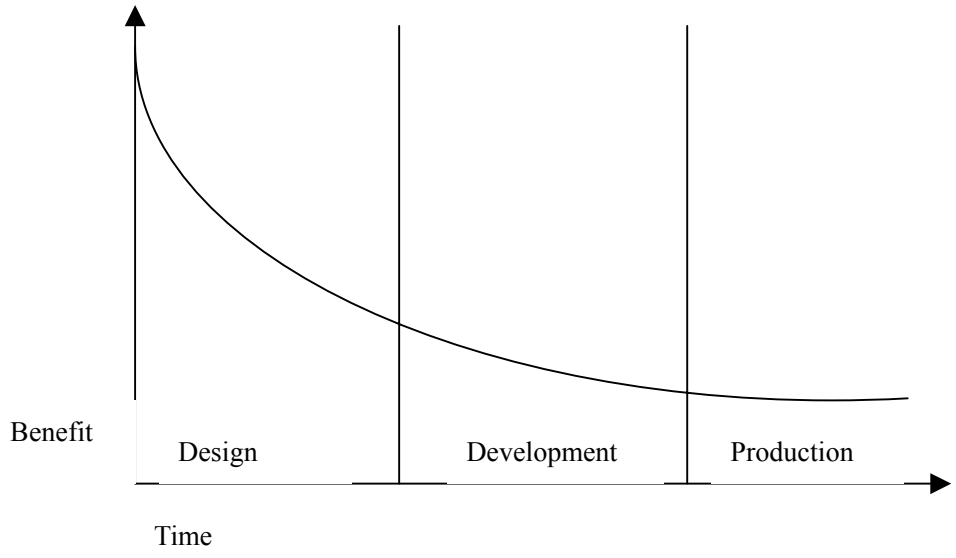


图 1-2　在应用生命周期中调整的收益

当然，即使在设计很好的系统中，也可能有性能降低。但这些性能降低应该是可控的和可以预见的。

调整目标

不管你正在设计或维护系统，你应该建立专门的性能目标，它使你知道何时要作调整。如果你试图胡乱地改动初始化参数或 SQl 语句，你可能会浪费调整系统的时间，而且无什么大的收益。调整你的系统的最有效方法如下：

- 当设计系统时考虑性能
- 调整操作系统的硬件和软件
- 识别性能瓶颈

- 确定问题的原因
- 采取纠正的动作

当你设计系统时，制定专门的目标；例如，响应时间小于 3 秒。当应用不能满足此目标时，识别造成变慢的瓶颈（例如，I/O 竞争），确定原因，采取纠正动作。在开发期间，你应测试应用研究，确定在采取应用之前是否满足设计的性能目标。

当你正在维护生产库系统时，有多种快速有效的方法来识别性能瓶颈。

不管怎样，调整通常是一系列开销。一旦你已确定了瓶颈，你可能要牺牲一些其它方面的指标来达到所要的结果。例如，如果 I/O 有问题，你可能需要更多内存或磁盘。如果不可能买，你可能要限制系统的并发性，来获取所需的性能。然而，如果你已经明确地定义了性能的目标，那用什么来交换高性能的决策就变的很容易的，因为你已经确定了哪些方面是最重要的，如过我的目标为高性能，可能牺牲一些空间资源。

随着应用的越来越庞大，硬件性能的提高，全面的调整应用逐渐变成代价高昂的行为，在这样情况下，要取得最大的投入/效率之比，较好的办法是调整应用的关键部分，使其达到比较高的性能，这样从总体上来说，整个系统的性能也是比较高的。这也就是有名的 20/80 原则，调整应用的 20%(关键部分)，能解决 80%的问题。

在设计和开发系统时作调整

良好设计的系统可以防止在应用生命周期中产生性能问题。系统设计人员和应用开发人员必须了解 Oracle 的查询处理机制以便写出高效的 SQL 语句。“第 2 章 有效的应用设计”讨论了你的系统中各种可用的配置，以及每种配置更适合哪种类型的应用。“第 5 章 优化器”讨论了 Oracle 的查询优化器，以及如何写语句以获取最快的结果。

当设计你的系统时，使用下列优化性能的准则：

- 消除客户机 / 服务器应用中不必要的网络传输。-- 使用存储过程。
- 使用适合你系统的相应 Oracle 服务器选件(例如,并行查询或分布式数据库)。
- 除非你的应用有特殊的需要，否则使用缺省的 Oracle 锁。
- 利用数据库记住应用模块，以便你能以每个模块为基础来追踪性能。
- 选择你的数据块的最佳大小。 -- 原则上来说大一些的性能较好。
- 分布你的数据，使得一个节点使用的数据本地存贮在该节点中。

调整产品系统

本节描述对应用系统快速、容易地找出性能瓶颈，并决定纠正动作的方法。这种方法依赖于对 Oracle 服务器体系结构和特性的了解程度。在试图调整你的系统前，你应熟悉 Oracle 调整的内容。

为调整你已有的系统，遵从下列步骤：

- 调整操作系统的硬件和软件
- 通过查询 V $SESSION_WAIT 视图，识别性能的瓶颈，这个动态性能视图列出了造成会话(session)等待的事件。
- 通过分析 V $SESSION_WAIT 中的数据，决定瓶颈的原因。
- 纠正存在的问题。

监控应用系统

这主要是通过监控 oracle 的动态视图来完成。

各种有用的动态视图：如 v$session_wait, v$session_event 等。

第 2 章 有效的应用设计

我们通常将最常用的应用分为 2 种类型：联机事务处理类型(OLTP)，决策支持系统(DSS)。

联机事务处理(OLTP)

该类型的应用是高吞吐量，插入、更新、删除操作比较多的系统，这些系统以不断增长的大容量数据为特征，它们提供给成百用户同时存取，典型的 OLTP 系统是订票系统，银行的业务系统，订单系统。OTLP 的主要目标是可用性、速度、并发性和可恢复性。

当设计这类系统时，必须确保大量的并发用户不能干扰系统的性能。还需要避免使用过量的索引与 cluster 表，因为这些结构会使插入和更新操作变慢。

决策支持(DSS)

该类型的应用将大量信息进行提取形成报告，协助决策者作出正确的判断。典型的情况是：决策支持系统将 OLTP 应用收集的大量数据进行查询。典型的应用为客户行为分析系统(超市，保险等)。

决策支持的关键目标是速度、精确性和可用性。

该种类型的设计往往与 OLTP 设计的理念背道而驰，一般建议使用数据冗余、大量索引、cluster table、并行查询等。

近年来，该类型的应用逐渐与 OLAP、数据仓库紧密的联系在一起，形成的一个新的应用方向。

第 3 章　SQL 语句处理的过程

在调整之前我们需要了解一些背景知识，只有知道这些背景知识，我们才能更好的去调整 sql 语句。

本节介绍了 SQL 语句处理的基本过程，主要包括：

- 查询语句处理
- DML 语句处理(insert, update, delete)
- DDL 语句处理(create .. , drop .. , alter .. ,)
- 事务控制(commit, rollback)

SQL 语句的执行过程(SQL Statement Execution)

图 3-1 概要的列出了处理和运行一个 sql 语句的需要各个重要阶段。在某些情况下，Oracle 运行 sql 的过程可能与下面列出的各个阶段的顺序有所不同。如 DEFINE 阶段可能在 FETCH 阶段之前，这主要依赖你如何书写代码。

对许多 oracle 的工具来说，其中某些阶段会自动执行。绝大多数用户不需要关心各个阶段的细节问题，然而，知道执行的各个阶段还是有必要的，这会帮助你写出更高效的 SQL 语句来，而且还可以让你猜测出性能差的 SQL 语句主要是由于哪一个阶段造成的，然后我们针对这个具体的阶段，找出解决的办法。

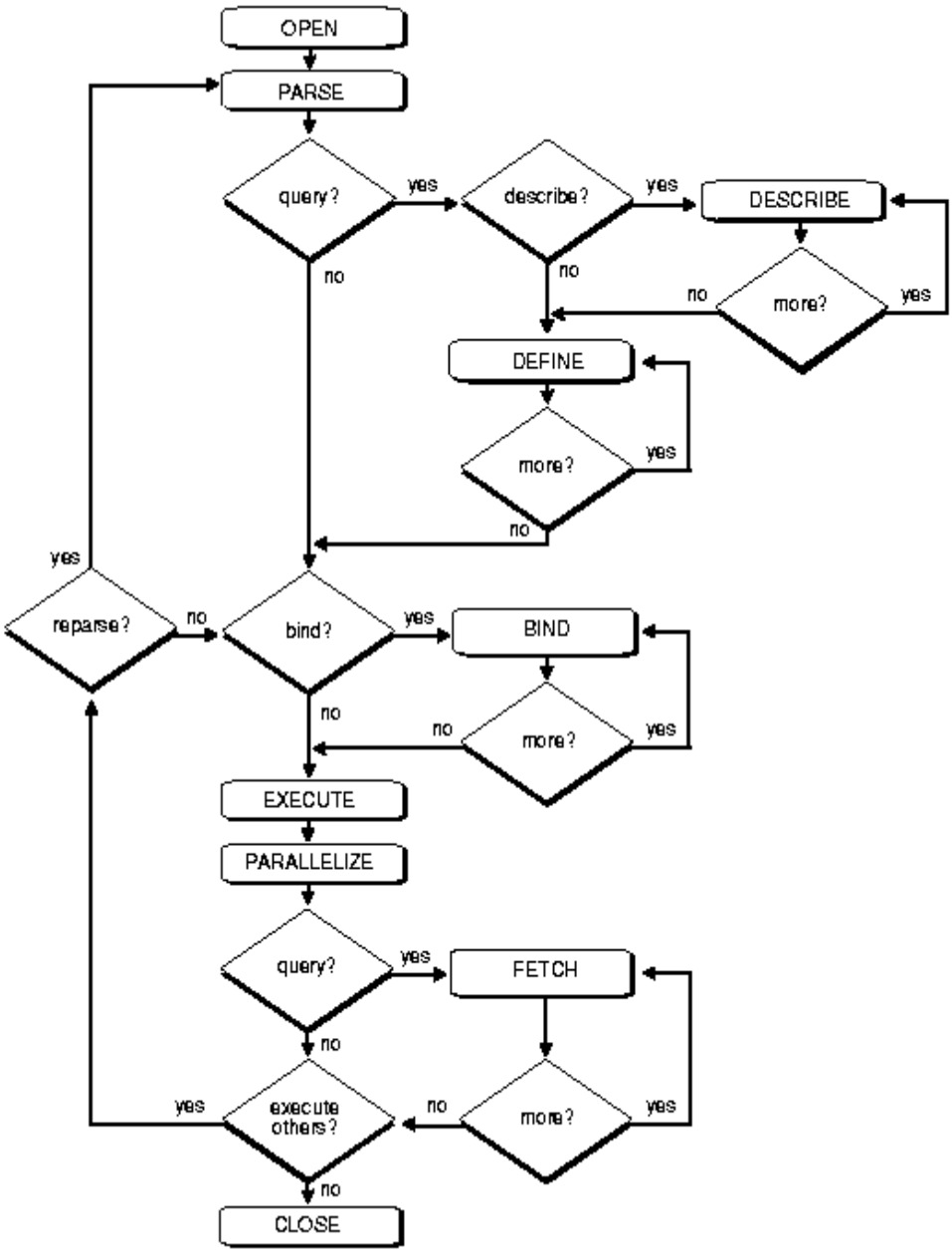


图 3-1　SQL 语句处理的各个阶段

DML 语句的处理

本节给出一个例子来说明在 DML 语句处理的各个阶段到底发生了什么事情。

假设你使用 Pro*C 程序来为指定部门的所有职员增加工资。程序已经连到正确的用户，你可以在你的程序中嵌入如下的 SQL 语句：

```
EXEC SQL UPDATE employees
                SET salary = 1.10 * salary
                WHERE department_id = :var_department_id;
```

var_department_id 是程序变量，里面包含部门号，我们要修改该部门的职员的工资。当这个 SQL 语句执行时，使用该变量的值。

每种类型的语句都需要如下阶段：

- 第 1 步: Create a Cursor　　创建游标
- 第 2 步: Parse the Statement　分析语句
- 第 5 步: Bind Any Variables　　绑定变量
- 第 7 步: Run the Statement　　运行语句
- 第 9 步: Close the Cursor　　关闭游标

如果使用了并行功能，还会包含下面这个阶段：

- 第 6 步: Parallelize the Statement　　并行执行语句

如果是查询语句，则需要以下几个额外的步骤，如图 3 所示：

- 第 3 步: Describe Results of a Query　　描述查询的结果集
- 第 4 步: Define Output of a Query　　定义查询的输出数据
- 第 8 步: Fetch Rows of a Query　　取查询出来的行

下面具体说一下每一步中都发生了什么事情：.

第 1 步: 创建游标(Create a Cursor)

由程序接口调用创建一个游标（cursor）。任何 SQL 语句都会创建它，特别在运行 DML 语句时，都是自动创建游标的，不需要开发人员干预。多数应用中，游标的创建是自动的。然而，在预编译程序(pro*c)中游标的创建，可能是隐含的，也可能显式的创建。在存储过程中也是这样的。

第 2 步:分析语句(Parse the Statement)

在语法分析期间，SQL 语句从用户进程传送到 Oracle，SQL 语句经语法分析后，SQL 语句本身与分析的信息都被装入到共享 SQL 区。在该阶段中，可以解决许多类型的错误。

语法分析分别执行下列操作：

- 翻译 SQL 语句，验证它是合法的语句，即书写正确
- 实现数据字典的查找，以验证是否符合表和列的定义
- 在所要求的对象上获取语法分析锁，使得在语句的语法分析过程中不改变这些对象的定义
- 验证为存取所涉及的模式对象所需的权限是否满足

- 决定此语句最佳的执行计划
- 将它装入共享 SQL 区
- 对分布的语句来说，把语句的全部或部分路由到包含所涉及数据的远程节点

以上任何一步出现错误，都将导致语句报错，中止执行。

只有在共享池中不存在等价 SQL 语句的情况下，才对 SQL 语句作语法分析。在这种情况下，数据库内核重新为该语句分配新的共享 SQL 区，并对语句进行语法分析。进行语法分析需要耗费较多的资源，所以要尽量避免进行语法分析，这是优化的技巧之一。

语法分析阶段包含了不管此语句将执行多少次，而只需分析一次的处理要求。Oracle 只对每个 SQL 语句翻译一次，在以后再次执行该语句时，只要该语句还在共享 SQL 区中，就可以避免对该语句重新进行语法分析，也就是此时可以直接使用其对应的执行计划对数据进行存取。这主要是通过绑定变量(bind variable)实现的，也就是我们常说的共享 SQL，后面会给出共享 SQL 的概念。

虽然语法分析验证了 SQL 语句的正确性，但语法分析只能识别在 SQL 语句执行之前所能发现的错误(如书写错误、权限不足等)。因此，有些错误通过语法分析是抓不到的。例如，在数据转换中的错误或在数据中的错（如企图在主键中插入重复的值）以及死锁等均是只有在语句执行阶段期间才能遇到和报告的错误或情况。

查询语句的处理

查询与其它类型的 SQL 语句不同，因为在成功执行后作为结果将返回数据。其它语句只是简单地返回成功或失败，而查询则能返回一行或许多行数据。查询的结果均采用表格形式，结果行被一次一行或者批量地被检索出来。从这里我们可以得知批量的 fetch 数据可以降低网络开销，所以批量的 fetch 也是优化的技巧之一。

有些问题只与查询处理相关，查询不仅仅指 SELECT 语句，同样也包括在其它 SQL 语句中的隐含查询。例如，下面的每个语句都需要把查询作为它执行的一部分：

INSERT INTO table SELECT...

UPDATE table SET x = y WHERE...

DELETE FROM table WHERE...

CREATE table AS SELECT...

具体来说，查询

- 要求读一致性
- 可能使用回滚段作中间处理
- 可能要求 SQL 语句处理描述、定义和取数据阶段

第 3 步: 描述查询结果(Describe Results of a Query)

描述阶段只有在查询结果的各个列是未知时才需要；例如，当查询由用户交互地输入需要输出的列名。在这种情况要用描述阶段来决定查询结果的特征（数据类型，长度和名字）。

第 4 步: 定义查询的输出数据(Define Output of a Query)

在查询的定义阶段，你指定与查询出的列值对应的接收变量的位置、大小和数据类型，这样我们通过接收变量就可以得到查询结果。如果必要的话，Oracle 会自动实现数据类型的转换。这是将接收变量的类型与对应的列类型相比较决定的。

第 5 步: 绑定变量(Bind Any Variables)

此时，Oracle 知道了 SQL 语句的意思，但仍没有足够的信息用于执行该语句。Oracle 需要得到在语句中列出的所有变量的值。在该例中，Oracle 需要得到对 department_id 列进行限定的值。得到这个值的过程就叫绑定变量(**binding variables**)

此过程称之为将变量值捆绑进来。程序必须指出可以找到该数值的变量名（该变量被称为捆绑变量，变量名实质上是一个内存地址，相当于指针）。应用的最终用户可能并没有发觉他们正在指定捆绑变量，因为 Oracle 的程序可能只是简单地指示他们输入新的值，其实这一切都在程序中自动做了。

因为你指定了变量名，在你再次执行之前无须重新捆绑变量。你可以改变绑定变量的值，而 Oracle 在每次执行时，仅仅使用内存地址来查找此值。

如果 Oracle 需要实现自动数据类型转换的话（除非它们是隐含的或缺省的），你还必须对每个值指定数据类型和长度。关于这些信息可以参考 oracle 的相关文档，如 Oracle Call Interface Programmer's Guide

第 6 步: 并行执行语句(Parallelize the Statement)

ORACLE 可以在 SELECTs, INSERTs, UPDATEs, MERGEs, DELETEs 语句中执行相应并行查询操作，对于某些 DDL 操作，如创建索引、用子查询创建表、在分区表上的操作，也可以执行并行操作。并行化可以导致多个服务器进程(oracle server processes)为同一个 SQL 语句工作，使该 SQL 语句可以快速完成，但是会耗费更多的资源，所以除非很有必要，否则不要使用并行查询。

第 7 步: 执行语句(Run the Statement)

到了现在这个时候，Oracle 拥有所有需要的信息与资源，因此可以真正运行 SQL 语句了。如果该语句为 SELECT 查询或 INSERT 语句，则不需要锁定任何行，因为没有数据需要被改变。然而，如果语句为 UPDATE 或 DELETE 语句，则该语句影响的所有行都被锁定，防止该用户提交或回滚之前，别的用户对这些数据进行修改。这保证了数据的一致性。

对于某些语句，你可以指定执行的次数，这称为批处理(array processing)。指定执行 N 次，则绑定变量与定义变量被定义为大小为 N 的数组的开始位置，这种方法可以减少网络开销，也是优化的技巧之一。

第 8 步: 取出查询的行(Fetch Rows of a Query)

在 fetch 阶段，行数据被取出来，每个后续的存取操作检索结果集中的下一行数据，直到最后一行被取出来。上面提到过，批量的 fetch 是优化的技巧之一。

第 9 步: 关闭游标(Close the Cursor)

SQL 语句处理的最后一个阶段就是关闭游标

DDL 语句的处理(DDL Statement Processing)

DDL 语句的执行不同与 DML 语句和查询语句的执行，这是因为 DDL 语句执行成功后需要对数据字典数据进行修改。对于 DDL 语句，语句的分析阶段实际上包括分析、查找数据字典信息和执行。

事务管理语句、会话管理语句、系统管理语句只有分析与执行阶段，为了重新执行该语句，会重新分析与执行该语句。

事务控制(Control of Transactions)

一般来说，只有使用 ORACLE 编程接口的应用设计人员才关心操作的类型，并把相关的操作组织在一起，形成一个事务。一般来说，我门必须定义事务，这样在一个逻辑单元中的所有工作可以同时被提交或回滚，保证了数据的一致性。一个事务应该由逻辑单元中的所有必须部分组成，不应该多一个，也不应该少一个。

- 在事务开始和结束的这段时间内，所有被引用表中的数据都应该在一致的状态(或可以被回溯到一致的状态)
- 事务应该只包含可以对数据进行一致更改(one consistent change to the data)的 SQL 语句

例如，在两个帐号之间的转帐(这是一个事务或逻辑工作单元)，应该包含从一个帐号中借钱(由一个 SQL 完成)，然后将借的钱存入另一个帐号(由另一个 SQL 完成)。这 2 个操作作为一个逻辑单元，应该同时成功或同时失败。其它不相关的操作，如向一个帐户中存钱，不应该包含在这个转帐事务中。

在设计应用时，除了需要决定哪种类型的操作组成一个事务外，还需要决定使用 BEGIN_DISCRETE_TRANSACTIO 存储过程是否对提高小的、非分布式的事务的性能有作用。

第 4 章 ORACLE 的优化器

优化器有时也被称为查询优化器，这是因为查询是影响数据库性能最主要的部分，不要以为只有 SELECT 语句是查询。实际上，带有任何 WHERE 条件的 DML(INSERT、UPDATE、DELETE)语句中都包含查询要求，在后面的文章中，当说到查询时，不一定只是指 SELECT 语句，也有可能指 DML 语句中的查询部分。优化器是所有关系数据库引擎中的最神秘、最富挑战性的部件之一，从性能的角度看也是最重要的部分，它性能的高低直接关系到数据库性能的好坏。

我们知道，SQL 语句同其它语言(如 C 语言)的语句不一样，它是非过程化(non-procedural)的语句，即当你要取数据时，不需要告诉数据库通过何种途径去取数据，如到底是通过索引取数据，还是应该将表中的每行数据都取出来，然后再通过一一比较的方式取数据(即全表扫描)，这是由数据库的优化器决定的，这就是非过程化的含义，也就是说，如何取数据是由优化器决定，而不是应用开发者通过编程决定。在处理 SQL 的 SELECT、UPDATE、INSERT 或 DELETE 语句时，Oracle 必须访问语句所涉及的数据，Oracle 的优化器部分用来决定访问数据的有效路径，使得语句执行所需的 I/O 和处理时间最小。

为了实现一个查询，内核必须为每个查询定制一个查询策略，或为取出符合条件的数据生成一个执行计划(execution plan)。典型的，对于同一个查询，可能有几个执行计划都符合要求，都能得到符合条件的数据。例如，参与连接的表可以有多种不同的连接方法，这取决于连接条件和优化器采用的连接方法。为了在多个执行计划中选择最优的执行计划，优化器必须使用一些实际的指标来衡量每个执行计划使用的资源(I/0 次数、CPU 等)，这些资源也就是我们所说的代价(cost)。如果一个执行计划使用的资源多，我们就说使用执行计划的代价大。以执行计划的代价大小作为衡量标准，优化器选择代价最小的执行计划作为真正执行该查询的执行计划，并抛弃其它的执行计划。

在 ORACLE 的发展过程中，一共开发过 2 种类型的优化器：基于规则的优化器和基于代价的优化器。这 2 种优化器的不同之处关键在于：取得代价的方法与衡量代价的大小不同。现对每种优化器做一下简单的介绍：

基于规则的优化器 -- Rule Based (Heuristic) Optimization(简称 RBO)：

在 ORACLE7 之前，主要是使用基于规则的优化器。ORACLE 在基于规则的优化器中采用启发式的方法(Heuristic Approach)或规则(Rules)来生成执行计划。例如，如果一个查询的 where 条件(where clause)包含一个谓词(predicate，其实就是一个判断条件，如”=”, “>”, ”<”等)，而且该谓词上引用的列上有有效索引，那么优化器将使用索引访问这个表，而不考虑其它因素，如表中数据的多少、表中数据的易变性、索引的可选择性等。此时数据库中没有关于表与索引数据的统计性描述，如表中有多上行，每行的可选择性等。优化器也不考虑实例参数，如 multi block i/o、可用排序内存的大小等，所以优化器有时就选择了次优化的计划作为真正的执行计划，导致系统性能不高。

如，对于

```
select * from emp where deptno = 10;
```

这个查询来说，如果是使用基于规则的优化器，而且 deptno 列上有有效的索引，则会通过 deptno 列上的索引来访问 emp 表。在绝大多数情况下，这是比较高效的，但是在一些特殊情况下，使用索引访问也有比较低效的时候，现举例说明：

1) emp 表比较小，该表的数据只存放在几个数据块中。此时使用全表扫描比使用索引访问 emp 表反而要好。因为表比较小，极有可能数据全在内存中，所以此时做全表扫描是最快的。而如果使用索引扫描，需要先从索引中找到符合条件记录的 rowid，然后再一一根据这些 rowid 从 emp 中将数据取出来，在这种条件下，效率就会比全表扫描的效率要差一些。

2) emp 表比较大时，而且 deptno = 10 条件能查询出表中大部分的数据如(50%)。如该表共有 4000 万行数据，共放在有 500000 个数据块中，每个数据块为 8k，则该表共有约 4G，则这么多的数据不可能全放在内存中，绝大多数需要放在硬盘上。此时如果该查询通过索引查询，则是你梦魇的开始。db_file_multiblock_read_count 参数的值 200。如果采用全表扫描，则需要 500000/db_file_multiblock_read_count=500000/200=2500 次 I/O。但是如果采用索引扫描，假设 deptno 列上的索引都已经 cache 到内存中，所以可以将访问索引的开销忽略不计。因为要读出 4000 万 x 50% = 2000 万数据，假设在读这 2000 万数据时，有 99.9%的命中率，则还是需要 20000 次 I/O,比上面的全表扫描需要的 2500 次多多了，所以在这种情况下，用索引扫描反而性能会差很多。在这样的情况下，用全表扫描的时间是固定的，但是用索引扫描的时间会随着选出数据的增多使查询时间相应的延长。

上面是枯燥的假设数据，现在以具体的实例给予验证：

环境: oracle 817 + linux + 阵列柜，表 SWD_BILLDETAIL 有 3200 多万数据；

表的 id 列、cn 列上都有索引

经查看执行计划，发现执行 select count(id) from SWD_BILLDETAIL;使用全表扫描，执行完用了大约 1.50 分钟(4 次执行取平均,每次分别为 1.45 1.51 2.00 1.46)。而执行 select count(id) from SWD_BILLDETAIL where cn <'6';却用了 2 个小时还没有执行完，经分析该语句使用了 cn 列上的索引，然后利用查询出的 rowid 再从表中查询数据。我为什么不使用 select count(cn) from SWD_BILLDETAIL where cn <'6';呢？后面在分析执行路径的索引扫描时时会给出说明。

下面就是基于规则的优化器使用的执行路径与各个路径对应的等级：

RBO Path 1: Single Row by Rowid(等级最高)
RBO Path 2: Single Row by Cluster Join
RBO Path 3: Single Row by Hash Cluster Key with Unique or Primary Key
RBO Path 4: Single Row by Unique or Primary Key
RBO Path 5: Clustered Join
RBO Path 6: Hash Cluster Key
RBO Path 7: Indexed Cluster Key
RBO Path 8: Composite Index
RBO Path 9: Single-Column Indexes
RBO Path 10: Bounded Range Search on Indexed Columns
RBO Path 11: Unbounded Range Search on Indexed Columns
RBO Path 12: Sort Merge Join
RBO Path 13: MAX or MIN of Indexed Column
RBO Path 14: ORDER BY on Indexed Column
RBO Path 15: Full Table Scan(等级最低)

上面的执行路径中，RBO 认为越往下执行的代价越大，即等级越低。在 RBO 生成执行计划时，如果它发现有等级高的执行路径可用，则肯定会使用等级高的路径，而不管任何其它影响性能的元素，即 RBO 通过上面的路径的等级决定执行路径的代价，执行路径的等级越高，则使用该执行路径的代价越小。如上面 2 个例子所述，如果使用 RBO，则肯定使

用索引访问表，也就是选择了比较差的执行计划，这样会给数据库性能带来很大的负面影响。为了解决这个问题，从 ORACLE 7 开始 oracle 引入了基于代价的优化器，下面给出了介绍。

基于代价的优化器 -- Cost Based Optimization(简称 CBO)

Oracle 把一个代价引擎(Cost Engine)集成到数据库内核中，用来估计每个执行计划需要的代价，该代价将每个执行计划所耗费的资源进行量化，从而 CBO 可以根据这个代价选择出最优的执行计划。一个查询耗费的资源可以被分成 3 个基本组成部分：I/O 代价、CPU 代价、network 代价。I/O 代价是将数据从磁盘读入内存所需的代价。访问数据包括将数据文件中数据块的内容读入到 SGA 的数据高速缓存中，在一般情况下，该代价是处理一个查询所需要的最主要代价，所以我们在优化时，一个基本原则就是降低查询所产生的 I/O 总次数。CPU 代价是处理在内存中数据所需要的代价，如一旦数据被读入内存，则我们在识别出我们需要的数据后，在这些数据上执行排序(sort)或连接(join)操作，这需要耗费 CPU 资源。

对于需要访问跨节点(即通常说的服务器)数据库上数据的查询来说，存在 network 代价，用来量化传输操作耗费的资源。查询远程表的查询或执行分布式连接的查询会在 network 代价方面花费比较大。

在使用 CBO 时，需要有表和索引的统计数据(分析数据)作为基础数据，有了这些数据，CBO 才能为各个执行计划计算出相对准确的代价，从而使 CBO 选择最佳的执行计划。所以定期的对表、索引进行分析是绝对必要的，这样才能使统计数据反映数据库中的真实情况。否则就会使 CBO 选择较差的执行计划，影响数据库的性能。分析操作不必做的太频繁，一般来说，每星期一次就足够了。切记如果想使用 CBO，则必须定期对表和索引进行分析。

对于分析用的命令，随着数据库版本的升级，用的命令也发生了变换，在 oracle 8i 以前，主要是用 ANALYZE 命令。在 ORACLE 8I 以后，又引入了 DBMS_STATS 存储包来进行分析。幸运的是从 ORACLE 10G 以后，分析工作变成自动的了，这减轻的 DBA 的负担，不过在一些特殊情况下，还需要一些手工分析。

如果采用了 CBO 优化器，而没有对表和索引进行分析，没有统计数据，则 ORACLE 使用缺省的统计数据(至少在 ORACLE 9I 中是这样)，这可以从 oracle 的文档上找到。使用的缺省值肯定与系统的实际统计值不一致，这可能会导致优化器选择错误的执行计划，影响数据库的性能。

要注意的是：虽然 CBO 的功能随着 ORACLE 新版本的推出，功能越来越强，但它不是能包治百病的神药，否则就不再需要 DBA 了，那我就惨了!!! 实际上任何一个语句，随着硬件环境与应用数据的不同，该语句的执行计划可能需要随之发生变化，这样才能取得最好的性能。所以有时候不在具体的环境下而进行 SQL 性能调整是徒劳的。

在 ORACLE8I 推出的时候，ORACLE 极力建议大家使用 CBO，说 CBO 有种种好处，但是在那是 ORACLE 开发的应用系统还是使用基于规则的优化器，从这件事上我们可以得出这样的结论：1) 如果团队的数据库水平很高而且都熟悉应用数据的特点，RBO 也可以取得很好的性能。2）CBO 不是很稳定，但是一个比较有前途的优化器，Oracle 极力建议大家用是为了让大家尽快发现它的 BUG，以便进一步改善，但是 ORACLE 为了对自己开发的应用系统负责，他们还是使用了比较熟悉而且成熟的 RBO。从这个事情上给我们的启发就是：我们在以后的开发中，应该尽量采用我们熟悉并且成熟的技术，而不要一味的采用新技术，一味采用新技术并不一定能开发出好的产品。幸运的是从 ORACLE 10G 后，CBO 已经足够的强大与智能，大家可以放心的使用该技术，因为 ORACLE 10G 后，Oracle 自己开发的应用系统也使用 CBO 优化器了。而且 ORACLE 规定，从 ORACLE 10G 开始，开始废弃 RBO 优化器。这句话并不是指在 ORACLE 10G 中不能使用 RBO，而是从 ORACLE

10G 开始开始，不再为 RBO 的 BUG 提供修补服务。

在上面的第 2 个例子中，如果采用 CBO 优化器，它就会考虑 emp 表的行数，deptno 列的统计数据，发现对该列做查询会查询出过多的数据，并且考虑 db_file_multiblock_read_count 参数的设置，发现用全表扫描的代价比用索引扫描的代价要小，从而使用全表扫描从而取得良好的执行性能。

判断当前数据库使用何种优化器：

主要是由 optimizer_mode 初始化参数决定的。该参数可能的取值为：first_rows_[1 | 10 | 100 | 1000] | first_rows | all_rows | choose | rule。具体解释如下：

RULE 为使用 RBO 优化器。

CHOOSE 则是根据实际情况，如果数据字典中包含被引用的表的统计数据，即引用的对象已经被分析，则就使用 CBO 优化器，否则为 RBO 优化器。

ALL_ROWS 为 CBO 优化器使用的第一种具体的优化方法，是以数据的吞吐量为主要目标，以便可以使用最少的资源完成语句。

FIRST_ROWS 为优化器使用的第二种具体的优化方法，是以数据的响应时间为主要目标，以便快速查询出开始的几行数据。

FIRST_ROWS_[1 | 10 | 100 | 1000] 为优化器使用的第三种具体的优化方法，让优化器选择一个能够把响应时间减到最小的查询执行计划，以迅速产生查询结果的前 n 行。该参数为 ORACLE 9I 新引入的。

从 ORACLE V7 以来，optimizer_mode 参数的缺省设置应是"choose"，即如果对已分析的表查询的话选择 CBO，否则选择 RBO。在此种设置中，如果采用了 CBO，则缺省为 CBO 中的 all_rows 模式。

注意：即使指定数据库使用 RBO 优化器，但有时 ORACLE 数据库还是会采用 CBO 优化器，这并不是 ORACLE 的 BUG，主要是由于从 ORACLE 8I 后引入的许多新特性都必须在 CBO 下才能使用，而你的 SQL 语句可能正好使用了这些新特性，此时数据库会自动转为使用 CBO 优化器执行这些语句。

什么是优化

优化是选择最有效的执行计划来执行 SQL 语句的过程，这是在处理任何数据的语句（SELECT,INSERT,UPDATE 或 DELETE）中的一个重要步骤。对 Oracle 来说，执行这样的语句有许多不同的方法，譬如说，将随着以什么顺序访问哪些表或索引的不同而不同。所使用的执行计划可以决定语句能执行得有多快。Oracle 中称之为优化器（Optimizer）的组件用来选择这种它认为最有效的执行计划。

由于一系列因素都会会影响语句的执行，优化器综合权衡各个因素，在众多的执行计划中选择认为是最佳的执行计划。然而，应用设计人员通常比优化器更知道关于特定应用的数据特点。无论优化器多么智能，在某些情况下开发人员能选择出比优化器选择的最优执行计划还要好的执行计划。这是需要人工干预数据库优化的主要原因。事实表明，在某些情况下，确实需要 DBA 对某些语句进行手工优化。

注：从 Oracle 的一个版本到另一个版本，优化器可能对同一语句生成不同的执行计划。在将来的 Oracle 版本中，优化器可能会基于它可以用的更好、更理想的信息，作出更优的决策，从而导致为语句产生更优的执行计划。

第 5 章 ORACLE 的执行计划

背景知识：

为了更好的进行下面的内容我们必须了解一些概念性的术语：

共享 sql 语句

为了不重复解析相同的 SQL 语句(因为解析操作比较费资源，会导致性能下降)，在第一次解析之后，ORACLE 将 SQL 语句及解析后得到的执行计划存放在内存中。这块位于系统全局区域 SGA(system global area)的共享池(shared buffer pool)中的内存可以被所有的数据库用户共享。因此，当你执行一个 SQL 语句(有时被称为一个游标)时，如果该语句和之前的执行过的某一语句完全相同，并且之前执行的该语句与其执行计划仍然在内存中存在，则 ORACLE 就不需要再进行分析，直接得到该语句的执行路径。ORACLE 的这个功能大大地提高了 SQL 的执行性能并大大节省了内存的使用。使用这个功能的关键是将执行过的语句尽可能放到内存中，所以这要求有大的共享池(通过设置 shared buffer pool 参数值)和尽可能的使用绑定变量的方法执行 SQL 语句。

当你向 ORACLE 提交一个 SQL 语句，ORACLE 会首先在共享内存中查找是否有相同的语句。这里需要注明的是，ORACLE 对两者采取的是一种严格匹配，要达成共享，SQL 语句必须完全相同(包括空格,换行等)。

下面是判断 SQL 语句是否与共享内存中某一 SQL 相同的步骤：

1). 对所发出语句的文本串进行 hashed。如果 hash 值与已在共享池中 SQL 语句的 hash 值相同，则进行第 2 步：

2) 将所发出语句的文本串（包括大小写、空白和注释）与在第 1 步中识别的所有已存在的 SQL 语句相比较。

例如：

```
SELECT * FROM emp WHERE empno = 1000;
```

和下列每一个都不同

```
SELECT * from emp WHERE empno = 1000;
SELECT * FROM EMP WHERE empno = 1000;
SELECT * FROM emp WHERE empno = 2000;
```

在上面的语句中列值都是直接 SQL 语句中的，今后我们将这类 sql 成为硬编码 SQL 或字面值 SQL

使用绑定变量的 SQL 语句中必须使用相同的名字的绑定变量(bind variables) ，例如：

a. 该 2 个 sql 语句被认为相同

```
select pin , name from people where pin = :blk1.pin;
select pin , name from people where pin = :blk1.pin;
```

b. 该 2 个 sql 语句被认为不相同

```
select pin , name from people where pin = :blk1.ot_ind;
select pin , name from people where pin = :blk1.ov_ind;
```

今后我们将上面的这类语句称为绑定变量 SQL。

3). 将所发出语句中涉及的对象与第 2 步中识别的已存在语句所涉及对象相比较。
例如:
如用户 user1 与用户 user2 下都有 EMP 表，则
用户 user1 发出的语句：SELECT * FROM EMP; 与
用户 user2 发出的语句：SELECT * FROM EMP; 被认为是不相同的语句，
因为两个语句中引用的 EMP 不是指同一个表。

4). 在 SQL 语句中使用的捆绑变量的捆绑类型必须一致。

如果语句与当前在共享池中的另一个语句是等同的话，Oracle 并不对它进行语法分析。而直接执行该语句，提高了执行效率，因为语法分析比较耗费资源。

注意的是，从oracle 8i开始，新引入了一个CURSOR_SHARING参数，该参数的主要目的就是为了解决在编程过程中已大量使用的硬编码SQL问题。因为在实际开发中，很多程序人员为了提高开发速度，而采用类似下面的开发方法：

```
str_sql string;
int_empno int;
int_empno = 2000;
str_sql = 'SELECT * FROM emp WHERE empno = ' + int_empno;
……
int_empno = 1000;
str_sql = 'SELECT * FROM emp WHERE empno = ' + int_empno;
```

上面的代码实际上使用了硬编码SQL，使我们不能使用共享SQL的功能，结果是数据库效率不高。但是从上面的2个语句来看，产生的硬编码SQL只是列值不同，其它部分都是相同的，如果仅仅因为列值不同而导致这2个语句不能共享是很可惜的，为了解决这个问题，引入了CURSOR_SHARING参数，使这类问题也可以使用共享SQL，从而使这样的开发也可以利用共享SQL功能。听起来不错，ORACLE真为用户着想，使用户在不改变代码的情况下还可以利用共享SQL的功能。真的如此吗？天上不会无缘无故的掉一个馅饼的，ORACLE对该参数的使用做了说明，建议在经过实际测试后再改该参数的值(缺省情况下，该参数的值为EXACT，语句完全一致才使用共享SQL)。因为有可能该变该值后，你的硬编码SQL是可以使用共享SQL了，但数据库的性能反而会下降。 我在实际应用中已经遇到这种情况。所以建议编写需要稳定运行程序的开发人员最好还是一开始就使用绑定变量的SQL。

Rowid 的概念：

rowid 是一个伪列，既然是伪列，那么这个列就不是用户定义，而是系统自己给加上的。对每个表都有一个 rowid 的伪列，但是表中并不物理存储 ROWID 列的值。不过你可以像使用其它列那样使用它，但是不能删除改列，也不能对该列的值进行修改、插入。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，即即使该行产生行迁移，行的 rowid 也不会改变。

为什么使用 ROWID

rowid 对访问一个表中的给定的行提供了最快的访问方法，通过 ROWID 可以直接定位

到相应的数据块上，然后将其读到内存。我们创建一个索引时，该索引不但存储索引列的值，而且也存储索引值所对应的行的 ROWID，这样我们通过索引快速找到相应行的 ROWID 后，通过该 ROWID，就可以迅速将数据查询出来。这也就是我们使用索引查询时，速度比较快的原因。

在 ORACLE8 以前的版本中，ROWID 由 FILE 、BLOCK、ROW NUMBER 构成。随着 oracle8 中对象概念的扩展，ROWID 发生了变化，ROWID 由 OBJECT、FILE、BLOCK、ROW NUMBER 构成。利用 DBMS_ROWID 可以将 rowid 分解成上述的各部分，也可以将上述的各部分组成一个有效的 rowid。

Recursive SQL概念

有时为了执行用户发出的一个 sql 语句，Oracle 必须执行一些额外的语句，我们将这些额外的语句称之为'recursive calls'或'recursive SQL statements'。如当一个 DDL 语句发出后，ORACLE 总是隐含的发出一些 recursive SQL 语句，来修改数据字典信息，以便用户可以成功的执行该 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中。用户不比关心这些 recursive SQL 语句的执行情况，在需要的时候，ORACLE 会自动的在内部执行这些语句。当然 DML 语句与 SELECT 都可能引起 recursive SQL。简单的说，我们可以将触发器视为 recursive SQL。

Row Source(行源)

用在查询中，由上一操作返回的符合条件的行的集合，即可以是表的全部行数据的集合；也可以是表的部分行数据的集合；也可以为对上 2 个 row source 进行连接操作(如 join 连接)后得到的行数据集合。

Predicate(谓词)

一个查询中的 WHERE 限制条件

Driving Table(驱动表)

该表又称为外层表(OUTER TABLE)。这个概念用于嵌套与 HASH 连接中。如果该 row source 返回较多的行数据，则对所有的后续操作有负面影响。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。一般说来，是应用查询的限制条件后，返回较少行源的表作为驱动表，所以如果一个大表在 WHERE 条件有有限制条件(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。在执行计划中，应该为靠上的那个 row source，后面会给出具体说明。在我们后面的描述中，一般将该表称为连接操作的 row source 1。

Probed Table(被探查表)

该表又称为内层表(INNER TABLE)。在我们从驱动表中得到具体一行的数据后，在该表中寻找符合连接条件的行。所以该表应当为大表(实际上应该为返回较大 row source 的表)且相应的列上应该有索引。在我们后面的描述中，一般将该表称为连接操作的 row source 2。

组合索引(concatenated index)

由多个列构成的索引，如 create index idx_emp on emp(col1, col2, col3, ……)，则我

们称 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。当我们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 = ? ”查询就不会使用该索引。所以限制条件中包含先导列时，该限制条件才会使用该组合索引。

可选择性(selectivity)：

比较一下列中唯一键的数量和表中的行数，就可以判断该列的可选择性。如果该列的”唯一键的数量/表中的行数”的比值越接近 1，则该列的可选择性越高，该列就越适合创建索引，同样索引的可选择性也越高。在可选择性高的列上进行查询时，返回的数据就较少，比较适合使用索引查询。

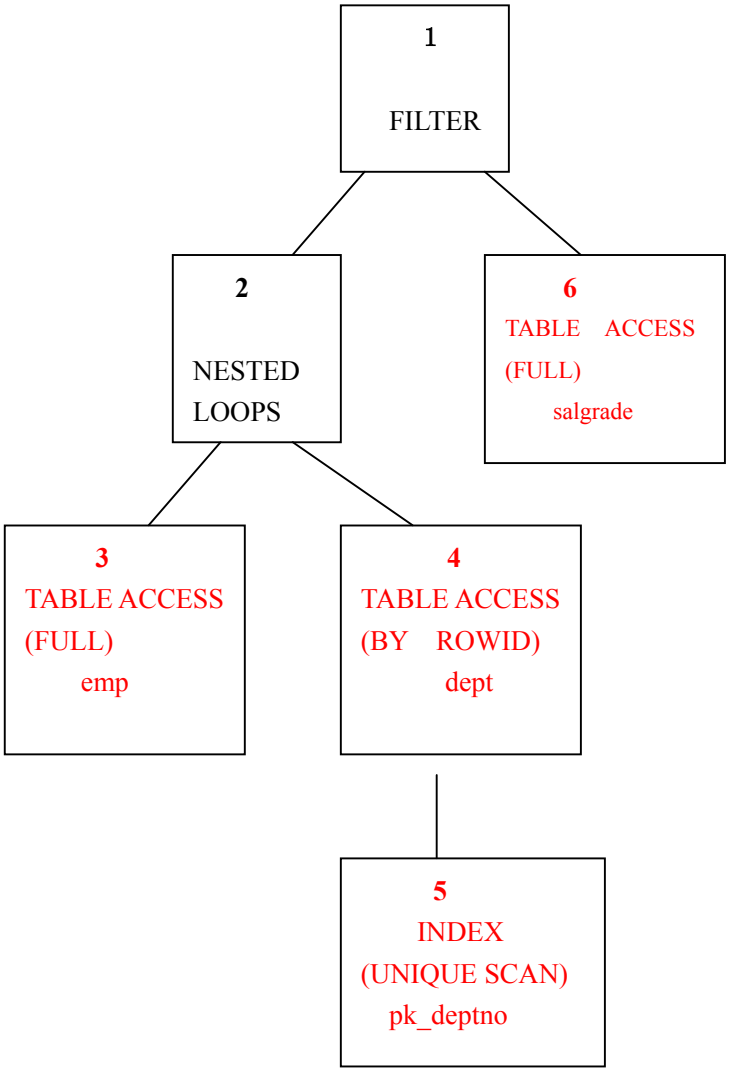
有了这些背景知识后就开始介绍执行计划。为了执行语句，Oracle 可能必须实现许多步骤。这些步骤中的每一步可能是从数据库中物理检索数据行，或者用某种方法准备数据行，供发出语句的用户使用。Oracle 用来执行语句的这些步骤的组合被称之为执行计划。执行计划是 SQL 优化中最为复杂也是最为关键的部分，只有知道了 ORACLE 在内部到底是如何执行该 SQL 语句后，我们才能知道优化器选择的执行计划是否为最优的。执行计划对于 DBA 来说，就象财务报表对于财务人员一样重要。所以我们面临的问题主要是：如何得到执行计划；如何分析执行计划，从而找出影响性能的主要问题。下面先从分析树型执行计划开始介绍，然后介绍如何得到执行计划，再介绍如何分析执行计划。

举例：

这个例子显示关于下面 SQL 语句的执行计划。

```
SELECT ename, job, sal, dname
    FROM emp, dept
 WHERE emp.deptno = derpt.deptno
    AND NOT EXISTS
      ( SELECT *
            FROM salgrade
          WHERE emp.sal BETWEEN losal AND hisal );
```

此语句查询薪水不在任何建议薪水范围内的所有雇员的名字，工作，薪水和部门名。下图 5-1 显示了一个执行计划的图形表示：



执行计划的步骤

执行计划的每一步返回一组行，它们或者为下一步所使用，或者在最后一步时返回给发出 SQL 语句的用户或应用。由每一步返回的一组行叫做行源(row source)。图 5-1 树状图显示了从一步到另一步行数据的流动情况。每步的编号反映了在你观察执行计划时所示步骤的顺序（如何观察执行计划将被简短地说明）。一般来说这并不是每一步被执行的先后顺序。执行计划的每一步或者从数据库中检索行，或者接收来自一个或多个行源的行数据作为输入：

由红色字框指出的步骤从数据库中的数据文件中物理检索数据。这种步骤被称之为存取路径，后面会详细介绍在 Oracle 可以使用的存取路径：

- 第 3 步和第 6 步分别的从 EMP 表和 SALGRADE 表读所有的行。
- 第 5 步在 PK_DEPTNO 索引中查找由步骤 3 返回的每个 DEPTNO 值。它找出与 DEPT 表中相关联的那些行的 ROWID。
- 第 4 步从 DEPT 表中检索出 ROWID 为第 5 步返回的那些行。

由黑色字框指出的步骤在行源上操作，如做 2 表之间的关联，排序，或过滤等操作，后面也会给出详细的介绍：

- 第 2 步实现嵌套的循环操作(相当于 C 语句中的嵌套循环)，接收从第 3 步和第 4 步来的行源，把来自第 3 步源的每一行与它第 4 步中相应的行连接在一起，返回结果行到第 1

步。

- 第 1 步完成一个过滤器操作。它接收来自第 2 步和第 6 步的行源，消除掉第 2 步中来的，在第 6 步有相应行的那些行，并将来自第 2 步的剩下的行返回给发出语句的用户或应用。

实现执行计划步骤的顺序

执行计划中的步骤不是按照它们编号的顺序来实现的：Oracle 首先实现图 5-1 树结构图形里作为叶子出现的那些步骤(例如步骤 3、5、6)。由每一步返回的行称为它下一步骤的行源。然后 Oracle 实现父步骤。

举例来说，为了执行图 5-1 中的语句，Oracle 以下列顺序实现这些步骤：

- 首先，Oracle 实现步骤 3 ，并一行一行地将结果行返回给第 2 步。
- 对第 3 步返回的每一行，Oracle 实现这些步骤：
  -- Oracle 实现步骤 5 ，并将结果 ROWID 返回给第 4 步。
  -- Oracle 实现步骤 4 ，并将结果行返回给第 2 步。
  -- Oracle 实现步骤 2 ，将接受来自第 3 步的一行和来自第 4 步的一行，并返回给第 1 步一行。
  -- Oracle 实现步骤 6 ，如果有结果行的话，将它返回给第 1 步。
  -- Oracle 实现步骤 1 ，如果从步骤 6 返回行，Oracle 将来自第 2 步的行返回给发出 SQL 语句的用户。

注意 Oracle 对由第 3 步返回的每一行实现步骤 5 ， 4 ， 2 ， 6 一次。许多父步骤在它们能执行之前只需要来自它们子步骤的单一行。对这样的父步骤来说，只要从子步骤已返回单一行时立即实现父步骤（可能还有执行计划的其余部分）。如果该父步骤的父步骤同样可以通过单一行返回激活的话，那么它也同样被执行。所以，执行可以在树上串联上去，可能包含执行计划的余下部分。对于这样的操作，可以使用 first_rows 作为优化目标以便于实现快速响应用户的请求。

对每个由子步骤依次检索出来的每一行，Oracle 就实现父步骤及所有串联在一起的步骤一次。对由子步骤返回的每一行所触发的父步骤包括表存取，索引存取，嵌套的循环连接和过滤器。

有些父步骤在它们被实现之前需要来自子步骤的所有行。对这样的父步骤，直到所有行从子步骤返回之前 Oracle 不能实现该父步骤。这样的父步骤包括排序，排序一合并的连接，组功能和总计。对于这样的操作，不能使用 first_rows 作为优化目标，而可以用 all_rows 作为优化目标，使该中类型的操作耗费的资源最少。

有时语句执行时，并不是象上面说的那样一步一步有先有后的进行，而是可能并行运行，如在实际环境中，3、5、4 步可能并行运行，以便取得更好的效率。从上面的树型图上，是很难看出各个操作执行的先后顺序，而通过 ORACLE 生成的另一种形式的执行计划，则可以很容易的看出哪个操作先执行，哪个后执行，这样的执行计划是我们真正需要的，后面会给出详细说明。现在先来看一些预备知识。

访问路径(方法) -- access path

优化器在形成执行计划时需要做的一个重要选择是如何从数据库查询出需要的数据。对于 SQL 语句存取的任何表中的任何行，可能存在许多存取路径(存取方法)，通过它们可以定位和查询出需要的数据。优化器选择其中自认为是最优化的路径。

在物理层，oracle 读取数据，一次读取的最小单位为数据库块(由多个连续的操作系统块组成)，一次读取的最大值由操作系统一次 I/O 的最大值与 multiblock 参数共同决定，所

以即使只需要一行数据，也是将该行所在的数据库块读入内存。逻辑上，oracle 用如下存取方法访问数据：

1) 全表扫描（Full Table Scans, FTS）

为实现全表扫描，Oracle 读取表中所有的行，并检查每一行是否满足语句的 WHERE 限制条件。Oracle 顺序地读取分配给表的每个数据块，直到读到表的最高水线处(high water mark, HWM，标识表的最后一个数据块)。一个多块读操作可以使一次 I/O 能读取多块数据块(db_block_multiblock_read_count 参数设定)，而不是只读取一个数据块，这极大的减少了 I/O 总次数，提高了系统的吞吐量，所以利用多块读的方法可以十分高效地实现全表扫描，而且只有在全表扫描的情况下才能使用多块读操作。在这种访问模式下，每个数据块只被读一次。由于 HWM 标识最后一块被读入的数据，而 delete 操作不影响 HWM 值，所以一个表的所有数据被 delete 后，其全表扫描的时间不会有改善，一般我们需要使用 truncate 命令来使 HWM 值归为 0。幸运的是 oracle 10G 后，可以人工收缩 HWM 的值。

由 FTS 模式读入的数据被放到高速缓存的 Least Recently Used (LRU)列表的尾部，这样可以使其快速交换出内存，从而不使内存重要的数据被交换出内存。

使用 FTS 的前提条件：在较大的表上不建议使用全表扫描，除非取出数据的比较多，超过总量的 5% -- 10%，或你想使用并行查询功能时。

使用全表扫描的例子：

~~~~~~~~~~~~~~~~~~~~~~~~~

```
SQL> explain plan for select * from dual;
Query Plan
----------------------------------------
SELECT STATEMENT        [CHOOSE] Cost=
  TABLE ACCESS FULL DUAL
```

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、数据块以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上，是 Oracle 存取单行数据的最快方法。

为了通过 ROWID 存取表，Oracle 首先要获取被选择行的 ROWID，或者从语句的 WHERE 子句中得到，或者通过表的一个或多个索引的索引扫描得到。Oracle 然后以得到的 ROWID 为依据定位每个被选择的行。

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。我们会经常在执行计划中看到该存取方法，如通过索引查询数据。

使用 ROWID 存取的方法：

```
SQL> explain plan for select * from dept where rowid = 'AAAAyGAADAAAAATAAF';
Query Plan
----------------------------------
SELECT STATEMENT [CHOOSE] Cost=1
TABLE ACCESS BY ROWID DEPT [ANALYZED]
```

3）索引扫描（Index Scan 或 index lookup）

我们先通过 index 查找到数据对应的 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 唯一的表示一行数据，该行对应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，除了存储每个索引的值外，索引还存储具有此值的行对应的 ROWID 值。索引扫描可以由 2 步组成：(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 CACHE 到内存中，所以第 1 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 2 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，这是一个机械操作，相对逻辑 I/O 来说，是极其费时间的。所以如果多大表进行索引扫描，取出的数据如果大于总量的 5% -- 10%，使用索引扫描会效率下降很多。

如下列所示：

```
SQL> explain plan for select empno, ename from emp where empno=10;
Query Plan
----------------------------------
SELECT STATEMENT [CHOOSE] Cost=1
TABLE ACCESS BY ROWID EMP [ANALYZED]
    INDEX UNIQUE SCAN EMP_I1
```

注意 TABLE ACCESS BY ROWID EMP 部分，这表明这不是通过 FTS 存取路径访问数据，而是通过 rowid lookup 存取路径访问数据的。在此例中，所需要的 rowid 是由于在索引查找 empno 列的值得到的，这种方式是 INDEX UNIQUE SCAN 查找，后面给予介绍，EMP_I1 为使用的进行索引查找的索引名字。

但是如果查询的数据能全在索引中找到，就可以避免进行第 2 步操作，避免了不必要的 I/O，此时即使通过索引扫描取出的数据比较多，效率还是很高的，因为这只会在索引中读取。所以上面我在介绍基于规则的优化器时，使用了 select count(id) from SWD_BILLDETAIL where cn <'6'，而没有使用 select count(cn) from SWD_BILLDETAIL where cn <'6'。因为在实际情况中，只查询被索引列的值的情况极为少，所以，如果我在查询中使用 count(cn)，则不具有代表性。

```
SQL> explain plan for select empno from emp where empno=10;   -- 只查询 empno 列值
Query Plan
----------------------------------
SELECT STATEMENT [CHOOSE] Cost=1
  INDEX UNIQUE SCAN EMP_I1
```

进一步讲，如果 sql 语句中对索引列进行排序，因为索引已经预先排序好了，所以在执行计划中不需要再对索引列进行排序

```
SQL> explain plan for select empno, ename from emp
where empno > 7876 order by empno;
Query Plan
----------------------------------------------------------------------------
SELECT STATEMENT     [CHOOSE] Cost=1
TABLE ACCESS BY ROWID EMP [ANALYZED]
  INDEX RANGE SCAN EMP_I1 [ANALYZED]
```

从这个例子中可以看到：因为索引是已经排序了的，所以将按照索引的顺序查询出符合条件的行，因此避免了进一步排序操作。

根据索引的类型与 where 限制条件的不同，有 4 种类型的索引扫描：

索引唯一扫描(index unique scan)

索引范围扫描(index range scan)

索引全扫描(index full scan)

索引快速扫描(index fast full scan)

(1) 索引唯一扫描(index unique scan)

通过唯一索引查找一个数值经常返回单个 ROWID。如果该唯一索引有多个列组成(即组合索引)，则至少要有组合索引的引导列参与到该查询中，如创建一个索引：create index idx_test on emp(ename, deptno, loc)。则 select ename from emp where ename = 'JACK' and deptno = 'DEV'语句可以使用该索引。如果该语句只返回一行，则存取方法称为索引唯一扫描。而 select ename from emp where deptno = 'DEV'语句则不会使用该索引，因为 where 子句种没有引导列。如果存在 UNIQUE 或 PRIMARY KEY 约束（它保证了语句只存取单行）的话，Oracle 经常实现唯一性扫描。使用唯一性约束的例子：

```
SQL> explain plan for
select empno,ename from emp where empno=10;
Query Plan
-----------------------------------
SELECT STATEMENT [CHOOSE] Cost=1
TABLE ACCESS BY ROWID EMP [ANALYZED]
     INDEX UNIQUE SCAN EMP_I1
```

(2) 索引范围扫描(index range scan)

使用一个索引存取多行数据，同上面一样，如果索引是组合索引，如(1)所示，而且 select ename from emp where ename = 'JACK' and deptno = 'DEV'语句返回多行数据，虽然该语句还是使用该组合索引进行查询，可此时的存取方法称为索引范围扫描。在唯一索引上使用索引范围扫描的典型情况下是在谓词(where 限制条件)中使用了范围操作符(如>、<、<>、>=、<=、between)

使用索引范围扫描的例子：

```
SQL> explain plan for select empno,ename from emp
where empno > 7876 order by empno;
Query Plan
-----------------------------------------------------------------------------
SELECT STATEMENT     [CHOOSE] Cost=1
TABLE ACCESS BY ROWID EMP [ANALYZED]
  INDEX RANGE SCAN EMP_I1 [ANALYZED]
```

在非唯一索引上，谓词 col = 5 可能返回多行数据，所以在非唯一索引上都使用索引范围扫描。

使用 index rang scan 的 3 种情况：

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

(b) 在组合索引上，只使用部分列进行查询，导致查询出多行

(c) 对非唯一索引列上进行的任何查询。

(3) 索引全扫描(index full scan)

与全表扫描对应，也有相应的全索引扫描。在某些情况下，可能进行全索引扫描而不是范围扫描，需要注意的是全索引扫描只在 CBO 模式下才有效。CBO 根据统计数值得知进行全索引扫描比进行全表扫描更有效时，才进行全索引扫描，而且此时查询出的数据都必须从索引中可以直接得到。

全索引扫描的例子：

An Index full scan will not perform single block i/o's and so it may prove to be inefficient.

e.g.

Index BE_IX is a concatenated index on big_emp (empno, ename)

```
SQL> explain plan for select empno, ename from big_emp order by empno,ename;
Query Plan
----------------------------------------------------------------------------
SELECT STATEMENT     [CHOOSE] Cost=26
  INDEX FULL SCAN BE_IX [ANALYZED]
```

(4) 索引快速扫描(index fast full scan)

扫描索引中的所有的数据块，与 index full scan 很类似，但是一个显著的区别就是它不对查询出的数据进行排序，即数据不是以排序顺序被返回。在这种存取方法中，可以使用多块读功能，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。

索引快速扫描的例子：

BE_IX 索引是一个多列索引：big_emp (empno,ename)

```
SQL> explain plan for select empno,ename from big_emp;
Query Plan
----------------------------------------
SELECT STATEMENT     [CHOOSE] Cost=1
  INDEX FAST FULL SCAN BE_IX [ANALYZED]
```

只选择多列索引的第 2 列：

```
SQL> explain plan for select ename from big_emp;
Query Plan
----------------------------------------
SELECT STATEMENT     [CHOOSE] Cost=1
  INDEX FAST FULL SCAN BE_IX [ANALYZED]
```

表之间的连接

Join 是一种试图将两个表结合在一起的谓词，一次只能连接 2 个表，表连接也可以被称为表关联。在后面的叙述中，我们将会使用”row source”来代替”表”，因为使用 row source

更严谨一些，并且将参与连接的 2 个 row source 分别称为 row source1 和 row source 2。Join 过程的各个步骤经常是串行操作，即使相关的 row source 可以被并行访问，即可以并行的读取做 join 连接的两个 row source 的数据，但是在将表中符合限制条件的数据读入到内存形成 row source 后，join 的其它步骤一般是串行的。有多种方法可以将 2 个表连接起来，当然每种方法都有自己的优缺点，每种连接类型只有在特定的条件下才会发挥出其最大优势。

row source(表)之间的连接顺序对于查询的效率有非常大的影响。通过首先存取特定的表，即将该表作为驱动表，这样可以先应用某些限制条件，从而得到一个较小的 row source，使连接的效率较高，这也就是我们常说的要先执行限制条件的原因。一般是在将表读入内存时，应用 where 子句中对该表的限制条件。

根据 2 个 row source 的连接条件的中操作符的不同，可以将连接分为等值连接(如 WHERE A.COL3 = B.COL4)、非等值连接(WHERE A.COL3 > B.COL4)、外连接(WHERE A.COL3 = B.COL4(+))。上面的各个连接的连接原理都基本一样，所以为了简单期间，下面以等值连接为例进行介绍。在后面的介绍中，都已：

```
SELECT A.COL1, B.COL2
FROM A, B
WHERE A.COL3 = B.COL4;
```

为例进行说明，假设 A 表为 Row Soruce1，则其对应的连接操作关联列为 COL 3；B 表为 Row Soruce2，则其对应的连接操作关联列为 COL 4；

连接类型：

目前为止，无论连接操作符如何，典型的连接类型共有 3 种：

排序 - - 合并连接(Sort Merge Join (SMJ))

嵌套循环(Nested Loops (NL))

哈希连接(Hash Join)

排序 - - 合并连接(Sort Merge Join, SMJ)

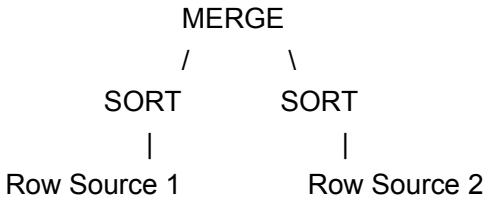
内部连接过程：

1) 首先生成 row source1 需要的数据，然后对这些数据按照连接操作关联列(如 A.col3)进行排序。

2) 随后生成 row source2 需要的数据，然后对这些数据按照与 sort source1 对应的连接操作关联列(如 B.col4)进行排序。

3) 最后两边已排序的行被放在一起执行合并操作，即将 2 个 row source 按照连接条件连接起来

下面是连接步骤的图形表示：



如果 row source 已经在连接关联列上被排序，则该连接操作就不需要再进行 sort 操作，这样可以大大提高这种连接操作的连接速度，因为排序是个极其费资源的操作，

特别是对于较大的表。 预先排序的 row source 包括已经被索引的列(如 a.col3 或 b.col4 上有索引)或 row source 已经在前面的步骤中被排序了。尽管合并两个 row source 的过程是串行的，但是可以并行访问这两个 row source(如并行读入数据，并行排序).

SMJ 连接的例子：

```
SQL> explain plan for
select /*+ ordered */ e.deptno, d.deptno
from emp e, dept d
where e.deptno = d.deptno
order by e.deptno, d.deptno;

Query Plan
-----------------------------------
SELECT STATEMENT [CHOOSE] Cost=17
  MERGE JOIN
    SORT JOIN
      TABLE ACCESS FULL EMP [ANALYZED]
    SORT JOIN
      TABLE ACCESS FULL DEPT [ANALYZED]
```

排序是一个费时、费资源的操作，特别对于大表。基于这个原因，SMJ 经常不是一个特别有效的连接方法，但是如果 2 个 row source 都已经预先排序，则这种连接方法的效率也是蛮高的。

嵌套循环(Nested Loops, NL)

这个连接方法有驱动表(外部表)的概念。其实，该连接过程就是一个 2 层嵌套循环，所以外层循环的次数越少越好，这也就是我们为什么将小表或返回较小 row source 的表作为驱动表(用于外层循环)的理论依据。但是这个理论只是一般指导原则，因为遵循这个理论并不能总保证使语句产生的 I/O 次数最少。有时不遵守这个理论依据，反而会获得更好的效率。如果使用这种方法，决定使用哪个表作为驱动表很重要。有时如果驱动表选择不正确，将会导致语句的性能很差、很差。

内部连接过程：

```
Row source1 的 Row 1 ----------    -- Probe ->    Row source 2
Row source1 的 Row 2 ----------    -- Probe ->    Row source 2
Row source1 的 Row 3 ----------    -- Probe ->    Row source 2
…….
Row source1 的 Row n ----------    -- Probe ->    Row source 2
```

从内部连接过程来看，需要用 row source1 中的每一行，去匹配 row source2 中的所有行，所以此时保持 row source1 尽可能的小与高效的访问 row source2(一般通过索引实现)是影响这个连接效率的关键问题。这只是理论指导原则，目的是使整个连接操作产生最少的物理 I/O 次数，而且如果遵守这个原则，一般也会使总的物理 I/O 数最少。但是如果不遵从这个指导原则，反而能用更少的物理 I/O 实现连接操作，那

尽管违反指导原则吧！因为最少的物理 I/O 次数才是我们应该遵从的真正的指导原则，在后面的具体案例分析中就给出这样的例子。

在上面的连接过程中，我们称 Row source1 为驱动表或外部表。Row Source2 被称为被探查表或内部表。

在 NESTED LOOPS 连接中，Oracle 读取 row source1 中的每一行，然后在 row sourc2 中检查是否有匹配的行，所有被匹配的行都被放到结果集中，然后处理 row source1 中的下一行。这个过程一直继续，直到 row source1 中的所有行都被处理。这是从连接操作中可以得到第一个匹配行的最快的方法之一，这种类型的连接可以用在需要快速响应的语句中，以响应速度为主要目标。

如果 driving row source(外部表)比较小，并且在 inner row source(内部表)上有唯一索引，或有高选择性非唯一索引时，使用这种方法可以得到较好的效率。NESTED LOOPS 有其它连接方法没有的的一个优点是：可以先返回已经连接的行，而不必等待所有的连接操作处理完才返回数据，这可以实现快速的响应时间。

如果不使用并行操作，最好的驱动表是那些应用了 where 限制条件后，可以返回较少行数据的的表，所以大表也可能称为驱动表，关键看限制条件。对于并行查询，我们经常选择大表作为驱动表，因为大表可以充分利用并行功能。当然，有时对查询使用并行操作并不一定会比查询不使用并行操作效率高，因为最后可能每个表只有很少的行符合限制条件，而且还要看你的硬件配置是否可以支持并行(如是否有多个 CPU，多个硬盘控制器)，所以要具体问题具体对待。

NL 连接的例子：

```
SQL> explain plan for
select a.dname,b.sql
from dept a,emp b
where a.deptno = b.deptno;

Query Plan
-------------------------
SELECT STATEMENT [CHOOSE] Cost=5
  NESTED LOOPS
    TABLE ACCESS FULL DEPT [ANALYZED]
    TABLE ACCESS FULL EMP [ANALYZED]
```

哈希连接(Hash Join, HJ)

这种连接是在 oracle 7.3 以后引入的，从理论上来说比 NL 与 SMJ 更高效，而且只用在 CBO 优化器中。

较小的 row source 被用来构建 hash table 与 bitmap，第 2 个 row source 被用来被 hansed，并与第一个 row source 生成的 hash table 进行匹配，以便进行进一步的连接。Bitmap 被用来作为一种比较快的查找方法，来检查在 hash table 中是否有匹配的行。特别的，当 hash table 比较大而不能全部容纳在内存中时，这种查找方法更为

有用。这种连接方法也有 NL 连接中所谓的驱动表的概念，被构建为 hash table 与 bitmap 的表为驱动表，当被构建的 hash table 与 bitmap 能被容纳在内存中时，这种连接方式的效率极高。

HASH 连接的例子：

```
SQL> explain plan for
select /*+ use_hash(emp) */ empno
from emp, dept
where emp.deptno = dept.deptno;

Query Plan
---------------------------
SELECT STATEMENT   [CHOOSE] Cost=3
  HASH JOIN
    TABLE ACCESS FULL DEPT
    TABLE ACCESS FULL EMP
```

要使哈希连接有效，需要设置 HASH_JOIN_ENABLED=TRUE，缺省情况下该参数为 TRUE，另外，不要忘了还要设置 hash_area_size 参数，以使哈希连接高效运行，因为哈希连接会在该参数指定大小的内存中运行，过小的参数会使哈希连接的性能比其他连接方式还要低。

总结一下，在哪种情况下用哪种连接方法比较好：

排序 - - 合并连接(Sort Merge Join, SMJ)：

a) 对于非等值连接，这种连接方式的效率是比较高的。
b) 如果在关联的列上都有索引，效果更好。
c) 对于将 2 个较大的 row source 做连接，该连接方法比 NL 连接要好一些。
d) 但是如果 sort merge 返回的 row source 过大，则又会导致使用过多的 rowid 在表中查询数据时，数据库性能下降，因为过多的 I/O。

嵌套循环(Nested Loops, NL)：

a) 如果 driving row source(外部表)比较小，并且在 inner row source(内部表)上有唯一索引，或有高选择性非唯一索引时，使用这种方法可以得到较好的效率。
b) NESTED LOOPS 有其它连接方法没有的的一个优点是：可以先返回已经连接的行，而不必等待所有的连接操作处理完才返回数据，这可以实现快速的响应时间。

哈希连接(Hash Join, HJ)：

a) 这种方法是在 oracle7 后来引入的，使用了比较先进的连接理论，一般来说，其效率应该好于其它 2 种连接，但是这种连接只能用在 CBO 优化器中，而且需要设置合适的 hash_area_size 参数，才能取得较好的性能。
b) 在 2 个较大的 row source 之间连接时会取得相对较好的效率，在一个 row source 较小时则能取得更好的效率。
c) 只能用于等值连接中

笛卡儿乘积(Cartesian Product)

当两个 row source 做连接，但是它们之间没有关联条件时，就会在两个 row source 中做笛卡儿乘积，这通常由编写代码疏漏造成(即程序员忘了写关联条件)。笛卡尔乘积是一个表的每一行依次与另一个表中的所有行匹配。在特殊情况下我们可以使用笛卡儿乘积，如在星形连接中，除此之外，我们要尽量使用笛卡儿乘积，否则，自己想结果是什么吧！

注意在下面的语句中，在 2 个表之间没有连接。

```
SQL> explain plan for
select emp.deptno,dept,deptno
from emp,dept

Query Plan
-----------------------------
SLECT STATEMENT [CHOOSE] Cost=5
  MERGE JOIN CARTESIAN
    TABLE ACCESS FULL DEPT
    SORT JOIN
      TABLE ACCESS FULL EMP
```

CARTESIAN 关键字指出了在 2 个表之间做笛卡尔乘积。假如表 emp 有 n 行，dept 表有 m 行，笛卡尔乘积的结果就是得到 n * m 行结果。

如何产生执行计划

要为一个语句生成执行计划，可以有 3 种方法：

1)．最简单的办法

```
Sql> set autotrace on
Sql> select * from dual;
```

执行完语句后，会显示 explain plan 与 统计信息。

这个语句的优点就是它的缺点，这样在用该方法查看执行时间较长的 sql 语句时，需要等待该语句执行成功后，才返回执行计划，使优化的周期大大增长。

如果不想执行语句而只是想得到执行计划可以采用：

```
Sql> set autotrace traceonly
```

这样，就只会列出执行计划，而不会真正的执行语句，大大减少了优化时间。虽然也列出了统计信息，但是因为没有执行语句，所以该统计信息没有用处，

如果执行该语句时遇到错误，解决方法为：

(1）在要分析的用户下：

```
 Sqlplus > @ ?\rdbms\admin\utlxplan.sql
```

(2) 用 sys 用户登陆

Sqlplus > @ ?\sqlplus\admin\plustrce.sql
Sqlplus > grant plustrace to user_name; - - user_name 是上面所说的分析用户

2)．用 explain plan 命令

(1) sqlplus > @ ?\rdbms\admin\utlxplan.sql

(2) sqlplus > explain plan set statement_id =’???’ for select ……

注意，用此方法时，并不执行 sql 语句，所以只会列出执行计划，不会列出统计信息，并且执行计划只存在 plan_table 中。所以该语句比起 set autotrace traceonly 可用性要差。需要用下面的命令格式化输出，所以这种方式我用的不多：

```
set linesize 150
set pagesize 500
col PLANLINE for a120
SELECT EXECORD EXEC_ORDER, PLANLINE
  FROM (SELECT PLANLINE, ROWNUM EXECORD, ID, RID
        FROM (SELECT PLANLINE, ID, RID, LEV
              FROM (SELECT lpad(' ',2*(LEVEL),rpad(' ',80,'        '))||
                           OPERATION||'   '||                       -- Operation
                           DECODE(OPTIONS,NULL,'','('||OPTIONS || ') ')||   -- Options
                           DECODE(OBJECT_OWNER,null,'','OF   '''||   OBJECT_OWNER||'.')||     --
Owner
                           DECODE(OBJECT_NAME,null,'',OBJECT_NAME|| ''' ')|| -- Object Name
                           DECODE(OBJECT_TYPE,null,'','('||OBJECT_TYPE||   ')   ')||     --   Object
Type
                           DECODE(ID,0,'OPT_MODE:')||           -- Optimizer
                           DECODE(OPTIMIZER,null,'','ANALYZED','', OPTIMIZER)||
                           DECODE(NVL(COST,0)+NVL(CARDINALITY,0)+NVL(BYTES,0),
                           0,null,' (COST='||TO_CHAR(COST)||',CARD='||
                           TO_CHAR(CARDINALITY)||',BYTES='||TO_CHAR(BYTES)||')')
                           PLANLINE, ID, LEVEL LEV,
                           (SELECT MAX(ID)
                            FROM PLAN_TABLE PL2
                            CONNECT BY PRIOR ID = PARENT_ID
                                  AND PRIOR STATEMENT_ID = STATEMENT_ID
                            START WITH ID = PL1.ID
                                  AND STATEMENT_ID = PL1.STATEMENT_ID) RID
                      FROM PLAN_TABLE PL1
                      CONNECT BY PRIOR ID = PARENT_ID
                          AND PRIOR STATEMENT_ID = STATEMENT_ID
                      START WITH ID = 0
                          AND STATEMENT_ID = 'aaa')
                ORDER BY RID, -LEV))
ORDER BY ID;
```

上面这 2 种方法只能为在本会话中正在运行的语句产生执行计划，即我们需要已经知道了哪条语句运行的效率很差，我们是有目的只对这条 SQL 语句去优化。其实，在很多情况下，我们只会听一个客户抱怨说现在系统运行很慢，而我们不知道是哪个 SQL 引起的。此时有许多现成的语句可以找出耗费资源比较多的语句，如：

```
SELECT ADDRESS,
        substr(SQL_TEXT,1,20) Text,
        buffer_gets,
        executions,
        buffer_gets/executions AVG
FROM    v$sqlarea
WHERE   executions>0
AND     buffer_gets > 100000
ORDER BY 5;
```

从而对找出的语句进行进一步优化。当然我们还可以为一个正在运行的会话中运行的所有 SQL 语句生成执行计划，这需要对该会话进行跟踪，产生 trace 文件，然后对该文件用 tkprof 程序格式化一下，这种得到执行计划的方式很有用，因为它包含其它额外信息，如 SQL 语句执行的每个阶段(如 Parse、Execute、Fetch)分别耗费的各个资源情况(如 CPU、DISK、elapsed 等)。

3)．用 dbms_system 存储过程生成执行计划

因为使用 dbms_system 存储过程可以跟踪另一个会话发出的 sql 语句，并记录所使用的执行计划，而且还提供其它对性能调整有用的信息。因其使用方式与上面 2 种方式有些不太一样，所以在附录中单独介绍。这种方法是对 SQL 进行调整比较有用的方式之一，有些情况下非它不可。具体内容参见附录。

如何分析执行计划

例 1：

假设 LARGE_TABLE 是一个较大的表，且 username 列上没有索引，则运行下面的语句：

```
SQL> SELECT * FROM LARGE_TABLE where USERNAME = 'TEST';
Query Plan
----------------------------------------
SELECT STATEMENT        Optimizer=CHOOSE (Cost=1234 Card=1 Bytes=14)
  TABLE ACCESS FULL LARGE_TABLE [:Q65001] [ANALYZED]
```

在这个例子中，TABLE ACCESS FULL LARGE_TABLE 是第一个操作，意思是在 LARGE_TABLE 表上做全表扫描。当这个操作完成之后，产生的 row source 中的数据被送往下一步骤进行处理，在此例中，SELECT STATEMENT 操作是这个查询语句的最后一步。

Optimizer=CHOOSE 指明这个查询的 optimizer_mode，即 optimizer_mode 初始化参数指定的值，它并不是指语句执行时真的使用了该优化器。决定该语句使用何种优化器的唯一方法是看后面的 cost 部分。例如，如果给出的是下面的形式，则表明使用的是 CBO 优化器，此处的 cost 表示优化器认为该执行计划的代价:

```
SELECT STATEMENT        Optimizer=CHOOSE (Cost=1234 Card=1 Bytes=14)
```

然而假如执行计划中给出的是类似下面的信息，则表明是使用 RBO 优化器，因为 cost 部分的值为空，或者压根就没有 cost 部分。

SELECT STATEMENT Optimizer=CHOOSE Cost=

SELECT STATEMENT Optimizer=CHOOSE

这样我们从 Optimizer 后面的信息中可以得出执行该语句时到底用了什么样的优化器。特别的，如果 Optimizer=ALL_ROWS| FIRST_ROWS| FIRST_ROWS_n，则使用的是 CBO 优化器；如果 Optimizer=RULE，则使用的是 RBO 优化器。

cost 属性的值是一个在 oracle 内部用来比较各个执行计划所耗费的代价的值，从而使优化器可以选择最好的执行计划。不同语句的 cost 值不具有可比性，只能对同一个语句的不同执行计划的 cost 值进行比较。

[:Q65001] 表明该部分查询是以并行方式运行的。里面的数据表示这个操作是由并行查询的一个 slave 进程处理的，以便该操作可以区别于串行执行的操作。

[ANALYZED] 表明操作中引用的对象被分析过了，在数据字典中有该对象的统计信息可以供 CBO 使用。

例 2：

假定 A、B、C 都是不是小表，且在 A 表上一个组合索引：A(a.col1,a.col2) ，注意 a.col1 列为索引的引导列。

考虑下面的查询：

```
select   A.col4
from     A , B , C
where   B.col3 = 10    and   A.col1 = B.col1   and   A.col2 = C.col2   and   C.col3 = 5
Execution Plan
----------------------------------------------------------
   0        SELECT STATEMENT Optimizer=CHOOSE
   1     0    MERGE JOIN
   2     1       SORT (JOIN)
   3     2          NESTED LOOPS
   4     3             TABLE ACCESS (FULL) OF 'B'
   5     3             TABLE ACCESS (BY INDEX ROWID) OF 'A'
   6     5                INDEX (RANGE SCAN) OF 'INX_COL12A' (NON-UNIQUE)
   7     1       SORT (JOIN)
   8     7          TABLE ACCESS (FULL) OF 'C'

Statistics
----------------------------------------------------------
             0   recursive calls
             8   db block gets
             6   consistent gets
             0   physical reads
```

```
      0  redo size
    551  bytes sent via SQL*Net to client
    430  bytes received via SQL*Net from client
      2  SQL*Net roundtrips to/from client
      2  sorts (memory)
      0  sorts (disk)
      6  rows processed
```

在表做连接时，只能 2 个表先做连接，然后将连接后的结果作为一个 row source，与剩下的表做连接，在上面的例子中，连接顺序为 B 与 A 先连接，然后再与 C 连接：

```
   B       <---> A <--->      C
col3=10                     col3=5
```

如果没有执行计划，分析一下，上面的 3 个表应该拿哪一个作为第一个驱动表？从 SQL 语句看来，只有 B 表与 C 表上有限制条件，所以第一个驱动表应该为这 2 个表中的一个，到底是哪一个呢？

B 表有谓词 B.col3 = 10，这样在对 B 表做全表扫描的时候就将 where 子句中的限制条件(B.col3 = 10)用上，从而得到一个较小的 row source, 所以 B 表应该作为第一个驱动表。而且这样的话，如果再与 A 表做关联，可以有效利用 A 表的索引(因为 A 表的 col1 列为 leading column)。

当然上面的查询中 C 表上也有谓词(C.col3 = 5)，有人可能认为 C 表作为第一个驱动表也能获得较好的性能。让我们再来分析一下：如果 C 表作为第一个驱动表，则能保证驱动表生成很小的 row source，但是看看连接条件 A.col2 = C.col2，此时就没有机会利用 A 表的索引，因为 A 表的 col2 列不为 leading column，这样 nested loop 的效率很差，从而导致查询的效率很差。所以对于 NL 连接选择正确的驱动表很重要。

因此上面查询比较好的连接顺序为(B - - > A) - - > C。如果数据库是基于代价的优化器，它会利用计算出的代价来决定合适的驱动表与合适的连接顺序。一般来说，CBO 都会选择正确的连接顺序，如果 CBO 选择了比较差的连接顺序，我们还可以使用 ORACLE 提供的 hints 来让 CBO 采用正确的连接顺序。如下所示：

```
select /*+ ordered */ A.col4
from     B,A,C
where    B.col3 = 10
and      A.col1 = B.col1
and      A.col2 = C.col2
and      C.col3 = 5
```

既然选择正确的驱动表这么重要，那么让我们来看一下执行计划，到底各个表之间是如何关联的，从而得到执行计划中哪个表应该为驱动表：

在执行计划中，需要知道哪个操作是先执行的，哪个操作是后执行的，这对于判断哪个表为驱动表有用处。判断之前，如果对表的访问是通过 rowid，且该 rowid 的值是从索引

扫描中得来得，则将该索引扫描先从执行计划中暂时去掉。然后在执行计划剩下的部分中，判断执行顺序的指导原则就是：最右、最上的操作先执行。具体解释如下：
得到去除妨碍判断的索引扫描后的执行计划：

```
Execution Plan
----------------------------------------------------------
   0        SELECT STATEMENT Optimizer=CHOOSE
   1     0    MERGE JOIN
   2     1      SORT (JOIN)
   3     2        NESTED LOOPS
   4     3          TABLE ACCESS (FULL) OF 'B'
   5     3          TABLE ACCESS (BY INDEX ROWID) OF 'A'
   7     1      SORT (JOIN)
   8     7        TABLE ACCESS (FULL) OF 'C'
```

看执行计划的第 3 列，即字母部分，每列值的左面有空格作为缩进字符。在该列值左边的空格越多，说明该列值的缩进越多，该列值也越靠右。如上面的执行计划所示：第一列值为 6 的行的缩进最多，即该行最靠右；第一列值为 4、5 的行的缩进一样，其靠右的程度也一样，但是第一列值为 4 的行比第一列值为 5 的行靠上；谈论上下关系时，只对连续的、缩进一致的行有效。

从这个图中我们可以看到，对于 NESTED LOOPS 部分，最右、最上的操作是 TABLE ACCESS (FULL) OF 'B'，所以这一操作先执行，所以该操作对应的 B 表为第一个驱动表(外部表)，自然，A 表就为内部表了。从图中还可以看出，B 与 A 表做嵌套循环后生成了新的 row source ，对该 row source 进行来排序后，与 C 表对应的排序了的 row source(应用了 C.col3 = 5 限制条件)进行 MSJ 连接操作。所以从上面可以得出如下事实：B 表先与 A 表做嵌套循环，然后将生成的 row source 与 C 表做排序—合并连接。

通过分析上面的执行计划，我们不能说 C 表一定在 B、A 表之后才被读取，事实上，B 表有可能与 C 表同时被读入内存，因为将表中的数据读入内存的操作可能为并行的。事实上许多操作可能为交叉进行的，因为 ORACLE 读取数据时，如果就是需要一行数据也是将该行所在的整个数据块读入内存，而且还有可能为多块读。

看执行计划时，我们的关键不是看哪个操作先执行，哪个操作后执行，而是关键看表之间连接的顺序(如得知哪个为驱动表，这需要从操作的顺序进行判断)、使用了何种类型的关联及具体的存取路径(如判断是否利用了索引)

在从执行计划中判断出哪个表为驱动表后，根据我们的知识判断该表作为驱动表(就像上面判断 ABC 表那样)是否合适，如果不合适，对 SQL 语句进行更改，使优化器可以选择正确的驱动表。

对于 RBO 优化器：

在 ORACLE 文档上说：对于 RBO 来说，以 from 子句中从右到左的顺序选择驱动表，即最右边的表为第一个驱动表，这是其英文原文：All things being equal RBO chooses the driving order by taking the tables in the FROM clause RIGHT to LEFT。不过，在我做的测试中，从来也没有验证过这种说法是正确的。我认为，即使在 RBO 中，也是有一套规则来决定使用哪种连接类型和哪个表作为驱动表，在选择时肯定会考虑当前索引的情况，还可能会考虑 where 中的限制条件，但是肯定是与 where 中限制条件的位置无关。

测试：

如果我创建 3 个表：

```
create table A(col1 number(4,0),col2 number(4,0), col4 char(30));
create table B(col1 number(4,0),col3 number(4,0), name_b char(30));
create table C(col2 number(4,0),col3 number(4,0), name_c char(30));
create index inx_col12A on a(col1,col2);
```

执行查询：

```
select A.col4
from     B, A, C
where    B.col3 = 10
and      A.col1 = B.col1
and      A.col2 = C.col2
and      C.col3 = 5;
Execution Plan
----------------------------------------------------------
   0         SELECT STATEMENT Optimizer=RULE
   1      0    MERGE JOIN
   2      1       SORT (JOIN)
   3      2          NESTED LOOPS
   4      3             TABLE ACCESS (FULL) OF 'B'
   5      3             TABLE ACCESS (BY INDEX ROWID) OF 'A'
   6      5                INDEX (RANGE SCAN) OF 'INX_COL12A' (NON-UNIQUE)
   7      1       SORT (JOIN)
   8      7          TABLE ACCESS (FULL) OF 'C'

select A.col4
from     B, A, C
where    A.col1 = B.col1
and      A.col2 = C.col2;
Execution Plan
----------------------------------------------------------
   0         SELECT STATEMENT Optimizer=RULE
   1      0    MERGE JOIN
   2      1       SORT (JOIN)
   3      2          NESTED LOOPS
   4      3             TABLE ACCESS (FULL) OF 'B'
   5      3             TABLE ACCESS (BY INDEX ROWID) OF 'A'
   6      5                INDEX (RANGE SCAN) OF 'INX_COL12A' (NON-UNIQUE)
   7      1       SORT (JOIN)
   8      7          TABLE ACCESS (FULL) OF 'C'
```

将 A 表上的索引 inx_col12A 删除后：

```
select A.col4
```

```
from     B, A, C
where    A.col1 = B.col1
and      A.col2 = C.col2;
Execution Plan
----------------------------------------------------------
   0        SELECT STATEMENT Optimizer=RULE
   1     0    MERGE JOIN
   2     1      SORT (JOIN)
   3     2        MERGE JOIN
   4     3          SORT (JOIN)
   5     4            TABLE ACCESS (FULL) OF 'C'
   6     3          SORT (JOIN)
   7     6            TABLE ACCESS (FULL) OF 'A'
   8     1      SORT (JOIN)
   9     8        TABLE ACCESS (FULL) OF 'B'
```

通过上面的这些例子，使我对 oracle 文档上的" All things being equal RBO chooses the driving order by taking the tables in the FROM clause RIGHT to LEFT"这句话持怀疑态度。此时，我也不能使用 hints 来强制优化器使用 nested loop，如果使用了 hints，这样就自动使用 CBO 优化器，而不是 RBO 优化器了。

对于 CBO 优化器：

CBO 根据统计信息选择驱动表，假如没有统计信息，则在 from 子句中从左到右的顺序选择驱动表。这与 RBO 选择的顺序正好相反。这是英文原文(CBO determines join order from costs derived from gathered statistics. If there are no stats then CBO chooses the driving order of tables from LEFT to RIGHT in the FROM clause. This is OPPOSITE to the RBO) 。我还是没法证实这句话的正确性。不过经过验证："如果用 ordered 提示(此时肯定用 CBO)，则以 from 子句中按从左到右的顺序选择驱动表"这句话是正确的。实际上在 CBO 中，如果有统计数据(即对表与索引进行了分析)，则优化器会自动根据 cost 值决定采用哪种连接类型，并选择合适的驱动表，这与 where 子句中各个限制条件的位置没有任何关系。如果我们要改变优化器选择的连接类型或驱动表，则就需要使用 hints 了，具体 hints 的用法在后面会给予介绍。

测试：

如果我创建的 3 个表：

```
create table A(col1 number(4,0),col2 number(4,0), col4 char(30));
create table B(col1 number(4,0),col3 number(4,0), name_b char(30));
create table C(col2 number(4,0),col3 number(4,0), name_c char(30));
create index inx_col12A on a(col1,col2);
```

执行查询：

```
select A.col4
from     B, A, C
where    B.col3 = 10
```

```
and       A.col1 = B.col1
and       A.col2 = C.col2
and       C.col3 = 5;
Execution Plan
----------------------------------------------------------
   0          SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=1 Bytes=110)
   1       0     NESTED LOOPS (Cost=3 Card=1 Bytes=110)
   2       1        MERGE JOIN (CARTESIAN) (Cost=2 Card=1 Bytes=52)
   3       2           TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=1 Bytes=26)
   4       2           SORT (JOIN) (Cost=1 Card=1 Bytes=26)
   5       4              TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=1 Bytes=26)
   6       1        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)

select A.col4
from     B, A, C
where    A.col1 = B.col1
and      A.col2 = C.col2;
Execution Plan
----------------------------------------------------------
   0          SELECT STATEMENT Optimizer=ALL_ROWS (Cost=5 Card=55 Bytes=4620)
   1       0     HASH JOIN (Cost=5 Card=55 Bytes=4620)
   2       1        HASH JOIN (Cost=3 Card=67 Bytes=4757)
   3       2           TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=82 Bytes=1066)
   4       2           TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)
   5       1        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=82 Bytes=1066)
```

将 A 表上的索引 inx_col12A 删除后：

```
select A.col4
from     B, A, C
where    A.col1 = B.col1
and      A.col2 = C.col2;
Execution Plan
----------------------------------------------------------
   0          SELECT STATEMENT Optimizer=ALL_ROWS (Cost=5 Card=55 Bytes=4620)
   1       0     HASH JOIN (Cost=5 Card=55 Bytes=4620)
   2       1        HASH JOIN (Cost=3 Card=67 Bytes=4757)
   3       2           TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=82 Bytes=1066)
   4       2           TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)
   5       1        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=82 Bytes=1066)

select /*+ ORDERED */A.col4
from     C, A, B
where    B.col3 = 10
and      A.col1 = B.col1
```

```
and       A.col2 = C.col2
and       C.col3 = 5;
Execution Plan
----------------------------------------------------------
   0        SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=1 Bytes=110)
   1     0    NESTED LOOPS (Cost=3 Card=1 Bytes=110)
   2     1      NESTED LOOPS (Cost=2 Card=1 Bytes=84)
   3     2        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=1 Bytes=26)
   4     2        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)
   5     1      TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=1 Bytes=26)
```

这个查询验证了通过 ORDERED 提示可以正确的提示优化器选择哪个表作为优化器。

如何干预执行计划 - - 使用 hints 提示

基于代价的优化器是很聪明的，在绝大多数情况下它会选择正确的优化器，减轻了DBA的负担。但有时它也聪明反被聪明误，选择了很差的执行计划，使某个语句的执行变得奇慢无比。此时就需要DBA进行人为的干预，告诉优化器使用我们指定的存取路径或连接类型生成执行计划，从而使语句高效的运行。例如，如果我们认为对于一个特定的语句，执行全表扫描要比执行索引扫描更有效，则我们就可以指示优化器使用全表扫描。在ORACLE中，是通过为语句添加hints(提示)来实现干预优化器优化的目的。

hints是oracle提供的一种机制，用来告诉优化器按照我们的告诉它的方式生成执行计划。我们可以用hints来实现：

1) 使用的优化器的类型
2) 基于代价的优化器的优化目标，是all_rows还是first_rows。
3) 表的访问路径，是全表扫描，还是索引扫描，还是直接利用rowid。
4) 表之间的连接类型
5) 表之间的连接顺序
6) 语句的并行程度

除了”RULE”提示外，一旦使用的别的提示，语句就会自动的改为使用CBO优化器，此时如果你的数据字典中没有统计数据，就会使用缺省的统计数据。所以建议大家如果使用CBO或HINTS提示，则最好对表和索引进行定期的分析。

如何使用 hints:

Hints 只应用在它们所在 sql 语句块(statement block，由 select、update、delete 关键字标识)上，对其它 SQL 语句或语句的其它部分没有影响。如：对于使用 union 操作的 2 个 sql 语句，如果只在一个 sql 语句上有 hints，则该 hints 不会影响另一个 sql 语句。

我们可以使用注释(comment)来为一个语句添加 hints，一个语句块只能有一个注释，而且注释只能放在 SELECT, UPDATE, or DELETE 关键字的后面

使用 hints 的语法：

{DELETE|INSERT|SELECT|UPDATE} /*+ hint [text] [hint[text]]... */

or

{DELETE|INSERT|SELECT|UPDATE} --+ hint [text] [hint[text]]...

注解：

1) DELETE、INSERT、SELECT 和 UPDATE 是标识一个语句块开始的关键字，包含提示的注释只能出现在这些关键字的后面，否则提示无效。

2) “+”号表示该注释是一个 hints，该加号必须立即跟在”/*”的后面，中间不能有空格。

3) hint 是下面介绍的具体提示之一，如果包含多个提示，则每个提示之间需要用一个或多个空格隔开。

4) text 是其它说明 hint 的注释性文本

如果你没有正确的指定 hints，Oracle 将忽略该 hints，并且不会给出任何错误。

使用全套的 hints：

当使用 hints 时，在某些情况下，为了确保让优化器产生最优的执行计划，我们可能指定全套的 hints。例如，如果有一个复杂的查询，包含多个表连接，如果你只为某个表指定了 INDEX 提示(指示存取路径在该表上使用索引)，优化器需要来决定其它应该使用的访问路径和相应的连接方法。因此，即使你给出了一个 INDEX 提示，优化器可能觉得没有必要使用该提示。这是由于我们让优化器选择了其它连接方法和存取路径，而基于这些连接方法和存取路径，优化器认为用户给出的 INDEX 提示无用。为了防止这种情况，我们要使用全套的 hints，如：不但指定要使用的索引，而且也指定连接的方法与连接的顺序等。

下面是一个使用全套 hints 的例子，ORDERED 提示指出了连接的顺序，而且为不同的表指定了连接方法：

```
SELECT /*+ ORDERED INDEX (b, jl_br_balances_n1) USE_NL (j b)
    USE_NL (glcc glf) USE_MERGE (gp gsb) */
    b.application_id, b.set_of_books_id ,
    b.personnel_id, p.vendor_id Personnel,
    p.segment1 PersonnelNumber, p.vendor_name Name
FROM jl_br_journals j, jl_br_balances b,
    gl_code_combinations glcc, fnd_flex_values_vl glf,
    gl_periods gp, gl_sets_of_books gsb, po_vendors p
WHERE ...
```

指示优化器的方法与目标的 hints：

```
ALL_ROWS          -- 基于代价的优化器，以吞吐量为目标
FIRST_ROWS(n)     -- 基于代价的优化器，以响应时间为目标
CHOOSE            -- 根据是否有统计信息，选择不同的优化器
RULE              -- 使用基于规则的优化器
```

例子：

```
SELECT /*+ FIRST_ROWS(10) */ employee_id, last_name, salary, job_id
FROM employees
WHERE department_id = 20;
```

```
SELECT /*+ CHOOSE */ employee_id, last_name, salary, job_id
FROM employees
WHERE employee_id = 7566;

SELECT /*+ RULE */ employee_id, last_name, salary, job_id
FROM employees
WHERE employee_id = 7566;
```

指示存储路径的 hints：

FULL　　/*+ FULL (table) */

指定该表使用全表扫描

ROWID　　/*+ ROWID (table) */

指定对该表使用 rowid 存取方法，该提示用的较少

INDEX　　/*+ INDEX (table [index]) */

使用该表上指定的索引对表进行索引扫描

INDEX_FFS　　/*+ INDEX_FFS (table [index]) */

使用快速全表扫描

NO_INDEX　　/*+ NO_INDEX (table [index]) */

不使用该表上指定的索引进行存取，仍然可以使用其它的索引进行索引扫描

```
SELECT /*+ FULL(e) */ employee_id, last_name
FROM employees e
WHERE last_name LIKE :b1;

SELECT /*+ROWID(employees)*/ *
FROM employees
WHERE rowid > 'AAAAtkAABAAAFNTAAA' AND employee_id = 155;

SELECT /*+ INDEX(A sex_index) use sex_index because there are few
    male patients */ A.name, A.height, A.weight
FROM patients A
WHERE A.sex = 'm';

SELECT /*+NO_INDEX(employees emp_empid)*/ employee_id
FROM employees
WHERE employee_id > 200;
```

指示连接顺序的 hints:

ORDERED　/*+ ORDERED */

按 from 字句中表的顺序从左到右的连接

STAR　　/*+ STAR */

指示优化器使用星型查询

```
  SELECT /*+ORDERED */ o.order_id, c.customer_id, l.unit_price * l.quantity
FROM customers c, order_items l, orders o
WHERE c.cust_last_name = :b1
      AND o.customer_id = c.customer_id
      AND o.order_id = l.order_id;

/*+ ORDERED USE_NL(FACTS) INDEX(facts fact_concat) */
```

指示连接类型的 hints：

USE_NL　　/*+ USE_NL (table [,table, ...]) */

使用嵌套连接

USE_MERGE　　/*+ USE_MERGE (table [,table, ...]) */

使用排序- -合并连接

USE_HASH　　/*+ USE_HASH (table [,table, ...]) */

使用 HASH 连接

注意：如果表有 alias(别名)，则上面的 table 指的是表的别名，而不是真实的表名

具体的测试实例：

```
create table A(col1 number(4,0),col2 number(4,0), col4 char(30));
create table B(col1 number(4,0),col3 number(4,0), name_b char(30));
create table C(col2 number(4,0),col3 number(4,0), name_c char(30));

select A.col4
from    C , A , B
where   C.col3 = 5    and   A.col1 = B.col1   and   A.col2 = C.col2
    and     B.col3 = 10;
Execution Plan
----------------------------------------------------------
   0        SELECT STATEMENT Optimizer=CHOOSE
   1    0     MERGE JOIN
   2    1       SORT (JOIN)
   3    2         MERGE JOIN
   4    3           SORT (JOIN)
   5    4             TABLE ACCESS (FULL) OF 'B'
   6    3           SORT (JOIN)
   7    6             TABLE ACCESS (FULL) OF 'A'
   8    1       SORT (JOIN)
   9    8         TABLE ACCESS (FULL) OF 'C'

select /*+ ORDERED */ A.col4
 from    C , A , B
 where   C.col3 = 5    and   A.col1 = B.col1   and   A.col2 = C.col2
 and     B.col3 = 10;
Execution Plan
```

```
----------------------------------------------------------
    0        SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=1 Bytes=110)
    1    0     HASH JOIN (Cost=5 Card=1 Bytes=110)
    2    1       HASH JOIN (Cost=3 Card=1 Bytes=84)
    3    2         TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=1 Bytes=26)
    4    2         TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)
    5    1       TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=1 Bytes=26)

select /*+ ORDERED USE_NL (A C)*/ A.col4
 from     C , A , B
 where    C.col3 = 5     and    A.col1 = B.col1    and    A.col2 = C.col2
 and      B.col3 = 10;
Execution Plan
----------------------------------------------------------
    0        SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=1 Bytes=110)
    1    0     HASH JOIN (Cost=4 Card=1 Bytes=110)
    2    1       NESTED LOOPS (Cost=2 Card=1 Bytes=84)
    3    2         TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=1 Bytes=26)
    4    2         TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)
    5    1       TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=1 Bytes=26)
```

创建索引：

```
create index inx_col12A on a(col1,col2);
select A.col4
 from     C , A , B
 where    C.col3 = 5     and    A.col1 = B.col1    and    A.col2 = C.col2
 and      B.col3 = 10;
Execution Plan
----------------------------------------------------------
    0        SELECT STATEMENT Optimizer=CHOOSE
    1    0     MERGE JOIN
    2    1       SORT (JOIN)
    3    2         NESTED LOOPS
    4    3           TABLE ACCESS (FULL) OF 'B'
    5    3           TABLE ACCESS (BY INDEX ROWID) OF 'A'
    6    5             INDEX (RANGE SCAN) OF 'INX_COL12A' (NON-UNIQUE)
    7    1       SORT (JOIN)
    8    7         TABLE ACCESS (FULL) OF 'C'

select /*+ ORDERED */ A.col4
 from     C , A , B
 where    C.col3 = 5     and    A.col1 = B.col1    and    A.col2 = C.col2
 and      B.col3 = 10;
Execution Plan
```

```
---------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=1 Bytes=110)
    1    0    HASH JOIN (Cost=5 Card=1 Bytes=110)
    2    1      HASH JOIN (Cost=3 Card=1 Bytes=84)
    3    2        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=1 Bytes=26)
    4    2        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)
    5    1      TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=1 Bytes=26)

select /*+ ORDERED USE_NL (A C)*/ A.col4
 from    C , A , B
 where   C.col3 = 5    and   A.col1 = B.col1   and   A.col2 = C.col2
 and     B.col3 = 10;
Execution Plan
---------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=1 Bytes=110)
    1    0    HASH JOIN (Cost=4 Card=1 Bytes=110)
    2    1      NESTED LOOPS (Cost=2 Card=1 Bytes=84)
    3    2        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=1 Bytes=26)
    4    2        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)
    5    1      TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=1 Bytes=26)

select /*+ USE_NL (A C)*/ A.col4
 from    C , A , B
 where   C.col3 = 5    and   A.col1 = B.col1   and   A.col2 = C.col2
 and     B.col3 = 10;
```

我们这个查询的意思是让 A、C 表做 NL 连接，并且让 A 表作为内表，但是从执行计划来看，没有达到我们的目的。

```
Execution Plan
---------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=1 Bytes=110)
    1    0    NESTED LOOPS (Cost=3 Card=1 Bytes=110)
    2    1      MERGE JOIN (CARTESIAN) (Cost=2 Card=1 Bytes=52)
    3    2        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=1 Bytes=26)
    4    2        SORT (JOIN) (Cost=1 Card=1 Bytes=26)
    5    4          TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=1 Bytes=26)
    6    1      TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=82 Bytes=4756)
```

对对象进行分析后：

```
analyze table a compute statistics;
analyze table b compute statistics;
analyze table c compute statistics;
analyze index inx_col12A compute statistics;
select A.col4
 from    C , A , B
```

```
 where   C.col3 = 5     and   A.col1 = B.col1   and   A.col2 = C.col2
 and     B.col3 = 10;
Execution Plan
----------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=8 Bytes=336)
    1    0    HASH JOIN (Cost=5 Card=8 Bytes=336)
    2    1      MERGE JOIN (CARTESIAN) (Cost=3 Card=8 Bytes=64)
    3    2        TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=2 Bytes=8)
    4    2        SORT (JOIN) (Cost=2 Card=4 Bytes=16)
    5    4          TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=4 Bytes=16)
    6    1      TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=30 Bytes=1020)

select /*+ ORDERED */ A.col4
 from    C , A , B
 where   C.col3 = 5     and   A.col1 = B.col1   and   A.col2 = C.col2
 and     B.col3 = 10;
Execution Plan
----------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=9 Bytes=378)
    1    0    HASH JOIN (Cost=5 Card=9 Bytes=378)
    2    1      HASH JOIN (Cost=3 Card=30 Bytes=1140)
    3    2        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=4 Bytes=16)
    4    2        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=30 Bytes=1020)
    5    1      TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=2 Bytes=8)

select /*+ ORDERED USE_NL (A C)*/ A.col4
 from    C , A , B
 where   C.col3 = 5     and   A.col1 = B.col1   and   A.col2 = C.col2
 and     B.col3 = 10;
Execution Plan
----------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=7 Card=9 Bytes=378)
    1    0    HASH JOIN (Cost=7 Card=9 Bytes=378)
    2    1      NESTED LOOPS (Cost=5 Card=30 Bytes=1140)
    3    2        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=4 Bytes=16)
    4    2        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=30 Bytes=1020)
    5    1      TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=2 Bytes=8)

select /*+ USE_NL (A C)*/ A.col4
 from    C , A , B
 where   C.col3 = 5     and   A.col1 = B.col1   and   A.col2 = C.col2
 and     B.col3 = 10;
Execution Plan
----------------------------------------------------------
```

```
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=7 Card=9 Bytes=378)
    1    0    HASH JOIN (Cost=7 Card=9 Bytes=378)
    2    1      NESTED LOOPS (Cost=5 Card=30 Bytes=1140)
    3    2        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=4 Bytes=16)
    4    2        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=30 Bytes=1020)
    5    1      TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=2 Bytes=8)

select /*+ ORDERED USE_NL (A B C) */ A.col4
 from    C , A , B
 where   C.col3 = 5    and   A.col1 = B.col1   and   A.col2 = C.col2
 and     B.col3 = 10;
Execution Plan
----------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=35 Card=9 Bytes=378)
    1    0    NESTED LOOPS (Cost=35 Card=9 Bytes=378)
    2    1      NESTED LOOPS (Cost=5 Card=30 Bytes=1140)
    3    2        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=4 Bytes=16)
    4    2        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=30 Bytes=1020)
    5    1      TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=2 Bytes=8)
```

对于这个查询我无论如何也没有得到类似下面这样的执行计划：

```
Execution Plan
----------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=35 Card=9 Bytes=378)
    1    0    NESTED LOOPS (Cost=35 Card=9 Bytes=378)
    2    1      TABLE ACCESS (FULL) OF 'B' (Cost=1 Card=2 Bytes=8)
    3    1      NESTED LOOPS (Cost=5 Card=30 Bytes=1140)
    4    3        TABLE ACCESS (FULL) OF 'C' (Cost=1 Card=4 Bytes=16)
    5    3        TABLE ACCESS (FULL) OF 'A' (Cost=1 Card=30 Bytes=1020)
```

从上面的这些例子我们可以看出：通过给语句添加 HINTS，让其按照我们的意愿执行，有时是一件很困难的事情，需要不断的尝试各种不同的 hints。对于 USE_NL 与 USE_HASH 提示，建议同 ORDERED 提示一起使用，否则不容易指定那个表为驱动表。

具体案例分析：

环境：oracle 817 + linux + 阵列柜
swd_billdetail 表 5000 万条数据
SUPER_USER 表 2800 条数据
连接列上都有索引，而且 super_user 中的一条对应于 swd_billdetail 表中的很多条记录
表与索引都做了分析。

实际应用的查询为：

```
select a.CHANNEL, B.user_class
```

```
from    swd_billdetail B, SUPER_USER A
where A.cn = B.cn;
```

这样在分析时导致查询出的数据过多，不方便，所以用count(a.CHANNEL||B.user_class)来代替，而且 count(a.CHANNEL||B.user_class)操作本身并不占用过多的时间，所以可以接受此种替代。

利用索引查询出 SWD_BILLDETAIL 表中所有记录的方法

```
SQL> select count(id) from SWD_BILLDETAIL;
  COUNT(ID)
----------
   53923574
Elapsed: 00:02:166.00
Execution Plan
----------------------------------------------------------
0       SELECT STATEMENT Optimizer=CHOOSE (Cost=18051 Card=1)
1    0    SORT (AGGREGATE)
2    1      INDEX (FAST FULL SCAN) OF 'SYS_C001851' (UNIQUE) (Cost=18051 Card=54863946)

Statistics
----------------------------------------------------------
          0  recursive calls
       1952  db block gets
     158776  consistent gets
     158779  physical reads
       1004  redo size
        295  bytes sent via SQL*Net to client
        421  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

利用全表扫描从 SWD_BILLDETAIL 表中取出全部数据的方法。

```
SQL> select count(user_class) from swd_billdetail;
COUNT(USER_CLASS)
-----------------
         53923574
Elapsed: 00:11:703.07
Execution Plan
----------------------------------------------------------
0       SELECT STATEMENT Optimizer=CHOOSE (Cost=165412 Card=1 Bytes=2)
1    0    SORT (AGGREGATE)
```

2 1 TABLE ACCESS (FULL) OF 'SWD_BILLDETAIL' (Cost=165412 Card=54863946 Bytes=109727892)

Statistics
----
0 recursive calls
8823 db block gets
1431070 consistent gets
1419520 physical reads
0 redo size
303 bytes sent via SQL*Net to client
421 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
1 sorts (memory)
0 sorts (disk)
1 rows processed

select count(a.CHANNEL||B.user_class)
from swd_billdetail B, SUPER_USER A
where A.cn = B.cn;
EXEC_ORDER PLANLINE
---------- ----------
6 SELECT STATEMENT OPT_MODE:CHOOSE (COST=108968,CARD=1,BYTES=21)
5 SORT (AGGREGATE) (COST=,CARD=1,BYTES=21)
4 NESTED LOOPS (COST=108968,CARD=1213745,BYTES=25488645)
1 TABLE ACCESS (FULL) OF 'SWORD.SUPER_USER' (COST=2,CARD=2794,BYTES=27940)
3 TABLE ACCESS (BY INDEX ROWID) OF 'SWORD.SWD_BILLDETAIL' (COST=39,CARD=54863946,BYTES=603503406)
2 INDEX (RANGE SCAN) OF 'SWORD.IDX_DETAIL_CN' (NON-UNIQUE) (COST=3,CARD=54863946,BYTES=)

这个查询耗费的时间很长，需要 1 个多小时。
运行后的信息如下：
COUNT(A.CHANNEL||B.USER_CLASS)

1186387

Elapsed: 01:107:6429.87

Execution Plan

```
-------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=108968 Card=1 Bytes=21)
   1    0    SORT (AGGREGATE)
   2    1      NESTED LOOPS (Cost=108968 Card=1213745 Bytes=25488645)
   3      2        TABLE ACCESS (FULL) OF 'SUPER_USER' (Cost=2
Card=2794Bytes=27940)
   4      2        TABLE ACCESS (BY INDEX ROWID) OF 'SWD_BILLDETAIL'
(Cost=39 Card=54863946 Bytes=603503406)
   5      4          INDEX (RANGE SCAN) OF 'IDX_DETAIL_CN' (NON-UNIQUE)
(Cost=3 Card=54863946)

Statistics
-------------------------------------------------------
          0  recursive calls
          4  db block gets
    1196954  consistent gets
    1165726  physical reads
          0  redo size
        316  bytes sent via SQL*Net to client
        421  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

将语句中加入 hints，让 oracle 的优化器使用嵌套循环，并且大表作为驱动表，生成新的执行计划：

```
select /*+ ORDERED USE_NL(A) */ count(a.CHANNEL||B.user_class)
from   swd_billdetail B, SUPER_USER A
where A.cn = B.cn;

EXEC_ORDER PLANLINE
---------- ----------------------------------------------------------------------------------------------------
         6          SELECT STATEMENT OPT_MODE:CHOOSE
(COST=109893304,CARD=1,BYTES=21)
         5     SORT (AGGREGATE) (COST=,CARD=1,BYTES=21)
         4                                   NESTED LOOPS
(COST=109893304,CARD=1213745,BYTES=25488645)
         1          TABLE ACCESS (FULL) OF 'SWORD.SWD_BILLDETAIL'
(COST=165412,CARD=54863946,BYTES=603503406)
         3                TABLE ACCESS (BY INDEX ROWID) OF
'SWORD.SUPER_USER' (COST=2,CARD=2794,BYTES=27940)
         2          INDEX (RANGE SCAN) OF 'SWORD.IDX_SUPER_USER_CN'
(NON-UNIQUE) (COST=1,CARD=2794,BYTES=)
```

这个查询耗费的时间较短，才 20 分钟，性能比较好。
运行后的信息如下：

```
COUNT(A.CHANNEL||B.USER_CLASS)
------------------------------
                       1186387

Elapsed: 00:20:1208.87

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=109893304 Card=1
Bytes=21)
   1    0   SORT (AGGREGATE)
   2    1     NESTED LOOPS (Cost=109893304 Card=1213745 Bytes=25488645)
   3    2       TABLE ACCESS (FULL) OF 'SWD_BILLDETAIL' (Cost=165412
Card=54863946 Bytes=603503406)
   4    2       TABLE ACCESS (BY INDEX ROWID) OF 'SUPER_USER'
(Cost=2Card=2794 Bytes=27940)
   5    4         INDEX (RANGE SCAN) OF 'IDX_SUPER_USER_CN'
(NON-UNIQUE) (Cost=1 Card=2794)

Statistics
----------------------------------------------------------
          0  recursive calls
       8823  db block gets
   56650250  consistent gets
    1413250  physical reads
          0  redo size
        316  bytes sent via SQL*Net to client
        421  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

总结：

因为上两个查询都是采用 nested loop 循环，这时采用哪个表作为 driving table 就很重要。在第一个 sql 中，小表(SUPER_USER)作为 driving table，符合 oracle 优化的建议，但是由于 SWD_BILLDETAIL 表中 cn 列的值有很多重复的，这样对于 SUPER_USER 中的每一行，都会在 SWD_BILLDETAIL 中有很多行，利用索引查询出这些行的 rowid 很快，但是再利用这些 rowid 去查询 SWD_BILLDETAIL 表中的 user_class 列的值，就比较慢了。原因是这些 rowid 是随机的，而且该表比较大，不可能缓存到内存，所以几乎每次按照 rowid 查询都需要读物理磁盘，这就是该执行计划比较慢的真正原因。从结果可以得到验证：查询

出 1186387 行，需要利用 rowid 从 SWD_BILLDETAIL 表中读取 1186387 次，而且大部分为从硬盘上读取。

反其道而行之，利用大表(SWD_BILLDETAIL)作为 driving 表，这样大表只需要做一次全表扫描(而且会使用多块读功能，每次物理 I/O 都会读取几个 oracle 数据块，从而一次读取很多行，加快了执行效率)，对于读出的每一行，都与 SUPER_USER 中的行进行匹配，因为 SUPER_USER 表很小，所以可以全部放到内存中，这样匹配操作就极快，所以该 sql 执行的时间与 SWD_BILLDETAIL 表全表扫描的时间差不多(SWD_BILLDETAIL 全表用 11 分钟，而此查询用 20 分钟)。

另外：如果 SWD_BILLDETAIL 表中 cn 列的值唯一，则第一个 sql 执行计划执行的结果或许也会不错。如果 SUPER_USER 表也很大，如 500 万行，则第 2 个 sql 执行计划执行的结果反而又可能会差。其实，如果 SUPER_USER 表很小，则第 2 个 sql 语句的执行计划如果不利用 SUPER_USER 表的索引，查询或许会更快一些，我没有对此进行测试。

所以在进行性能调整时，具体问题要具体分析，没有一个统一的标准。

第 6 章 其它注意事项

1. 不要认为将 optimizer_mode 参数设为 rule，就认为所有的语句都使用基于规则的优化器
   不管 optimizer_mode 参数如何设置，只要满足下面 3 个条件，就一定使用 CBO。
   1) 如果使用 Index Only Tables(IOTs), 自动使用 CBO.
   2) Oracle 7.3 以后，如果表上的 Paralle degree option 设为>1，
      则自动使用 CBO, 而不管是否用 rule hints.
   3) 除 rlue 以外的任何 hints 都将导致自动使用 CBO 来执行语句

总结一下，一个语句在运行时到底使用何种优化器可以从下面的表格中识别出来，从上到下看你的语句到底是否满足 description 列中描述的条件：

Description	对象是否被分析	优化器的类型
~~	~~~~~~~~~~~~	~~~~~~~~~
Non-RBO Object (Eg:IOT)	n/a	#1
Parallelism > 1	n/a	#1
RULE hint	n/a	RULE
ALL_ROWS hint	n/a	ALL_ROWS
FIRST_ROWS hint	n/a	FIRST_ROWS
*Other Hint	n/a	#1
OPTIMIZER_GOAL=RULE	n/a	RULE
OPTIMIZER_GOAL=ALL_ROWS	n/a	ALL_ROWS
OPTIMIZER_GOAL=FIRST_ROWS	n/a	FIRST_ROWS
OPTIMIZER_GOAL=CHOOSE	NO	RULE
OPTIMIZER_GOAL=CHOOSE	YES	ALL_ROWS

#1 表示除非 OPTIMIZER_GOAL 被设置为 FIRST_ROWS ，否则将使用 ALL_ROWS。在 PL/SQL 中，则一直是使用 ALL_ROWS

*Other Hint 表示是指除 RULE、ALL_ROWS 和 FIRST_ROWS 以外的其它提示

2) 当 CBO 选择了一个次优化的执行计划时, 不要同 CBO 过意不去, 先采取如下措施:
   a) 检查是否在表与索引上又最新的统计数据
   b) 对所有的数据进行分析，而不是只分析一部分数据
   c) 检查是否引用的数据字典表，在 oracle 10G 之前，缺省情况下是不对数据字典表进行分析的。
   d) 试试 RBO 优化器，看语句执行的效率如何，有时 RBO 能比 CBO 产生的更好的执行计划
   e) 如果还不行，跟踪该语句的执行，生成 trace 信息，然后用 tkprof 格式化 trace 信息，这样可以得到全面的供优化的信息。

3) 假如利用附录的方法对另一个会话进行 trace，则该会话应该为专用连接

4) 不要认为绑定变量(bind variables)的缺点只有书写麻烦，而优点多多，实际上使用绑定

变量虽然避免了重复 parse，但是它导致优化器不能使用数据库中的列统计，从而选择了较差的执行计划。而使用硬编码的 SQL 则可以使用列统计。当然随着 CBO 功能的越来越强，这种情况会得到改善。目前就已经实现了在第一次运行绑定变量的 sql 语句时，考虑列统计。

5) 如果一个 row source 超过 10000 行数据，则可以被认为大 row source

6) 有(+)的表不是 driving table，注意：如果有外联接，而且 order hint 指定的顺序与外联结决定的顺序冲突，则忽略 order hint

7) 影响 CBO 选择 execution plan 的初始化参数:

这些参数会影响 cost 值
ALWAYS_ANTI_JOIN
B_TREE_BITMAP_PLANS
COMPLEX_VIEW_MERGING
DB_FILE_MULTIBLOCK_READ_COUNT
FAST_FULL_SCAN_ENABLED
HASH_AREA_SIZE
HASH_JOIN_ENABLED
HASH_MULTIBLOCK_IO_COUNT
OPTIMIZER_FEATURES_ENABLE
OPTIMIZER_INDEX_CACHING
OPTIMIZER_INDEX_COST_ADJ
OPTIMIZER_MODE> / GOAL
OPTIMIZER_PERCENT_PARALLEL
OPTIMIZER_SEARCH_LIMIT
PARTITION_VIEW_ENABLED
PUSH_JOIN_PREDICATE
SORT_AREA_SIZE
SORT_DIRECT_WRITES
SORT_WRITE_BUFFER_SIZE
STAR_TRANSFORMATION_ENABLED
V733_PLANS_ENABLED
CURSOR_SHARING

附录：

如何通过跟踪一个客户端程序发出的 sql 的方法来优化 SQL

简要说来，跟踪一个客户程序发出的 SQL 主要分成下面几步：

1) 识别要跟踪的客户端程序到数据库的连接(后面都用 session 代替)，主要找出能唯一识别一个 session 的 sid 与 serial#.
2) 设定相应的参数，如打开时间开关(可以知道一个 sql 执行了多长时间)，存放跟踪数据的文件的位置、最大值。
3) 启动跟踪功能
4) 让系统运行一段时间，以便可以收集到跟踪数据
5) 关闭跟踪功能
6) 格式化跟踪数据，得到我们易于理解的跟踪结果。

现在就每一步，给出详细的说明：

1) 识别要跟踪的客户端程序到数据库的数据库连接

查询 session 信息(在 sql*plus 中运行)：

```
set linesize 190
col machine format a30 wrap
col program for a40
col username format a15 wrap
set pagesize 500
select s.sid sid, s.SERIAL# "serial#", s.username, s.machine, s.program,
      p.spid ServPID, s.server
from v$session s, v$process p
where p.addr = s.paddr ;
```

如得到的一个查询结果如下：

```
SID   serial#  USERNAME MACHINE                   PROGRAM         SERVPID         SERVER
----  -------  -------- ------------------        -------------   ---------       ---------
  8        3  SCOTT    WORKGROUP\SUNNYXU    SQLPLUS.EXE     388             DEDICATED

LOGON_TIME
------------------
 2005.06.28 18:50:11
```

上面的结果中比较有用的列为：

sid, serial# : 这两个值联合起来唯一标识一个 session

username : 程序连接数据库的用户名

machine : 连接数据库的程序所在的机器的机器名，可以 hostname 得到

program : 连接数据库的程序名，所有用 java jdbc thin 的程序的名字都一样，

servpid : 与程序对应的服务器端的服务器进程的进程号，在 unix 下比较有用

server : 程序连接数据库的模式：专用模式(dedicaed)、共享模式(shared)。

只有在专用模式下的数据库连接，对其进程跟踪才有效
logon_time : 程序连接数据库的登陆时间

根据 machine, logon_time 可以方便的识别出一个数据库连接对应的 session，从而得到该 sesion 的唯一标识 sid, serial#, 为对该 session 进行跟踪做好准备

2) 设定相应的参数
参数说明：
timed_statistics : 收集跟踪信息时，是否将收集时间信息，如果收集，
则可以知道一个 sql 的各个执行阶段耗费的时间情况
user_dump_dest : 存放跟踪数据的文件的位置
max_dump_file_size : 放跟踪数据的文件的最大值，防止由于无意的疏忽，
使跟踪数据的文件占用整个硬盘，影响系统的正常运行

设置的方法：

```
SQL> exec sys.dbms_system.set_bool_param_in_session( -
    sid         => 8, -
    serial# => 3, -
    parnam    => 'timed_statistics', -
    bval        => true);

SQL> alter system set user_dump_dest='c:\temp';
```

-- 注意这个语句会改变整个系统的跟踪文件存放的位置，所以我一般不改这个参数，而用系统的缺省值，要查看当前系统的该参数的值，可以用 system 用户登陆后：

```
SQL> show parameter user_dump_dest

SQL> exec sys.dbms_system.set_int_param_in_session( -
    sid         => 8, -
    serial# => 3, -
    parnam    => 'max_dump_file_size', -
    intval    => 2147483647)
```

3) 启动跟踪功能

```
SQL> exec sys.dbms_system.set_sql_trace_in_session(8, 3, true);
```

注意，只有跟踪的 session 再次发出 sql 语句后，才会产生 trc 文件

4) 让系统运行一段时间，以便可以收集到跟踪数据

5) 关闭跟踪功能

```
SQL> exec sys.dbms_system.set_sql_trace_in_session(8,3,false);
```

6) 格式化跟踪数据，得到我们易于理解的跟踪结果。
对产生的 trace 文件进行格式化：
在命令提示符下，运行下面的命令

tkprof dsdb2_ora_18468.trc dsdb2_trace.out SYS=NO EXPLAIN=SCOTT/TIGER
其它使用 tkprof 的例子：
(a) tkprof tracefile.trc sort_1.prf explain=apps/your_apps_password print=10 sort='(prsqry,exeqry,fchqry,prscu,execu,fchcu)'
(b) tkprof tracefile.trc sort_2.prf explain=apps/your_apps_password print=10 sort='(prsela,exeela,fchela)'
(c) tkprof tracefile.trc sort_3.prf explain=apps/your_apps_password print=10 sort='(prscnt,execnt,fchcnt)'
(d) tkprof tracefile.trc normal.prf explain=apps/your_apps_password

现对 tkprof 程序做进一步的说明：

在打开跟踪功能后，oracle 将被跟踪 session 中正在执行的 SQL 的性能状态数据都收集到一个跟踪文件中。这个跟踪文件提供了许多有用的信息，例如一个 sql 的解析次数、执行次数、fetch 次数、物理读次数、逻辑读次数、CPU 使用时间等，利用这些信息可以诊断你的 sql 的问题，从而用来优化你的系统。不幸的是，生成的跟踪文件中的数据是我们难以理解的，所以要用 TKPROF 工具对其进行转换，转换成我们易于理解格式。tkprof 是 oracle 提供的实用工具，类似于 sql*plus，在安装完 oracle 客户端后就自动安装到系统中，直接在命令符下用就可以了。

当在打开跟踪功能时发生了 recursive calls，则 tkprof 也会产生这些 recursive calls 的统计信息，并清楚的在格式化输出文件中标名它们为 recursive calls。
注意：recursive calls 的统计数据是包含在 recursive calls 上的，并不包含在引起该 recursive calls 语句的 sql 语句上面。所以计算一个 sql 语句耗费的资源时，也要考虑该 sql 语句引起 recursive calls 语句花费的资源。通过将 sys 参数设为 no 时，我们变可以在格式化的输出文件中屏蔽掉这些 recursive calls 信息。

如何得到 tkprof 的帮助信息：
运行 tkprof 时，不带任何参数，就可以得到该工具的帮助信息。

执行计划：

一个语句的执行计划是 oracle 执行这个 sql 语句的一系列指令。通过检验执行计划，你可以更好的知道 oracle 如何执行你的 sql 语句，这个信息可以帮助你决定是否你写的 sql 语句已经使用了索引。
如果在 tkprof 中指定了 EXPLAIN 参数，tkprof 使用 EXPLAIN PLAN 命令来为每个被跟踪的 sql 语句产生执行计划。

使用说明：
TKPROF 工具接受一个 trace 文件作为输入文件，利用提供给命令的多个参数对 trace 文件进行分析，然后将格式化好的结果放到一个输出文件中。

TKPROF 的使用语法：

```
---------------
TKPROF command ::=
>>-- TKPROF traced_file formatted_file ----------------------------------------->
                                        |                                       |
                                        +- SORT = ------------------------------+
                                                  |                           |
                                                  +--        OPTION    --+
                                                  |                           |
                                                  |      +---- , ----+        |
                                                  |      V           |        |
                                                  |__(       OPTION       )__|

>------------------------------------------------------------------------------>
    |                            |    |                            |    |                    |
    +-- PRINT = integer --+      +-- INSERT = filname3 --+      +-- SYS = ---------+
                                                                                  |          |
                                                                                  +- YES -+
                                                                                  |          |
                                                                                  +- NO --+

>------------------------------------------------------------------------------>
    |                                                                             |
    +--------------------------------------   EXPLAIN = user/password ------+
      |                                            |
      +----    TABLE = schema.table    ----+

>------------------------------------------------------------------------------><
    |                                    |
    +----    RECORD = filname    ----+
```

各个参数的含义：

' traced_file '

指定输入文件，即 oracle 产生的 trace 文件，该文件中可以只包含一个 session 的跟踪信息，也可以包含系统中所有 session 的信息(此时需要在系统级进行跟踪)

'formatted_file'

指定输出文件，即我们想得到的易于理解的格式化文件，我们利用该文件对会话运行的 sql 进行分析。

'EXPLAIN'

利用哪个用户对 trace 文件中的 sql 进行分析，从而得到该 sql 语句的执行计划，这也说明在 trace file 中并没有各个 sql 语句的执行计划，只是在运行 tkprof 程序时才将 trace file 文件中的 sql 语句用 explian 参数指定的用户连接到数据库，然后运用 EXPLAIN PLAN 命令生成 sql 的执行计划。

这个用户一般是你的程序中连接数据库的用户

'TABLE'

在对 sql 语句进行分析时，将产生的执行计划暂时存放到该表中。
一般不需要该参数，这样当表不存在时，tkprof 会自动创建相应的表，
并在分析完之后，将创建的表自动删除。如果要指定自定义的表，该表的结构必须与 utlxplan.sql 文件中指定的表的结构一样。
我一般不设置这个参数，让其采用默认的表名，并自动创建、删除

'SYS'

是否对 sys 用户运行的 sql 语句或被跟踪 session 产生的 recursive SQL 也进行分析，并将分析结果放到输出文件中。缺省值为 YES。
我一般设为 NO，这样输出文件中只包含我发出的 sql 语句，
而不包含系统产生的 sql。

SORT

按照指定的排序选项(条件)对格式化好的 sql 语句进行降序排列，然后存放到输出文件中。可以将多个排序选项组合起来，如果没有指定排序选项，则按照使用 sql 的先后顺序。

排序选项有：

prscnt number of times parse was called
prscpu cpu time parsing
prsela elapsed time parsing
prsdsk number of disk reads during parse
prsqry number of buffers for consistent read during parse
prscu number of buffers for current read during parse
prsmis number of misses in library cache during parse
execnt number of execute was called
execpu cpu time spent executing
exeela elapsed time executing
exedsk number of disk reads during execute
exeqry number of buffers for consistent read during execute
execu number of buffers for current read during execute
exerow number of rows processed during execute
exemis number of library cache misses during execute
fchcnt number of times fetch was called
fchcpu cpu time spent fetching
fchela elapsed time fetching
fchdsk number of disk reads during fetch
fchqry number of buffers for consistent read during fetch
fchcu number of buffers for current read during fetch
fchrow number of rows fetched
userid userid of user that parsed the cursor

PRINT

只列出指定数量的已排序的 sql 语句，排序的条件参见 SORT 参数。
如果忽略此参数，tkprof 将跟踪文件中的所有的 sql 语句及其相关的

分析数据存放到输出文件中。
Print 与 sort 参数组合在一起，可以实现:
找出某一阶段耗费 cpu 最多的前 n 个 sql
找出某一阶段读硬盘最多的前 n 个 sql 等等。

INSERT

创建一个 sql 脚本文件，里面包含 create table 与 insert 语句。
利用这个脚本文件创建一个表及插入数据后，可以得到跟踪文件中
所有 sql 语句(包含 recursive SQL)的统计信息。如
,depth,user_id,
parse_cnt,parse_cpu,parse_elap,parse_disk,
parse_query,parse_current,parse_miss
,exe_count,exe_cpu,exe_elap,exe_disk,exe_query,
exe_current,exe_miss,exe_rows
,fetch_count,fetch_cpu,fetch_elap,fetch_disk,
fetch_query,fetch_current,fetch_rows,ticks
,sql_statement。
利用这些信息，也可以发现有问题的 sql。即是格式化好的输出文件中
有关 sql 性能信息数据的数据库表的形式。
我一般不用该参数

RECORD

创建一个包含客户端程序发出的所有的 sql 语句的脚本文件。
注意，并不包含 recursive SQL 。想知道它的用处吗？
对了可以窥探别人程序是如何访问数据库的，从而对了解程序的访问流程。
此时，最好不用 sort 参数，这样就可以按先后发出的顺序的到 sql.

例子 1：

将跟踪文件"dsdb2_ora_18468.trc"进行分析，并将其格式的结果放到"dsdb2_trace.out"文件中：

TKPROF dsdb2_ora_18468.trc dsdb2_trace.out SYS=NO EXPLAIN=SCOTT/TIGER

上面的例子中：
EXPLAIN 参数让 TKPROF 程序连接到 SCOTT 用户，然后用 EXPLAIN PLAN 命令给跟踪文件中的 sql 语句产生执行计划。SYS 参数的值为 NO，这样 TKPROF 就会忽略该跟踪文件中的 recursive SQL。

例子 2：

TKPROF DLSUN12_JANE_FG_SVRMGR_007.TRC OUTPUTA.PRF
EXPLAIN=SCOTT/TIGER TABLE=SCOTT.TEMP_PLAN_TABLE_A
INSERT=STOREA.SQL SYS=NO SORT=(EXECPU,FCHCPU)

注意上面的所有命名应该都在一行中，否则需要有换行符。

上面的例子中：

TABLE 参数使 TKPROF 使用 scott 用户下的 TEMP_PLAN_TABLE_A 表作为临时存放 sql 执行计划的表。
INSERT 参数使 TKPROF 产生一个名为 STOREA.SQL 的脚本文件，存放所有被跟踪的 sql 语句的统计数据。
SORT 参数使 TKPROF 先按照 sql 语句使用的 cpu 执行时间与该语句 fetch 操作使用的 cpu 时间进行排序，然后将其写到输出文件中。

解释 tkprof 程序产生的格式化文件：

tkprof 的格式化输出文件主要包含两大部分：
header
body
summary

header：
主要包括一些描述信息，如 TKPROF 的版本、运行时间，各个统计项的描述。如：

TKPROF: Release 8.1.7.0.0 - Production on 星期四 6 月 30 13:10:59 2005
(c) Copyright 2000 Oracle Corporation. All rights reserved.
Trace file: D:\oracle\admin\xyj\udump\ORA01720.TRC
Sort options: default

```
********************************************************************************
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
********************************************************************************
```

body：
是我们主要关心的地方，有我们感兴趣的信息。如 sql 语句、sql 语句的统计信息、sql 语句的执行计划等。如

```
select *
from
  emp
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	3	0.00	0.00	1	0	1	0
Execute	3	0.00	0.00	0	0	0	0
Fetch	6	0.00	0.00	1	6	12	36

```
------- -----  ----- ------- ----- ------ --------  -----
total       12      0.00      0.00      2       6         13       36

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 19  (SCOTT)

Rows     Row Source Operation
-------  ---------------------------------------------------
     12  TABLE ACCESS FULL EMP

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
12    TABLE ACCESS (FULL) OF 'EMP'


DELETE FROM RM$HASH_ELMS
call       count     cpu       elapsed    disk    query      current    rows
-------    ------    ------    --------   -----   ------     --------   ----
Parse          2     0.00       0.00        0       0          0        0
Execute       29    12.04      12.61     6786    6853        108       19
Fetch          0     0.00       0.00        0       0          0        0
-------    ------    ------    --------   -----   ------     --------   ----
total         31    12.04      12.61     6786    6853        108       19

Misses in library cache during parse: 0
Optimizer hint: CHOOSE
Parsing user id: 9  (DES12A)   (recursive depth: 3)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  DELETE STATEMENT   HINT: CHOOSE
     16   TABLE ACCESS (FULL) OF 'RM$HASH_ELMS'
```

summary：

对所有的 sql 语句各个执行阶段的统计数据的汇总：

OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS - - 标明是用户发出的 sql 语句的统计数据的汇总

```
call       count       cpu        elapsed    disk    query    current     rows
-------    ------      --------   --------   -----   ------   --------    -----
Parse          7        0.00       0.00        2      201         2          0
```

Execute	7	0.00	0.00	1	2	7	1
Fetch	10	0.00	0.00	2	67	16	52
---	---	---	---	---	---	---	---
total	24	0.00	0.00	5	270	25	53

Misses in library cache during parse: 5

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS - - 标明是系统发出的 sql 语句的统计数据的汇总

call	count	cpu	elapsed	disk	query	current	rows
---	---	---	---	---	---	---	---
Parse	75	0.00	0.00	2	3	2	0
Execute	81	0.00	0.00	1	1	5	1
Fetch	153	0.00	0.00	21	355	0	110
---	---	---	---	---	---	---	---
total	309	0.00	0.00	24	359	7	111

Misses in library cache during parse: 29

```
    8  user    SQL statements in session.
   74  internal SQL statements in session.
   82  SQL statements in session.
    5  statements EXPLAINed in this session.
********************************************************************************
Trace file: D:\oracle\admin\xyj\udump\ORA01720.TRC
Trace file compatibility: 8.00.04
Sort options: default

       1  session in tracefile.
       8  user    SQL statements in trace file.
      74  internal SQL statements in trace file.
      82  SQL statements in trace file.
      34  unique SQL statements in trace file.
       5  SQL statements EXPLAINed using schema:
            SCOTT.prof$plan_table
              Default table was used.
              Table was created.
              Table was dropped.
     825  lines in trace file.
```

现只对 body 中的每部分作出详细说明：

1) sql 语句的统计信息

我们把 select 语句的执行过程分成 3 个阶段(分析→ 执行→ 取结果)，把 DML 语句的执行分成 2 个阶段(分析→ 执行)。格式化的输出文件中列出了 sql 语句执行的每一阶段所耗费资源统计信息，对于这些信息，是以行列的模式给出的。每一行代表每个 sql 执行中的具体某一阶段所耗费的相应资源。这样通过分析每个阶段耗费的资源，可以知道哪个 sql 有问题，并进一步知道该 sql 执行过程中哪一阶段出现了问题，从而快速定位问题，进而迅速解决问题。

下面对每个列进行说明：

call : 表示 sql 语句执行的每个阶段，每个 sql 语句的活动被分成以下 3 部分：

Parse: 语句被解析的次数，主要是生成执行计划。包含 hard parse 与 soft parse。
需要做的工作：权限检查，表、列、引用的表的存在性检查；
比较执行计划，选出最好的一个等等。

Execute: 真正执行语句的时间，对于 DML 语句，在此阶段中修改数据；
对于 select 语句，这步只是标识出查询出的行。

Fetch : 只对 select 语句有效，DML 语句的执行并没有该阶段

其它列的值都是与 sql 执行三个阶段中所耗费的资源的统计值

COUNT

一个语句被 parsed、executed、fetched 的次数

CPU

执行这个语句的每个阶段耗费的 cpu 时间

ELAPSED

执行这个语句的每个阶段耗费的总时间(包括上面的 cpu 时间与其它时间，如读数据)

DISK

每个阶段读硬盘的次数(有可能从 file system buffer 中取得数据)
对于该参数，我们希望该值越小越好，如果该值很大，该 sql 需要调整，
建立相关索引或看是否正确的使用了索引

QUERY

每个阶段以 consistent mode 方式从数据库 buffer 中查询的 buffers 数。
对于查询，其 buffer 一般都是以 consistent mode 模式被读取

CURRENT

每个阶段以 current mode 方式从数据库 buffer 中查询的 buffers 数。Buffers are often
对于 DML 语句，需要的 buffer 是以 current mode 模式被读取的。
QUERY + CURRENT 的和是该 sql 语句总的存取的 buffer 数目

ROWS

这个 sql 语句最后处理的行数，不包括子查询中查询出来的行数。
对于 select 语句，该值产生于 fetch 阶段；对于 dml 该值产生于 execute 阶段。

因为统计耗费的时间时，最小的计量单位为 0.01 秒，所以如果得到一个阶段中耗费的时间为 0，并不表示这个阶段没有耗费时间，而是极可能说明这个阶段耗费的时间小于 0.01 秒，小于计量单位，数据库无法计时，只要以 0.00 表示

2) 与执行计划有关的内容

```
Misses in library cache during parse: 1      -- 说明 hard parse 的次数
Optimizer goal: CHOOSE                       -- 采用的优化器
Parsing user id: 19   (SCOTT)                -- 那个用户执行的该 sql

Rows     Row Source Operation
-------  ---------------------------------------------------
      12  TABLE ACCESS FULL EMP

Rows     Execution Plan                      -- 下面是真正的执行计划
-------  ---------------------------------------------------
       0  SELECT STATEMENT     GOAL: CHOOSE
12     TABLE ACCESS (FULL) OF 'EMP'
```

Misses in library cache during parse: 这个统计值是一个比较重要的指标，如果该值与该语句的 parse 统计值基本相等，并且该值比较大，而且该 sql 的 parse 阶段耗费的资源比较多，则说明你的语句应该采用 bind variable 模式。

执行计划部分也比较重要，它能看出查询是否用了索引，和各种关联操作所采用的方法。

建议用 autotrace 进行跟踪而不是用上面方法进行跟踪的原因：

虽然上面的方法给出了 sql 语句的执行计划、统计数据等信息，但是因为 tkprof 的格式化输出不给出详尽的 costs 与 statistics 信息，这使我们在利用格式化输出判断系统到底是使用基于代价的优化方法还是基于规则的优化方法时，感到很迷茫，我们不能明确的知道到底系统在使用那种优化器。

但是仅仅因为这个原因就使我们放弃上面跟踪方法而该用 autotrace 功能是不够的，因为：
1) 到底系统是使用基于规则的还是基于代价的优化器我们可以借助与其它信息来识别，而且从 10G 以后，oracle 就宣布不再使用基于规则的优化器了。
2) 上面的跟踪办法能实现 autotrace 不能完成的功能，因为 autotrace 只能跟踪本会话，而不能跟踪其它会话，这样假如有一个正在运行的程序出现了性能问题，我们就不能使用 autrace 去跟踪 sql 的执行情况。

下面以一个具体的例子的数据说明如何利用格式化的输出文件进行 sql 的调整：

第一步： - 现看格式化输出文件最后部分，即汇总部分

```
========================================================================
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

| call      | count   |   cpu   | elapsed |       disk |    query | current   |     rows |
|---------  |-------  |------   |---------  |---------   |--------  |---------  |--------  |
| Parse     | [A] 7   | 1.87    |      4.53 |     385    | [G] 553 |         22 |          0 |
| Execute | [E] 7   | 0.03    |      0.11 | [P]    0   | [C]    0 | [D]    0   | [F]   0    |
| Fetch     | [E] 6   | 1.39    |      4.21 | [P] 182   | [C] 820 | [D]    3   | [F] 20    |
```

Misses in library cache during parse: 5
Misses in library cache during execute: 1

      8 user SQL statements in session.
     12 internal SQL statements in session.
[B] 54 SQL statements in session.
      3 statements EXPLAINed in this session.

(1). 通过比较 [A] 与 [B]，我们可以发现是否有过量的 parsing 现象。在上面的例子中，我们可以看到在 session 中执行了 54 个语句，但是只有 7 次 parses，所以这是比较正常的，没有过量的 parse 现象。

(2). 利用 [P], [C] & [D] 来决定数据库高速缓存的命中率问题
Hit Ratio is logical reads/physical reads:
Logical Reads = Consistent Gets + DB Block Gets
Logical Reads = query + current
Logical Reads = Sum[C] + Sum[D]
Logical Reads = 0+820 + 0+3
Logical Reads = 820 + 3
Logical Reads = 823

Hit Ratio = 1 - (Physical Reads / Logical Reads)
Hit Ratio = 1 - (Sum[P] / Logical Reads)
Hit Ratio = 1 - (128 / 823)
Hit Ratio = 1 - (0.16)
Hit Ratio = 0.84 or 84%

(3). 我们希望 fetch 的次数要比 rows 小，即一次 fetch 可以取多行数据(array fetching)，
可以我们可以更高效的取得查询数据。
这可以通过比较[E]与[F].

[E] = 6 = Number of Fetches
[F] = 20 = Number of Rows

从上面的信息中我们可以看到，6 次 fetch 总共取了 20 行数据，结果不是很坏。如果使用了经过良好配置 arrayfetching，则可以用更少的 fetch 次数取到同样数量的数据，性能会更好。

(4). [G] 表示为了对语句进行分析，读数据字典告诉缓存的次数
- 这个参数对性能的影响不大，一般不用关心。而且这个统计值一般不是我们可以控制的。

第二步 – 检查耗费大量资源的语句

==========

update ...

where ...

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	7	122	0	0	0	0
Execute	1	75	461	5	[H] 297	[I] 3	[J] 1
Fetch	0	0	0	0	0	0	0

[H] 表明需要访问 297 个数据块才能找到我们需要修改的数据。

[I] 表明我们的修改操作才修改 3 个数据块中的数据

[J] 表明我们只修改了一行数据(其它数据块的修改应为 undo、redo 信息)

为了修改一行数据而要搜寻 297 个数据块。

考虑是否需要在查许的列上建一个索引!

第三步 – 查看是否有过量的 parse 现象

==========

select ...

call	count	cpu	elapsed	disk	query	current	rows
Parse	[M] 2	[N] 221	329	0	45	0	0
Execute	[O] 3	[P] 9	17	0	0	0	0
Fetch	3	6	8	0	[L] 4	0	[K] 1

Misses in library cache during parse: 2 [Q]

[K] 表明这个查询只返回一行数据

[L] 表明我们需要 fetch 4 次才能得到数据，这是正常的，因为需要额外的 fetch 操作以便检查是否 fetch 到 cursor 的最后，当然还可能有其它 fetch 开销。

[M] 表明我们进行了两次 parse(包含 hard parse 与 soft parse) – 这是我们不想看到的，特别是当 parse 阶段操作耗费 cpu 资源比 execute 阶段耗费的 cpu 资源([O] & [P]) 多得多得时候。 **[Q]** 表明这两个 parse 操作都是 hard parse。**如果[Q]的值为 1，这这个语句有一个 hard parse，然后跟着一个 soft parse(仅仅从库缓存中得到上次分析的信息，比 hard parse 要高效的多)。**

对上面的例子来说，结果并不是特别的坏，因为该语句只执行 2 次，然而如果对于频繁执行的 sql 来说，如果几乎每次执行都需要 hard parse，则结果就会变的很坏，此时我们说该语句有过量的 parse 现象(excessive parsing)。

o 解决该问题的方法：

- 使用 bind variables
- 使 shared pool 足够大，从而在内存中容纳你执行过的每一条语句，以便下一次可以重用该语句。但这种方法治标不治本，在繁忙的系统中有时会引起 ora-04031: unable to allocate %s bytes of shared memory (%s,%s,%s) 错误。
- 使用 8i 新引入的参数 cursor_sharing，建议在经过测试后再使用该参数，因为有时使用该参数后会引起系统性能下降

如何降低 parse 阶段使用的 cpu 时间

1.Rewrite the application so statements do not continually reparse.

2.Reduce parsing by using the initialization parameter SESSION_CACHED_CURSORS.

3.Use bind variables to reduce parsing.

注意：

记住如果 cursor 没有被关闭，将在 tkprof 的输出文件中看不到任何该 sql 的输出。设置 SQL_TRACE = false 并不能关闭 PL/SQL 的 child cursors，所以要在自己的存储过程中养成及时关闭显式 cursor 的习惯。令我们高兴的是，在 SQL*Plus 中，语句一旦执行完毕，该语句对应的 cursor 也自动关闭了。

参考信息：

1. METALINK