



Oracle中 SQL语句的性能调整

GuoHong Zhou



课程安排



一、应遵循的调整方法



目的

- 了解性能管理流程
- 描述性能问题的原因
- 描述性能调整方法
- 解释按照顺序调整性能的优势
- 明了开发人员的性能调整责任和步骤



概述

- 性能管理
- 性能问题
- 调整方法论
- **SQL**语句调整
- 应用方法



性能管理

- 及早开始
- 设定明确调整目标
- 进行调整并监控
- 团队工作
 - **DBA**、系统管理员、开发人员
- 处理意外的变化



性能管理因素

- **Schema**
 - 数据设计、索引的使用
- 应用程序
 - **SQL**语句、流程控制代码
- 数据库实例
- 数据库文件
- 对性能合理的期望



性能问题

- 可消耗资源的不足
 - CPU、I/O、内存、网络
- 设计上资源使用的不合理
- 锁资源



关键资源

- 性能依赖于下列因素
 - 需要该资源的客户端数目
 - 等待该资源的时间
 - 使用该资源的时间
- 限制资源需求以取得反应时间



对资源的过度请求

- 增加反应时间，降低处理能力
- 尽量避免以维持合适的反应时间



系统的可扩充性

- 系统处理更多工作的能力
 - 可能增加对资源的需求
- 由系统设计者和性能调整专家考虑
- 不好的可扩充性导致
 - 负荷增加时资源耗尽而无法增加处理能力



系统的可扩充性

- 系统资源耗尽源于：
 - CPU资源耗尽
 - 大量的表扫描导致的I/O冲突
 - 过度的数据交流导致网络拥塞
 - 过多的进程或线程分配导致OS调度不及



系统设计对可扩充性的影响

- 不好的Schema设计
 - 导致昂贵的SQL
- 不好的事务处理设计
 - 导致锁资源冲突
- 不好的连接管理
 - 导致相应速度减慢和系统的不可靠



性能调整方法

- 调整业务功能
- 调整数据设计
- 调整过程设计
- 调整SQL语句
- 调整物理结构
- 调整内存分配
- 调整I/O
- 调整内存冲突
- 调整OS



调整工作的角色分配

- 业务分析员
 - 调整业务功能
- 系统设计人员
 - 调整数据设计
 - 调整过程设计
- 应用开发人员
 - 调整SQL语句
 - 调整系统物理结构



调整工作的角色分配

- 数据库管理人员
 - 调整内存分配
 - 调整I/O
 - 调整内存冲突
- 操作系统管理人员
 - 调整OS
- 网络管理人员
 - 调整网络问题



调整SQL语句

- 调整Schema
 - 增加索引，如果需要的话
 - 适当考虑**Index Organized Table**
 - 建立**Cluster**
- 选择语言工具
 - **SQL**
 - **PL/SQL**



调整SQL语句

- 设计SQL的重用优化
- 设计并调整SQL语句
- 利用Oracle优化器使性能最优



应用调整方法

- 主动调整：由上至下
- 如果只能被动调整：由下至上
 - 建立可量化目标
 - 建立可重复的测试环境
 - 避免任何推测
 - 测试并及时记录
 - 达到目标立即停止



小结

- 管理性能
 - 尽早开始：主动调整
 - 设定明确量化目标
 - 监控调整过程与目标是否一致
 - 恰当处理意外
- 认定性能问题
 - 不足够的资源
 - 不当的设计
 - 关键资源与过度需求



小结

- 调整SQL语句
 - 明确性能需求
 - 每一步骤都及时分析结果
 - 调整Schema
 - 重用SQL
 - 设计与调整SQL语句
 - 善用Oracle优化器



SQL语句的处理



目的

- 了解SQL语句的基本处理步骤
- 监控SQL共享区域
- 使用可共享SQL语句
- 了解使用`cursor_sharing`参数
- 管理PGA内存区

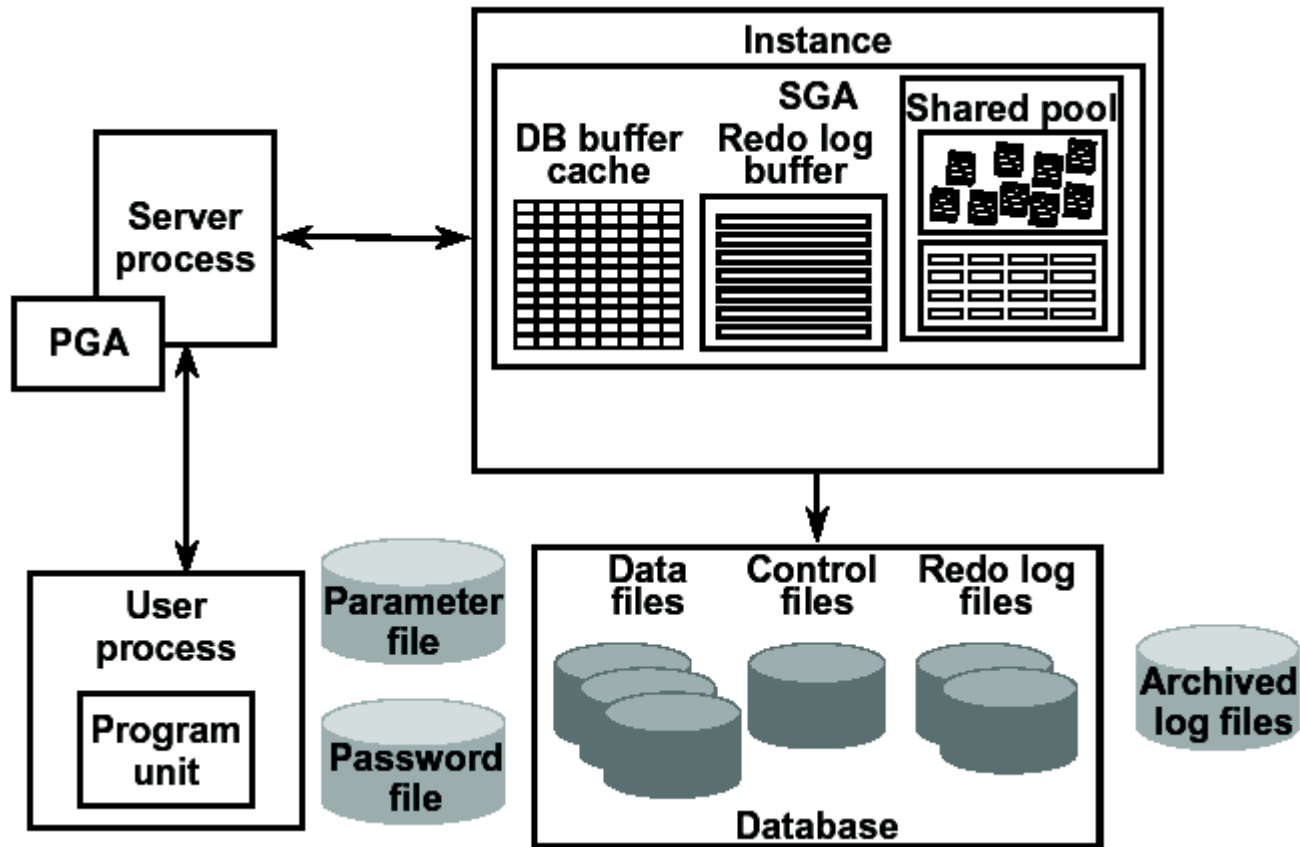


概述

- **System Global Area (SGA)**
- **SQL共享内存区**
- **Program Global Area (PGA)**
- **SQL语句处理阶段**
- **SQL语句编写标准**
- **Cursor共享**
- **PGA管理**



System Global Area (SGA)



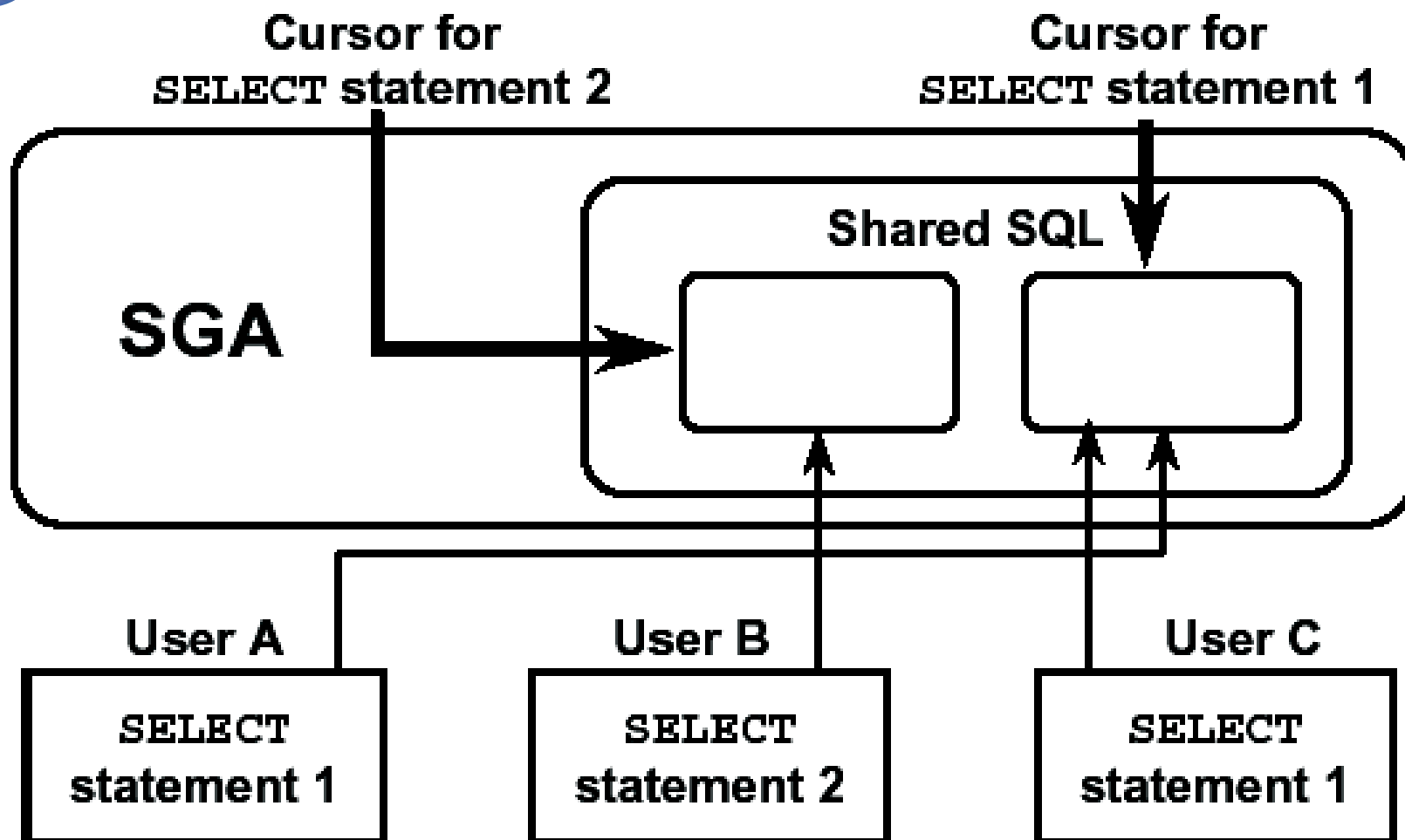


Shared Pool

- **Data Dictionary Cache (数据字典)**
- **Library Cache**
 - SQL语句
 - 经过编译的PL/SQL代码
 - Java类库



SQL共享内存区



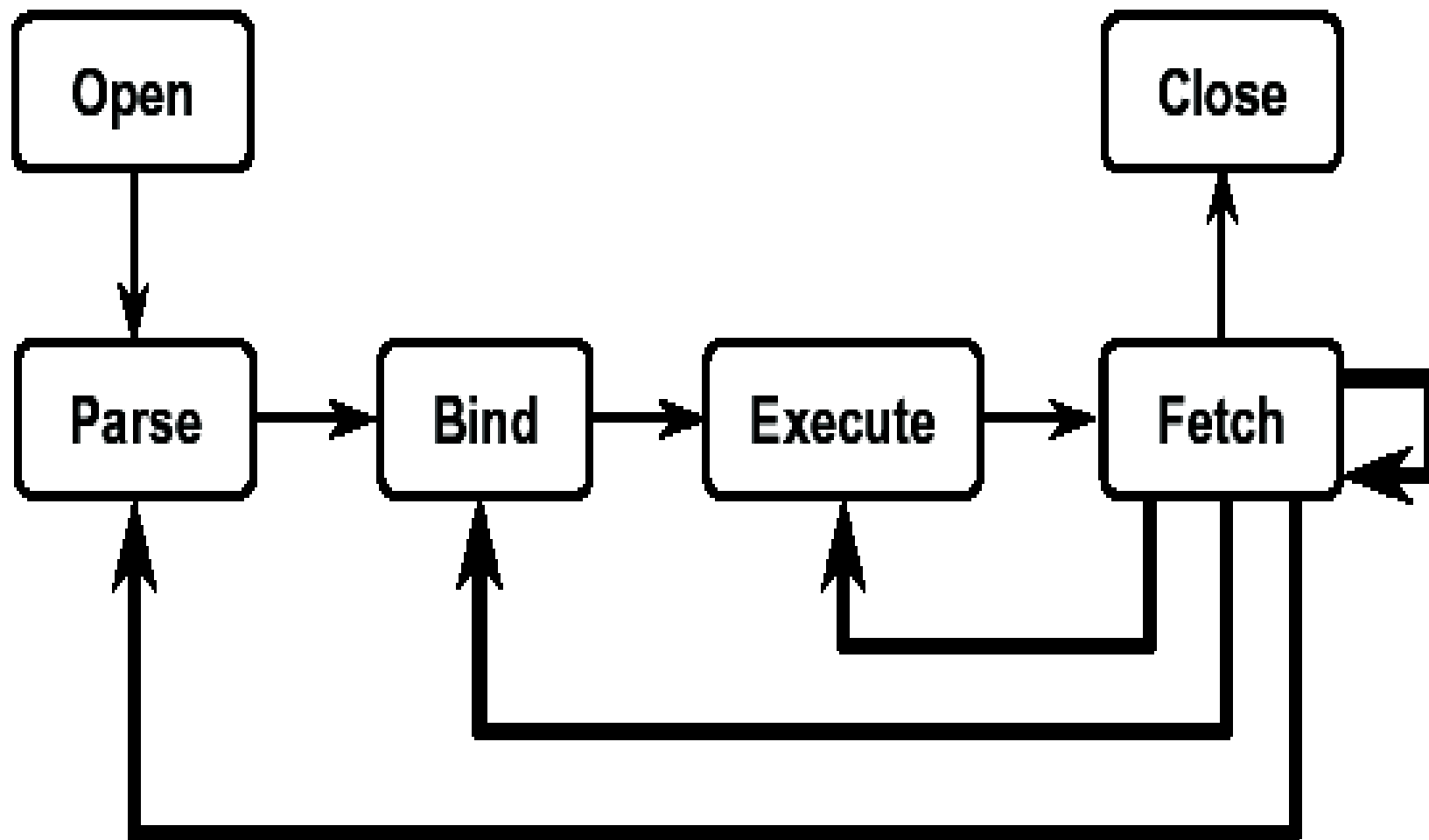


Program Global Area (PGA)

- **Shadow**进程中的缓存区
 - 包含数据和控制信息
- 可控制和调整工作区内存
 - **Optimal Size**
 - **One-pass Size**
 - **Multipass Size**



SQL语句处理阶段





SQL语句处理阶段

- **Parse**
 - 在**Shared Pool**中搜寻相同语句
 - 检查句法
 - 检查权限
 - 用子查询替代视图定义
 - 确定执行计划



SQL语句处理阶段

- **Bind**
 - 扫描语句查找**bind**变量
 - 给变量赋值
- **执行 execute**
 - 应用执行计划
 - 进行必要的**I/O**和排序动作



SQL语句处理阶段

- **Fetch**
 - 从查询获得结果（行）
 - 以排序的结果返回
 - 使用**array fetch**机制



共享SQL：优势

- 减少Parse
- 动态调整内存
- 增进内存使用效率



共享SQL：要求

- 相同的语句
 - 大小写相同
 - 单词间隔相同（空格等）
 - 注释都要相同
- 涉及到的数据库对象相同
- 捆绑变量类型相同



共享SQL

- 下面语句不同
 - **Select * from emp where empno=7788;**
 - **Select * From emp where empno=7788;**
- 如果SQL语句属于不同用户，则认为不同
 - **Select * from emp where empno=7788;**
 - **Select * from emp where empno=7788;**



Bind变量和共享cursor

```
select * from customers where cust_id = :c
```

```
select * from customers where cust_id = :d
```

如果:**c**和:**d**变量类型不同, 则语句不同

- 但命名不同没关系, Oracle重新内部表示
上述两个语句都会被内部表示成:

```
select * from customers where cust_id = :b1
```



编写共享SQL语句

- 使用存储过程和包 (**package**)
- 使用触发器 (**trigger**)
- 使用任何其他的库函数和过程
- 遵循标准：
 - 大小写
 - 空格与注释
 - 涉及数据库对象
 - **Bind**变量



监控SQL共享

- **V\$librarycache**
- 单独的SQL语句
 - **V\$sqltext**
 - **V\$sqltext_with_newlines**
 - **V\$sql_bind_data**
 - **V\$sql**
 - **V\$sqlarea**



V\$librarycache

NAMESPACE	The name of the library cache area
GETS	Total number of requests (lookups)
GETHITS	The number of times that an object's handle was found in memory
GETHITRATIO	The ratio of GETHITS to GETS
PINS	The number of objects in the library cache
PINHITS	The number of times that all the pieces of the object were found in memory
PINHITRATIO	The ratio of PINHITS to PINS
RELOADS	The number of library cache misses



V\$sqlarea

SQL_TEXT	Text of the SQL statement
VERSION_COUNT	Number of versions of this cursor
LOADS	Number of times the cursor has been loaded
INVALIDATIONS	Number of times the contents have been invalidated
PARSE_CALLS	Number of times a user has called this cursor
SORTS	Number of sorts performed by the statement
COMMAND_TYPE	Command type
PARSING_USER_ID	Parsing user ID (SYS = 0)



监控共享Cursor的使用

- 每个语句有一次**load**比较理想
- 每个版本有一次**load**也较好
- 每个版本多于一次**load**表明可能应该增加**shared pool**的大小



监控共享Cursor的使用

```
select sql_text, version_count, loads,  
       invalidations, parse_calls, sorts  
from   v$sqlarea  
where  parsing_user_id > 0  
and    command_type    = 3  
Order by sql_text;
```

sql_text	version count	loads	invali dations	parse calls	sorts
-----	-----	-----	-----	-----	-----
select * from customers where CUST_ID = 180	1	2	0	3	0
select * from customers where cust_id = 180	1	1	0	1	0



共享Cursor

使用参数

Cursor_sharing=similar|force

- 如果SQL语句不共享只是由于使用常量
- 由于library cache命中率低而导致系统相应速度慢



小结

- **SQL处理的4个阶段**
 - **Parse, Bind, Execute, Fetch**
- **SQL语句的共享**
 - 遵循固定的编码标准
 - 使用**bind**变量
 - 设置**cursor_sharing**参数
 - 利用存储过程



Explain和Autotrace



目的

- 使用**plan_table**
- 利用**explain plan**显示SQL的执行情况
- 使用**SQL*Plus**的**autotrace**监控SQL语句的执行计划和统计数据



建立plan_table

- 执行脚本建立
 - **\$ORACLE_HOME/rdbms/admin/utlxplan.sql**
 - 该脚本只是建立**plan_table**表



Explain plan命令

```
➤➤ EXPLAIN PLAN ➤➤  
    [ SET STATEMENT_ID  
      = 'text'  
    ]  
➤➤  
    [ INTO your plan table ]  
➤➤  
➤➤ FOR statement ➤➤
```




Explain plan例子

```
SQL> EXPLAIN PLAN
      2  set statement_id = 'demo01' for
      3  select *
      4  from    products
      5  where   prod_category = 'Men'
      6  and    prod_subcategory = 'Jeans - Men';
```

Explained.

注：explain plan命令并不真正执行SQL语句



显示执行计划

```
SQL> column "Query Plan" format a60

SQL> select  id
  2      ,    lpad(' ', 2*level) || operation
  3          || decode(id,0,' Cost = ' || position)
  4          || ' ' || options
  5          || ' ' || object_name as "Query Plan"
  6 from    plan_table
  7 where   statement_id = 'demo01'
  8 connect by prior id = parent_id
  9 start   with id = 0;
```

注：在8i中可使用脚本\$O_H/rdbms/admin/utlxpls.sql



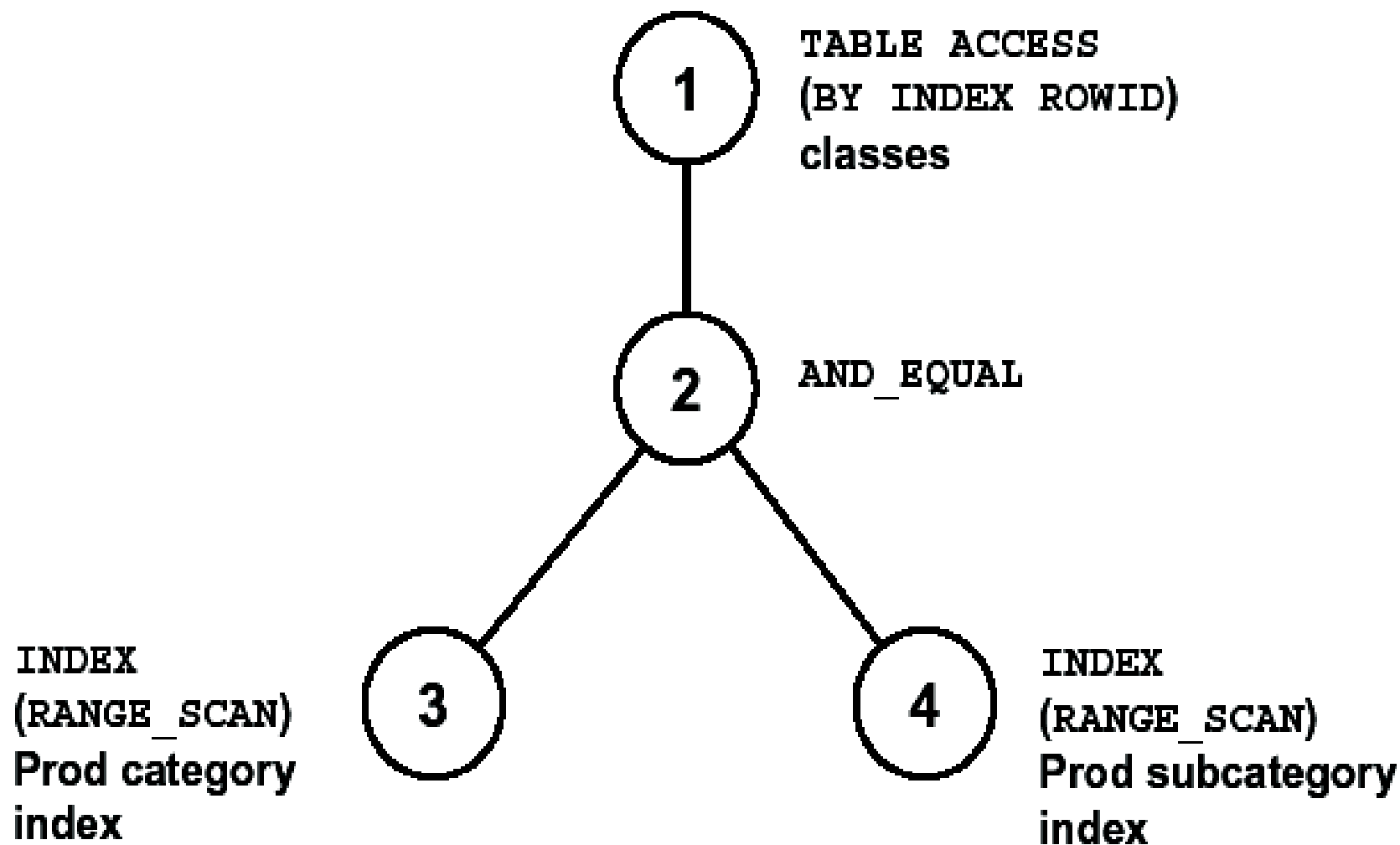
显示执行计划

ID Query Plan

```
-----  
0  SELECT STATEMENT Cost =  
1  TABLE ACCESS BY INDEX ROWID PRODUCTS  
2  AND-EQUAL  
3  INDEX RANGE SCAN PRODUCTS_PROD_CAT_IX  
4  INDEX RANGE SCAN PRODUCTS_PROD_SUBCAT_IX
```

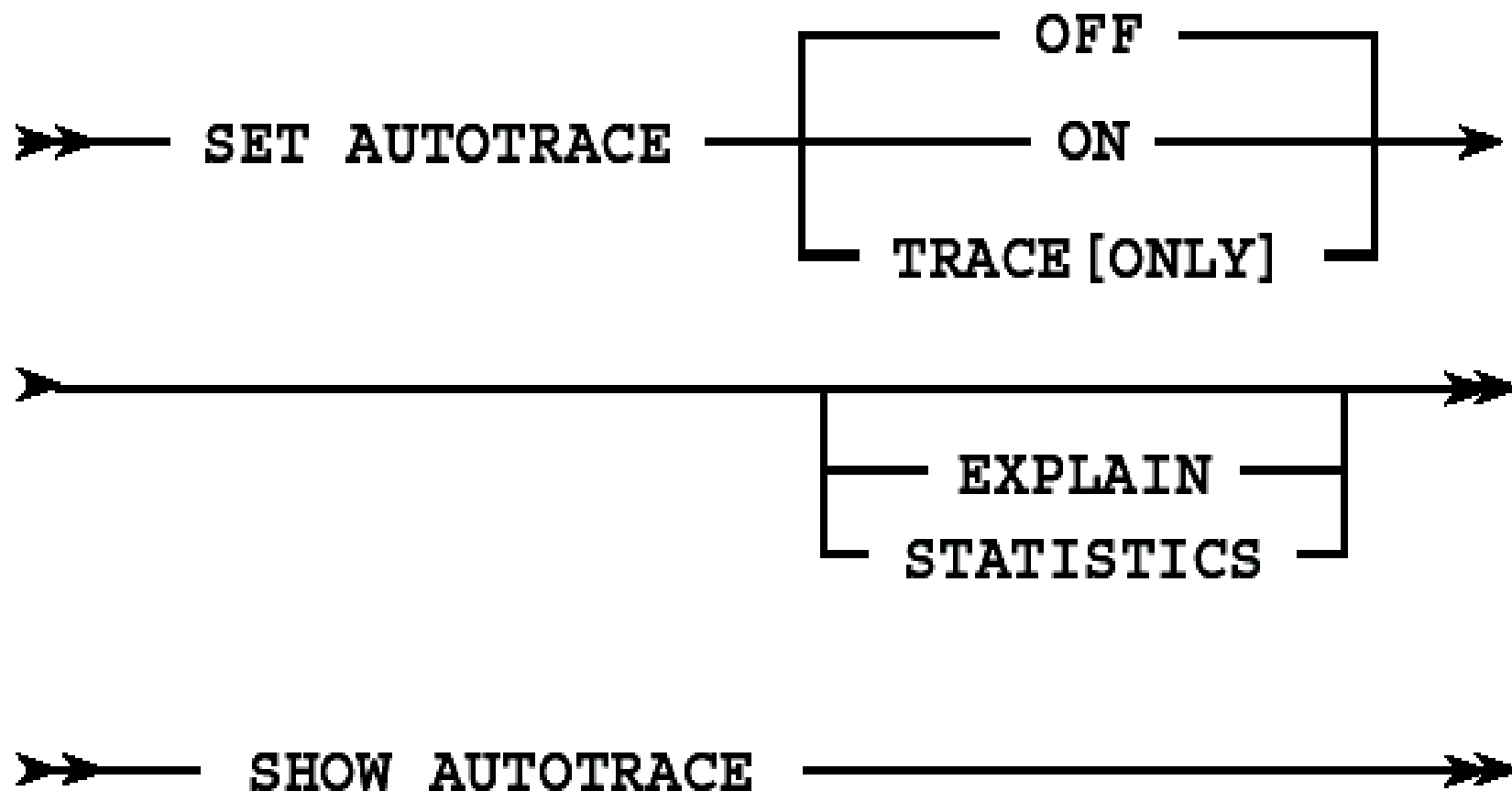


显示执行计划





Sql*plus的autotrace





Sql*plus autotrace举例

- 开始跟踪语句：
 - **Set autotrace on**
- 隐藏SQL语句执行结果（执行SQL语句）
 - **Set autotrace traceonly**
- 只查看执行统计数据（执行SQL语句）
 - **Set autotrace traceonly statistics**
- 只查看执行计划（不执行SQL语句）
 - **Set autotrace traceonly explain**



Sql*plus autotrace的统计数据

```
set autotrace traceonly statistics
select * from dual;
```

Statistics

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



小结

- 建立**plan_table**
- 使用**explain plan**命令来查看执行计划
- 利用**sql*plus**的**autotrace**设置查看SQL执行计划和统计数据



SQL Trace 和 tkprof



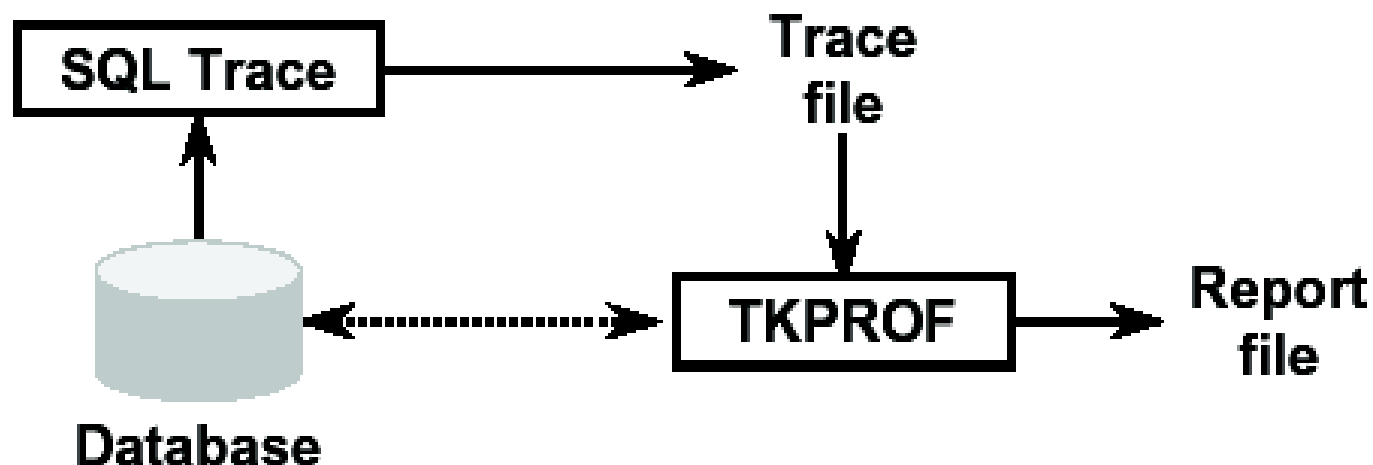
目的

- 配置SQL Trace来统计session数据
- 使用SQL Trace并查找trace文件
- 使用tkprof格式化trace文件
- 解释tkprof的结果



SQL Trace

- 可以在**session**一级或**instance**一级设定
- 收集**SQL**语句的统计数据
- 产生可以用**tkprof**工具格式化的跟踪文件





使用SQL Trace

- 在**init**参数文件设定
- 或打开**SQL Trace**开关
- 运行需跟踪的应用程序
- 关闭**SQL Trace**并格式化跟踪文件
- 解读跟踪结果



参数文件设定

- 跟踪整个instance全部SQL
 - **Timed_statistics = true**
 - **Max_dump_file_size = n**
 - **User_dump_dest = <dir>**
 - **Sql_trace = true**



打开SQL Trace

- 跟踪本session

SQL> alter session set sql_trace=true;

SQL> exec dbms_session.set_sql_trace(true);

- 跟踪其他session

**SQL> exec dbms_system.set_sql_trace_in_session
 (*session_id*, *serial#*, *true*);**



查找跟踪文件

- 到参数`user_dump_dest`指定目录下查找最新产生的`trace`文件
- 一般名为`ora_xxxxx.trc`
 - `Xxxxx`指进程号



格式化trace文件

- 句法

**\$ tkprof <tracefile name> <outputfile name>
<options>**

- 举例

**\$ tkprof ora_8339.trc 8339.out
explain=scott/tiger sys=no sort=execpu**



Tkprof命令的结果

- 被跟踪的SQL语句
- 系统处理SQL语句的统计数据

PARSE	Translates the SQL statement into an execution plan
EXECUTE	Executes the statement (This step modifies the data for INSERT, UPDATE, and DELETE statements.)
FETCH	Retrieves the rows returned by a query (Fetches are performed only for SELECT statements.)



Tkprof命令的结果

- 7项统计数据

Count	Number of times procedure was executed
CPU	Number of seconds to process
Elapsed	Total number of seconds to execute
Disk	Number of physical blocks read
Query	Number of logical buffers read for consistent read
Current	Number of logical buffers read in current mode
Rows	Number of rows processed by the fetch or execute



Tkprof命令的结果

- 同时也包括下列信息
 - **Recursive SQL**语句
 - **Library cache**命中情况
 - 执行**parse**动作的用户标识
 - 执行计划
 - 优化器模式或**hint**



Tkprof输出举例：没用索引

```
...
select cust_first_name, cust_last_name, cust_city, cust_state_province
from customers
where cust_last_name = 'Smith'

call      count          cpu    elapsed      disk      query    current    rows
-----
Parse      3         801.14      800.00        0         0         1         0
Execute    3          0.00        0.00        0         0         0         0
Fetch     21       1802.59     9510.00       48       4835        10       234
-----
total      27       2603.73    10310.00       48       4835        11       234

Misses in library cache during parse: 3
Optimizer goal: CHOOSE
Parsing user id: 44

Rows      Row Source Operation
-----
       78  TABLE ACCESS FULL CUSTOMERS
...
```



Tkprof输出举例：唯一性索引

```
select cust_first_name, cust_last_name, cust_city, cust_state_province
from customers
where cust_last_name = 'Smith'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	7	100.14	1500.00	2	87	0	78
total	9	100.14	1500.00	2	87	0	78

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: 44

Rows	Row Source Operation
78	TABLE ACCESS BY INDEX ROWID CUSTOMERS
78	INDEX RANGE SCAN (object id 32172)



Tkprof结果解释：一些误区

- 一致性读陷阱
 - 访问被其他Session改变的数据块时
- Schema陷阱
- 时间陷阱
 - 等待锁资源的影响
- 触发器陷阱
 - 触发器和recursive SQL的影响



小结

- 设置SQL Trace的初始化参数
 - Sql_trace, timed_statistics
 - User_dump_dest, max_dump_file_size
- 打开一个session的SQL Trace

```
ALTER SESSION set sql_trace = true  
dbms_session.set_sql_trace(...)  
dbms_system.set_sql_trace_in_session(...)
```

- 用tkprof格式化trace文件并解读



Rule Based Optimizer

还是

Cost Based Optimizer



目的

- 描述**Oracle**优化器的功能
- 分辨**RBO**和**CBO**的不同
- 确认**RBO**和**CBO**的使用环境
- **CBO**考虑的因素
- 影响**RBO**的因素
- 在**instance**或**session**一级设定使用的优化器



概述

- 基于规则的优化器 (**RBO**)
- 基于成本的优化器 (**CBO**)
- 设定使用哪种优化器
- **RBO**的规则级别
- 影响**RBO**的缺省动作



优化器功能

- 评估表达式和条件
- **SQL**语句的等效转换
- 确定数据访问路径
- 决定如何做表联合 (**Table Joins**)
- 确认最佳的访问路径



基于规则的优化器（RBO）

- 支持的最后版本为**9i**
 - 出于和以前版本兼容目的
- 由语句的写法决定访问路径
- 不使用数据库对象的统计数据
- 不计算不同访问路径的花费



基于成本的优化器（CBO）

- 从**Oracle7**版本开始使用
- 由统计数据决定访问路径
- 由计算成本做出决定
 - 逻辑I/O次数
 - 网络传输次数
- 在不断发展中
- 通常性能好于**RBO**
- 估算**CPU**的使用



选择RBO还是CBO?

- **Oracle的缺省动作**
 - 当没有数据库对象的统计数据可用时，**RBO**
 - 只要涉及到对象中的任何一个存在统计数据时，使用**CBO**
- 使用**RBO**还是**CBO**可以在下列层面上决定
 - **Instance**级别
 - **Session**级别
 - **SQL**语句级别



设置优化器

- 在**instance**级别， 设置参数

Optimizer_mode =

{choose | rule | first_rows | all_rows}

- 在**session**级别

Alter session set optimizer_mode =

{choose | rule | first_rows | all_rows}



First_rows优化方法

- **Instance级别**
 - **Optimizer_mode = first_rows**
- **Session级别**
 - **Alter session**
set optimizer_mode = first_rows;
- **SQL语句级别**
 - **Hint — /*+ first_rows */**



RBO

- 无条件使用索引
- 使用内置的优先级别决定访问路径
- 比较难以控制
 - 通常改写SQL语句来达到不使用索引目的



RBO内置优先级别

Rank	Access Path
1	Single row by ROWID
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite index
9	Single-column index
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort-merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed column
15	Full table scan



RBO举例

```
SQL> select cust_last_name  
2   from customers  
3   where country_id = 'FR'  
4   and cust_credit_limit between 11000 and 15000  
5   order by cust_credit_limit;
```

- **RBO**有下列访问路径可以使用
 - 全表扫描（**full table scan**），级别**15**
 - **Order by** 索引列，级别**14**
 - **Country_id**列索引等值扫描，级别**9**
 - **Cust_credit_limit**列索引范围扫描，级别**10**



控制RBO的缺省动作

- 在**from**子句中改变各个表出现的顺序
- 在**where**子句中改变条件的顺序
- 有意建立或删除索引
- 使用句法手段不使用索引
 - 数字列+0
 - 日期列+0
 - 字符列||‘ ’



小结

- **RBO**
- **CBO**
- **CBO**考虑的因素
- 设置优化器
- **RBO**内置操作优先级别
- 影响**RBO**的缺省动作



索引和基本的访问方法



目的

- 了解rowid
- 了解索引类型
- 了解索引和完整性约束的关系
- 了解索引和外键约束关系
- 了解基本的访问方法
- 监控索引使用
- **Cluster概述**



Rowid

- 表明一个数据行的地址
- 扩展的rowid和限制的rowid
- 每个表都有一个名为rowid的伪列

```
SQL> select rowid, c.*  
        from channels c;
```

ROWID	C	CHANNEL_DESC	CHANNEL_CLASS
-----	-	-----	-----
AAABQDAABAAAJ4yAAA	S	Direct Sales	Direct
AAABQDAABAAAJ4yAAB	T	Tele Sales	Direct
AAABQDAABAAAJ4yAAC	C	Catalog	Indirect
AAABQDAABAAAJ4yAAD	I	Internet	Indirect
AAABQDAABAAAJ4yAAE	P	Partners	Others



Rowid

- 格式（扩展的rowid）
 - **AAABQD** —数据对象号
 - **AAB** —相对文件号（表空间内唯一）
 - **AAAJ4Y** —数据块号
 - **AAA-AAE** —块内行号
- 操作
 - **Dbms_rowid**

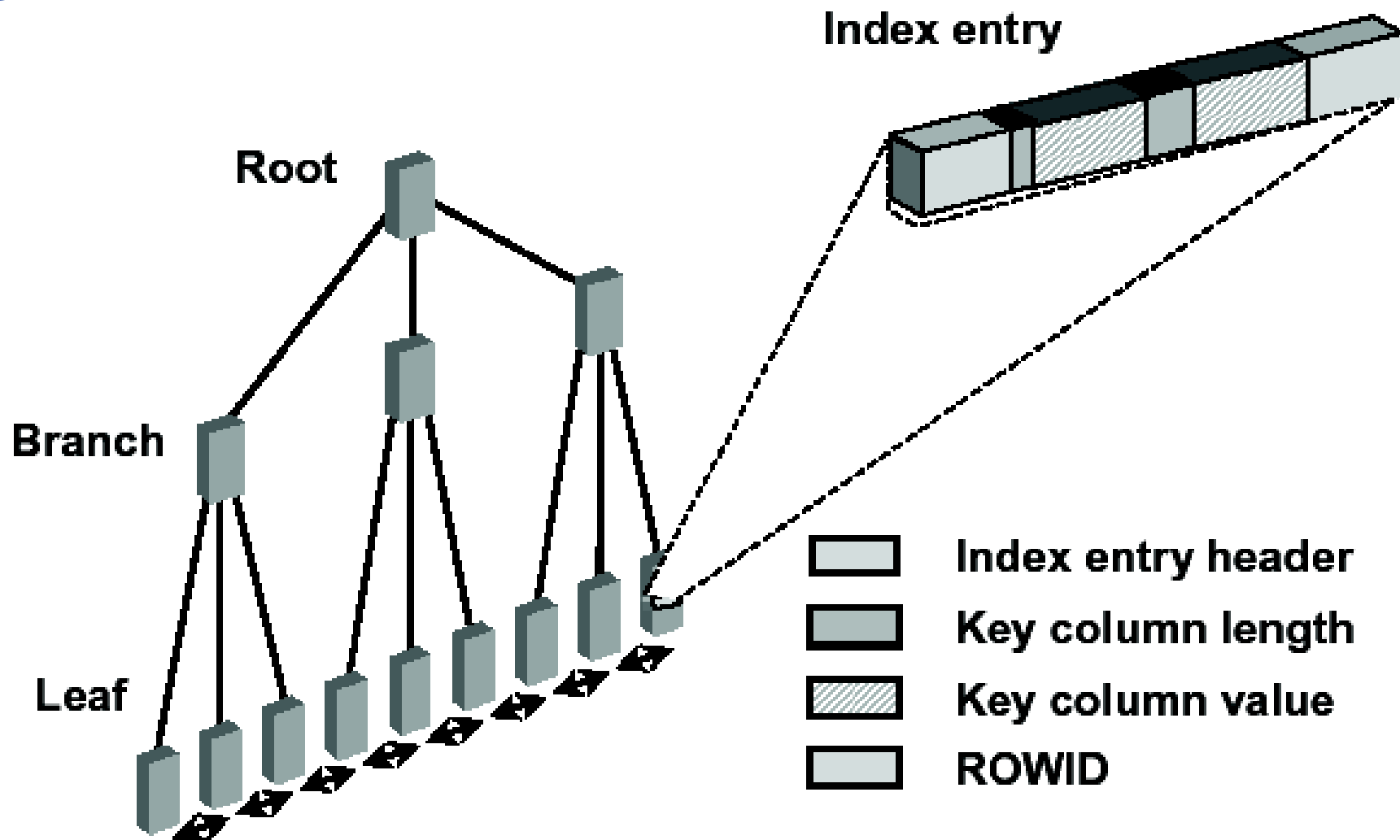


索引

- 唯一性和不唯一性索引
- 复合索引
- 索引的存储
 - **B*-tree**
 - Reverse Key
 - Descending
 - Function-based
 - **Bitmap**

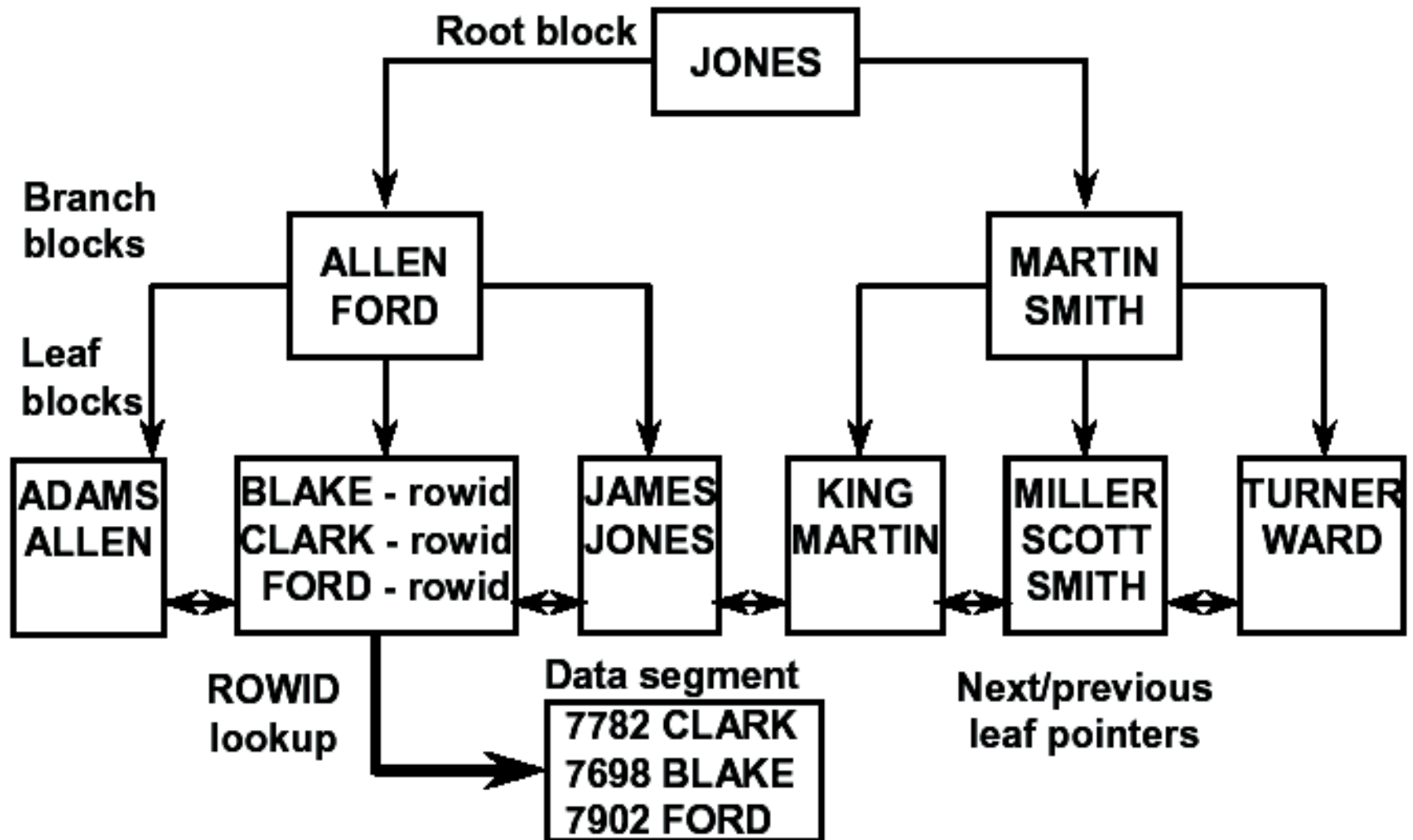


B*-tree索引





B*-tree索引举例





复合索引的使用

- 将最频繁访问的列放在前面
- 将最多限制的列放在前面
- 其余列添加到复合索引中



索引统计数据

- 索引统计信息可以由下面方法获得

```
SQL> analyze index index_name validate  
structure;
```

```
SQL> analyze index index_name estimate  
statistics;
```

```
SQL> select * from index_histogram;
```

```
SQL> select * from index_stats;
```



DML对索引的影响

- **Insert**操作导致索引对**leaf**块的**insert**操作
 - 可能发生**block split**
- **Delete**操作导致索引记录的删除操作
 - 空的索引块可以用来重新插入数据
- 对索引键值列的**update**导致对索引记录的删除和再插入操作



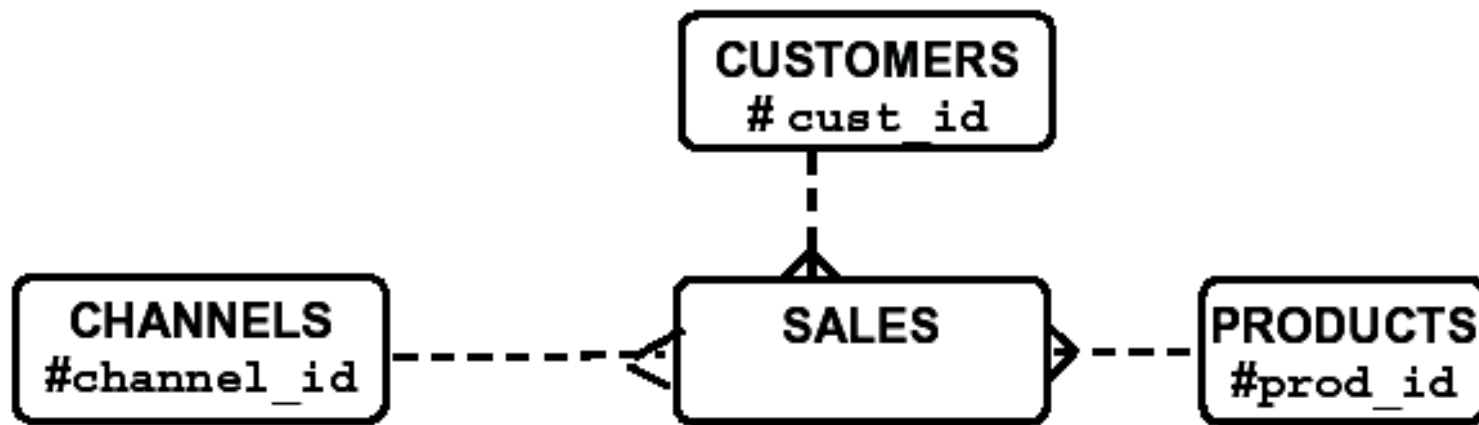
索引与约束（constraints）

- **Oracle**自动建立或使用**B*tree**索引当
 - 定义主键约束
 - 定义唯一性约束
- 开发人员应主动建立所有约束可能用到的索引
 - 而非依赖上述**Oracle**的自动行为



索引与外键约束

- 建立外键时没有索引自动建立
- 父表上的**DML**会对子表加锁





基本的访问方法

- 全表扫描
 - 使用**multiblock I/O**
 - 可以并行操作
- 索引扫描
 - 只访问索引
 - 伴随发生以**rowid**对表的访问
- **Fast full index scan**
 - 可以使用**multiblock I/O**
 - 可以并行操作



Cluster

- 不同表的行物理上存储在一起
- **Cluster**中的表是预联合在一起的
- 对用户和应用透明
- **DML**和全表扫描更加昂贵

Cluster举例



CUST_ID	CUST_LAST_NAME	CO	...
10	Kessel	UK	
20	Everett	US	
30	Odenwalld	US	
40	Sampson	US	
50	Nenninger	FR	
60	Rhodes	UK	

CO	COUNTRY_NAME	...
US	United States of America	
DE	Germany	
UK	United Kingdom	
NL	The Netherlands	



Unclustered CUSTOMERS and COUNTRIES tables

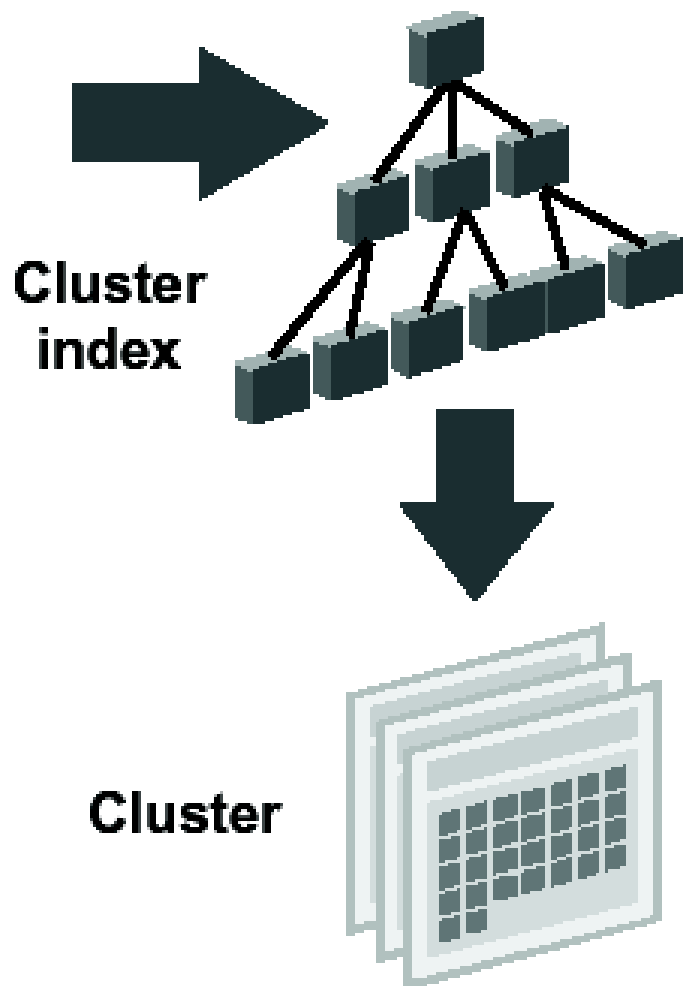
Cluster Key: COUNTRY_ID	
UK	United Kingdom
10	Kessel
60	Rhodes
280	Durby
650	Williamson
...	
US	United States of America
20	Everett
30	Odenwalld
40	Sampson
...	



Clustered CUSTOMERS and COUNTRIES tables



Index cluster



- 通过**Cluster index**访问**cluster**中的数据
- 包括**null**值



Index cluster性能特性

- 可以节省存储空间
- 减少物理I/O
- 缩短索引区段扫描的时间
- 增加**cluster**表的连接（**join**）性能



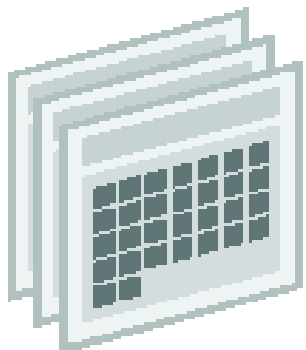
Index cluster的局限

- **DML性能不够好**
- **避免修改cluster键值**
- **Index cluster更加适合cluster键值分布均匀的情况**
- **Direct load时不能使用index cluster**

Hash cluster



Hash
function



- 使用**hash**函数
- 可以避免索引I/O
- 可能影响全表扫描的性能
- 在下列情况使用
 - 查询条件为 =操作时
 - 表很大时
 - 键值分布较均匀时



Hash cluster的局限

- 区域扫描不能使用**hash**函数
- **Cluster**键值的不均匀分布可能导致额外I/O
- 不可预知的键值可以导致**hash**函数的低效
- 全表扫描通常较慢



何时使用**cluster**

- 使用**cluster**
 - 当表主要操作为查询并且以**cluster**键值连接
 - 当**cluster**键值均匀分布
- **Cluster**可以减慢性能
 - 对于**DML**操作
 - 对于全表扫描操作



何时使用cluster

- 使用**index cluster**
 - 适当的物理数据结构
 - 不能使用**hash cluster**时
- 使用**hash cluster**
 - 当等值查询的性能是主要目标时
 - 当表很大时



小结

- **ROWID**
- 索引
 - 索引类型
 - **B*-tree**索引结构
 - **DML**与索引
 - 索引与完整性约束



小结

- 基本访问方法
 - 全表扫描
 - **B*-tree**索引扫描
 - 由**ROWID**访问
 - **Fast full index scan**
 - **Cluster**概述



收集统计数据

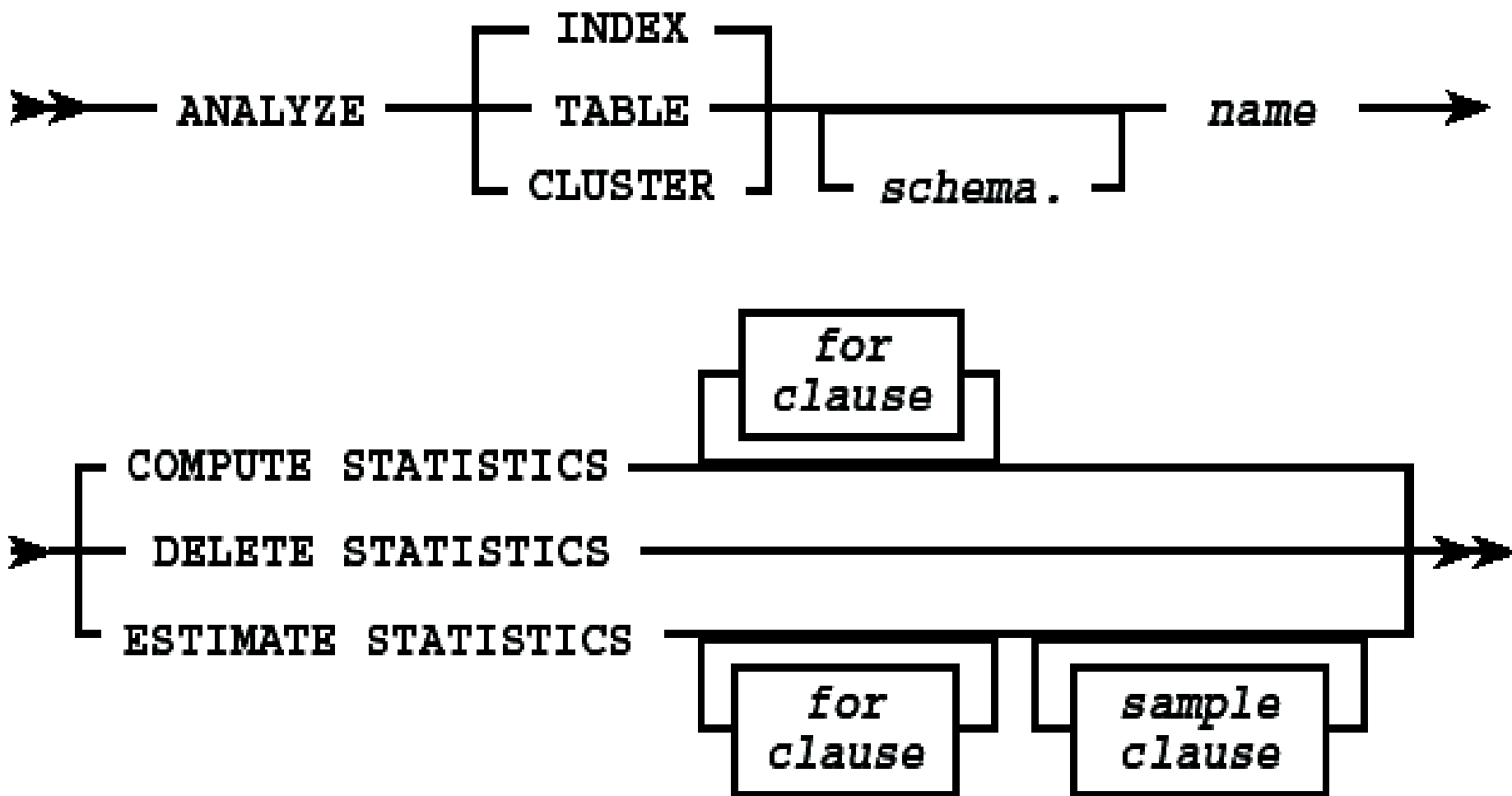


目的

- 使用**analyze**命令收集统计信息
- 认识表、索引、列、**cluster**统计数据
- 了解查询条件可选择性的计算
- 使用**dbms_stats**包
- 为数据分布畸形列建立**histogram**



Analyze命令

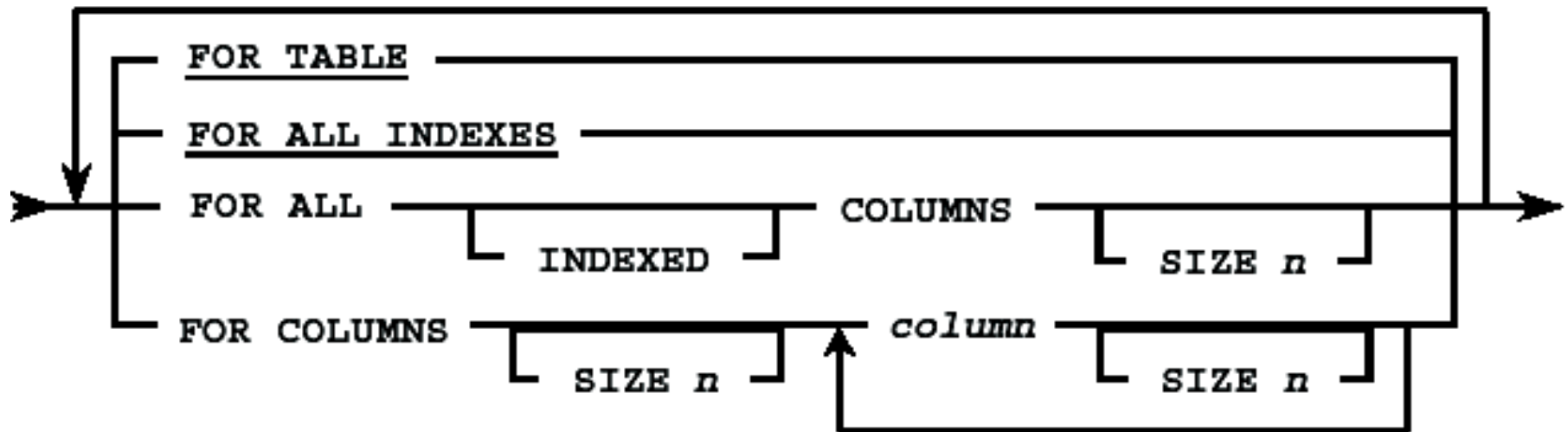




Analyze命令

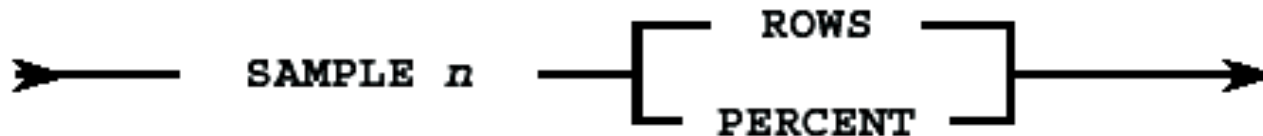
COMPUTE and ESTIMATE:

for
clause



ESTIMATE:

sample
clause





表的统计数据

- 逻辑行数
- 物理数据块数（精确）
- 物理空块数（精确）
- 平均剩余空间
- **Chained或migrated**行数
- 平均行长度
- 上次**analyze**时间和取样数
- 数据字典
 - **User_tables, all_tables**



索引统计数据

- 索引层数（精确）
- 叶块数（**leaf blocks**）
- 不同键值数
- 每个键值的平均叶块数
- 全部键值数
- **Clustering**因素
- 上次**analyze**时间和取样值



索引统计数据

- 新建或重建索引时收集统计数据
- 数据字典
 - User_indexes, all_indexes

```
SQL> select uniqueness, blevel, leaf_blocks,  
2> distinct_keys, clustering_factor  
3> from user_indexes  
4> where index_name='myindex';
```

UNIQUENES	BLEVEL	LEAF_BLOCKS	DISTINCT_KEYS	CLUSTERING_FACTOR
-----	-----	-----	-----	-----
UNIQUE	1	43	10000	709



列（column）统计数据

- 不同值的数量
- 最低值
- 最高值
- 空值数量
- 上次**analyze**时间和取样值
- 数据字典
 - **User_tab_col_statistics**
 - **All_tab_col_statistics**



Dbms_stats包

- 设置或得到个体的统计信息
- 在不同的数据字典或用户表之间传递统计信息
- 获取某种类别的统计信息



Dbms_stats: 产生统计信息

```
dbms_stats.GATHER_TABLE_STATS
('SH'                -- schema
,'CUSTOMERS'         -- table
, NULL               -- partition
, 20                  -- sample size(%)
, FALSE              -- block sample?
,'FOR ALL COLUMNS'   -- column spec
, 4                   -- degree of //
,'DEFAULT'            -- granularity
, TRUE               -- cascade to indexes
);
```



数据库之间拷贝统计信息

Data dictionary

**User-defined
statistics table**

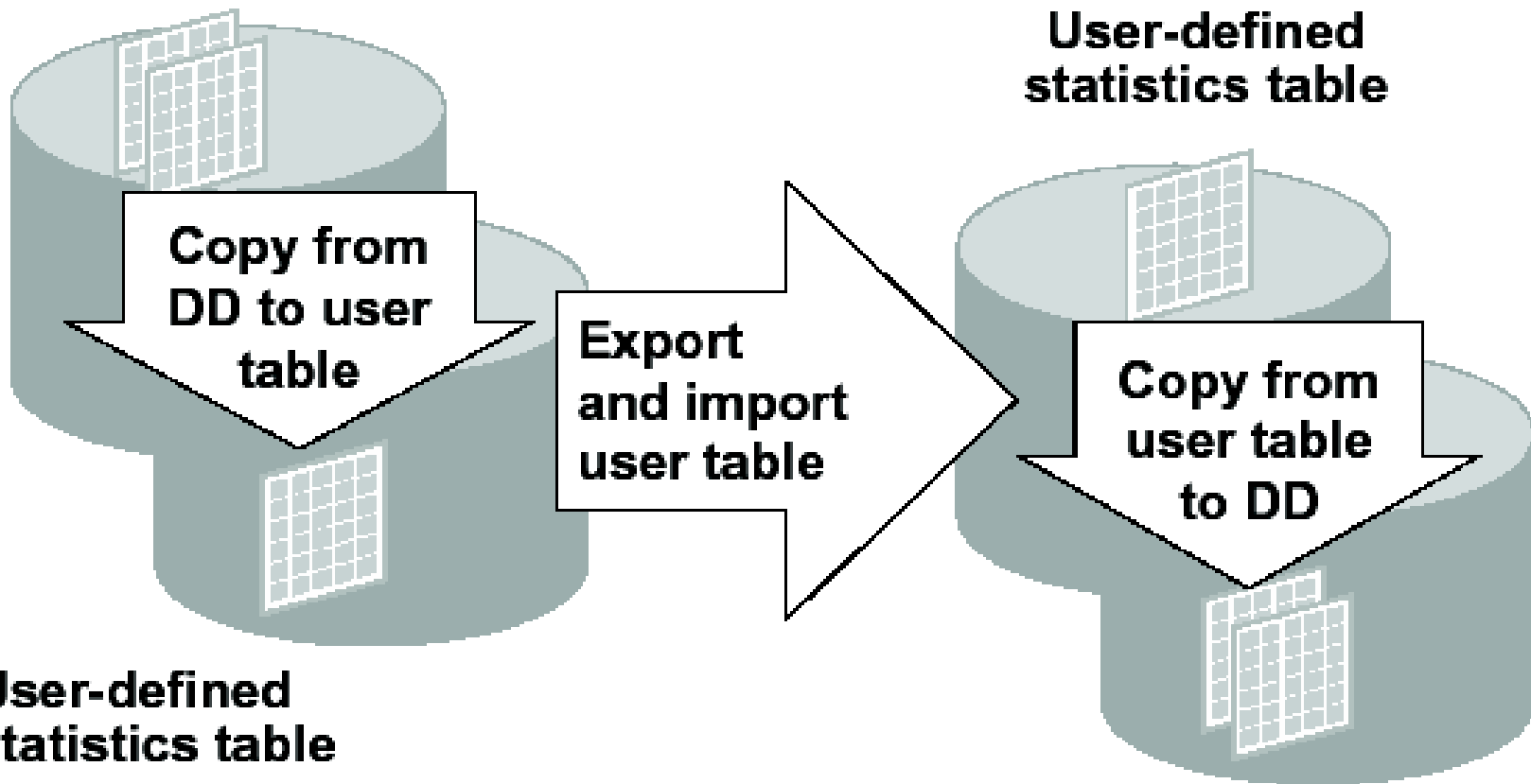
**Copy from
DD to user
table**

**Export
and import
user table**

**Copy from
user table
to DD**

**User-defined
statistics table**

Data dictionary ≡





数据库之间拷贝统计信息

- 使用**dbms_stats.create_stat_table**建立用户定义的存放统计信息的专用表
- 用**dbms_stats.export_schema_stats**将用户名下所有的统计信息放到上一步建立的表中
- **Exp、imp**该专用表
- 用**dbms_stats.import_schema_stats**将专用表中的统计信息放到数据字典中



拷贝统计信息一举例

```
dbms_stats.CREATE_STAT_TABLE  
( 'SH'          -- schema  
, 'STATS'       -- statistics table name  
, 'DATA01'     -- tablespace  
) ;
```

```
dbms_stats.EXPORT_TABLE_STATS  
( 'SH'          -- schema  
, 'CUSTOMERS'   -- table name  
, NULL         -- no partitions  
, 'STATS'       -- statistics table name  
, NULL         -- id for statistics  
, TRUE         -- index statistics  
) ;
```



收集统计信息一举例

```
begin
  dbms_stats.CREATE_STAT_TABLE
    ('SH', 'STATS');
  dbms_stats.GATHER_TABLE_STATS
    ('SH', 'CUSTOMERS'
     ,stattab => 'STATS');
end;
```

```
begin
  dbms_stats.DELETE_TABLE_STATS
    ('SH', 'CUSTOMERS');
  dbms_stats.IMPORT_TABLE_STATS
    ('SH', 'CUSTOMERS'
     ,stattab => 'STATS');
end;
```



条件的selectivity

- 唯一索引列 = 常量
 - 单行条件
- 非唯一索引列 = 常量

$$\text{Selectivity} = 1/\text{distinct_keys}$$

- 有界限或无界限的区域扫描

$$\text{Selectivity} = (\text{high}-\text{low}+1)/(\text{max}-\text{min}+1)$$

- **High:** 上限
- **Low:** 下限
- **Max,min:** 列统计信息

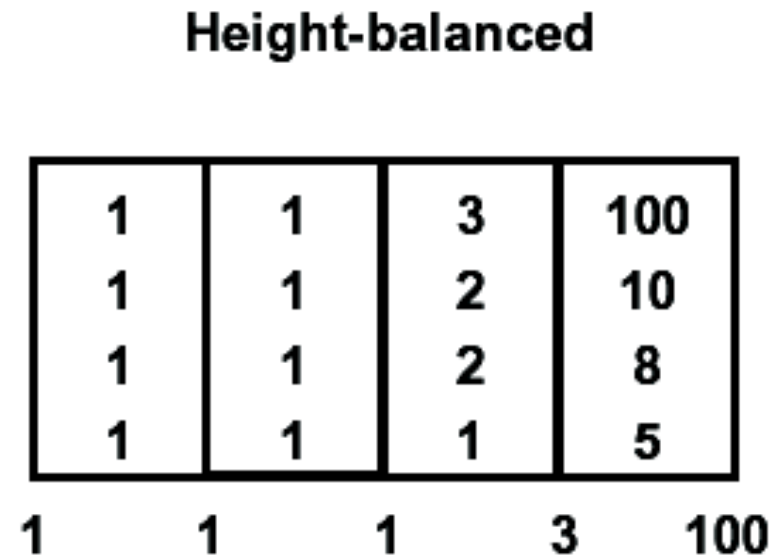
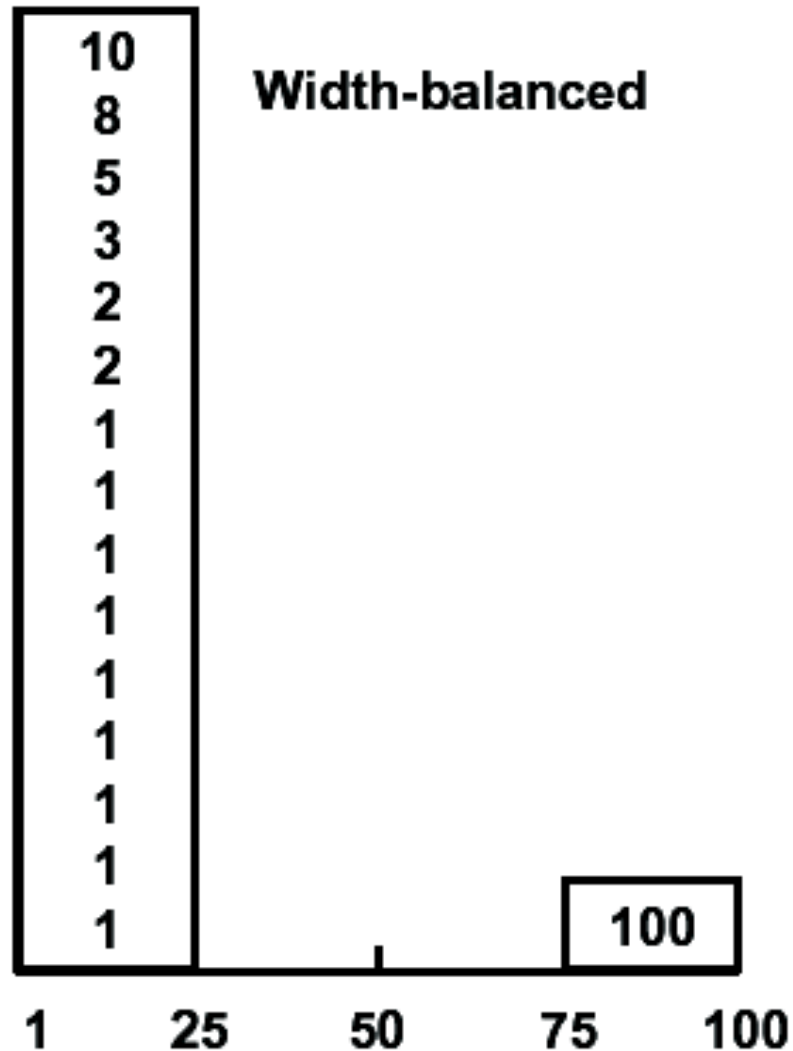


Bind变量与selectivity

- 等于条件：无区别
- 区域扫描条件：用内置假设的**selectivity**
- 如果性能要求很高
 - 考虑使用动态SQL，而非bind变量
 - **DBMS_SQL**
 - **EXECUTE IMMEDIATE**
 - 动态SQL意味着SQL共享程度的降低



Histograms



Height-balanced histograms give a better sense of the data distribution.



Histogram与selectivity

- 多数与少数值
- 区域扫描的**selectivity**取决于跨越的**bucket**数
- 等值查询的**selectivity**取决于
 - **Bucket**数量（多数值）
 - 分布密度（**density**）（少数值）



Histogram举例

- 生成emp表的sal列的统计数据，最多50个buckets

```
SQL> analyze table emp compute statistics for  
2> table for columns sal size 50 ;
```

- 重新计算统计数据，不指定bucket数

```
SQL> analyze table emp compute statistics  
2> for columns sal ;
```




Histogram举例

- 使用**dbms_stats**包

```
SQL> exec dbms_stats.getter_table_stats
```

```
2> ('scott', 'emp', method_opt=>
```

```
3> 'for columns size 50 sal');
```

- ‘**size 50**’指明了**histogram**最大**bucket**数
- **Size**设为**auto**时由系统自动决定哪个列需要**histogram**



Histogram小技巧

- 使用“**for all indexed columns**”选项
- 如果数据分布经常变化，需经常更新 **histogram**
- **Where**子句中使用**bind**变量时不使用 **histogram**
- 除非使用**histogram**对性能有很大提高，谨慎使用
- **Histogram**使用额外存储空间



何时使用**histogram**

- 当列的数据分布非常不平衡时
- 下列情况不使用**histogram**
 - 列没有用在**where**条件中
 - 列值是唯一的并且只用在等值条件中
 - 列参与的所有条件都使用**bind**变量的
 - 列值分布很均匀的



选择取样值

- 如果数据分布均匀，5%足够
- 如果不同值多于全部记录的10%时，使用更高的取样比例
- 当使用**histogram**时，取样的行数至少应是**histogram**的**bucket**数目的100倍



选择bucket数目

- 开始时使用缺省的**75个bucket**
- 尝试其他数目以获得最佳效果
- 如果某列有不多但经常出现的不同值时，**bucket数目应大于不同值的数目**



查看histogram信息

- **Histogram信息**

- User/all_histograms

```
SQL> select endpoint_number, endpoint_value  
2> from dba_tab_histograms  
3> where table_name='EMP'  
4> and column_name='SAL' ;
```

- **Histogram中bucket数目**

- User/all_tab_col_statistics



小结

- 使用**analyze**命令收集表、索引、列、**cluster**的统计数据
- 使用**dbms_stats**包
- 拷贝统计信息到不同的数据库中
- 确认查询条件中是否使用**bind**变量的区别
- 利用**histogram**
 - 多数值 (**popular value**)
 - 少数值 (**nonpopular value**)



影响与控制优化器



目的

- 在一些级别上影响优化器的行为
 - **Instance**和**session**级别
 - **SQL**语句级别
- 使用影响访问路径的**hint**
- 在视图上或视图使用**hint**



设定优化器模式

- 在**instance**级别，设定下列参数

Optimizer_mode =

{choose | rule | first_rows | all_rows}

- 在**session**级别，使用下列命令

SQL> alter session set optimizer_mode =

{choose | rule | first_rows | all_rows}

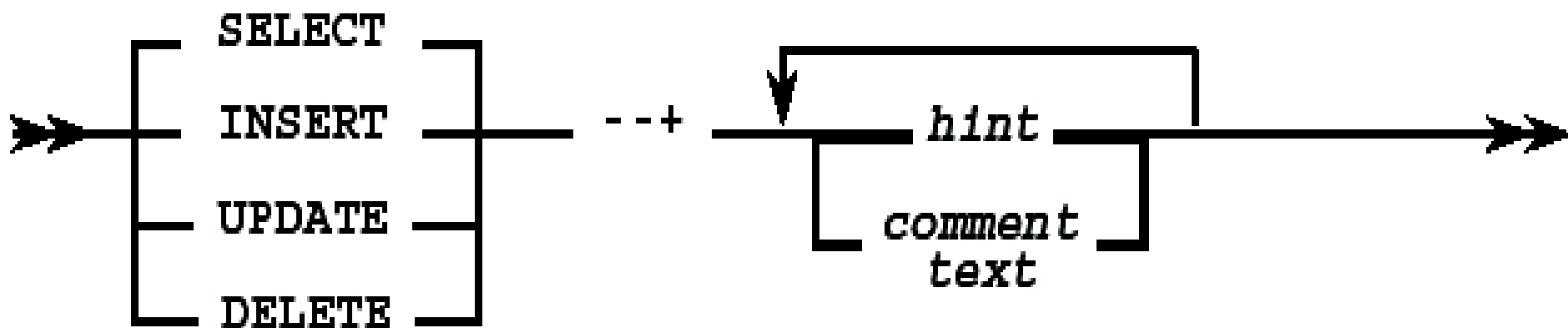
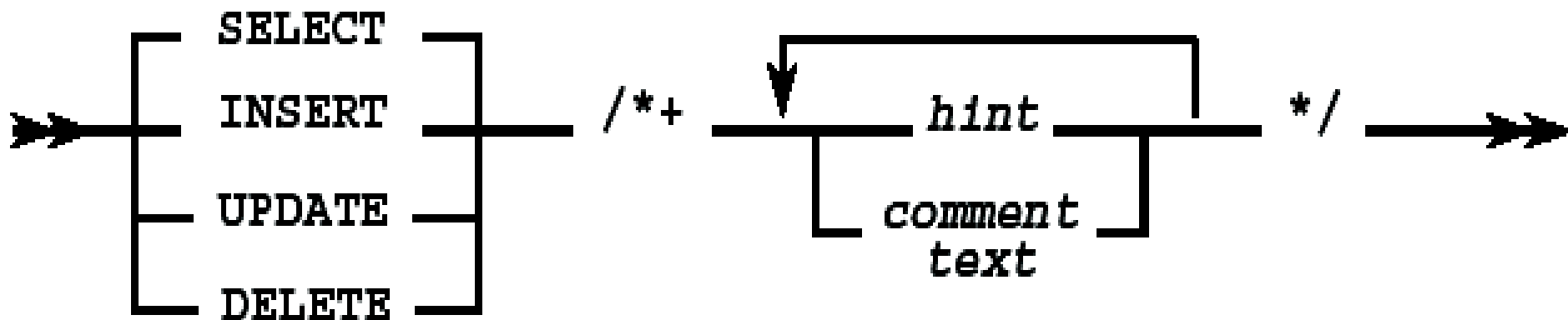


其他参数

- **Optimizer_mode_enable**
 - 设为当前版本可以使用最新的优化器特性
 - 缺省为当前版本
- **Optimizer_index_cost_adj**
 - 缺省为100，逻辑范围1—10000
 - 数值越小，优化器越趋于使用索引



优化器hint语法





使用**hints**规则

- 放置在**SQL**的第一个关键字后
- 每个**SQL**只能有一个**hint**段，但该**hint**段中可以使用多个**hint**
- **Hint**只对使用它的**SQL**有效
- 如果**SQL**语句使用**alias**，则**hint**中必须使用**alias**，而非具体表名



使用**hint**的推荐

- 小心使用**hint**，那意味着更高的维护成本
- 注意**hint**在数据库结构变化后对性能的负面影响

Hint举例



```
SQL> update --+ INDEX(p PROD_CATEGORY_IDX)
  2      products p
  3  set   p.prod_min_price =
  4      (select
  5          (pr.prod_list_price*.95)
  6          from products pr
  7          where p.prod_id = pr.prod_id)
  8  where p.prod_category = 'Men'
  9  and   p.prod_status = 'available, on stock'
 10  /
```



Hint类别

- 选择优化器模式
 - **Rule, choose**
 - **All_rows, first_rows**
- 选择访问方法
- 并行执行
- 连接方法与操作



基本访问路径hint

FULL	Performs a full table scan
ROWID	Accesses a table by ROWID
INDEX	Scans an index in ascending order
INDEX_ASC	Scans an index in ascending order
INDEX_DESC	Scans an index in descending order
AND_EQUAL	Merges single-column indexes
INDEX_FFS	Performs a fast full index scan
NO_INDEX	Disallows using a set of indexes



高级访问路径hint

CLUSTER	Performs a cluster scan
HASH	Performs a hash scan
HASH_AJ	Transforms a NOT IN subquery into a hash anti-join
MERGE_AJ	Transforms a NOT IN subquery into a merge anti-join
MERGE_SJ	Transforms a correlated EXISTS subquery into a merge semijoin
USE_CONCAT	Rewrites OR into UNION ALL and turns off INLIST processing
NO_EXPAND	Prevents OR-expansions



Buffer cache hint

CACHE	Places blocks at the MRU end of the LRU list (full table scan)
NOCACHE	Places blocks at the LRU end of the LRU list (full table scan) (default)



Hint与视图（view）

- 不要在视图中使用**hint**
- 使用视图优化技术
 - 语句转换
 - 象表一样访问视图
- **Hint**可以在可融合视图（**mergeable view**）和不可融合视图（**nonmergeable view**）



视图处理hint

MERGE	Merges complex views or subqueries with the surrounding query
NO_MERGE	Does not merge mergeable views



小结

- 设置优化器模式
- 使用**hint**语法
- 确定访问路径**hint**
- 分析**hint**对视图的影响



排序与连接



目的

- 优化排序操作性能
- 描述不同的连接技术
- 解释连接优化
- 寻求最优的连接执行计划



调整排序性能

- 大量的排序操作很昂贵
 - 调整排序参数
- **Distinct, group by**和大多数**set**操作导致排序操作
- 尽量减少排序操作
 - 不使用**distinct**和**group by**操作
 - 使用**union all**而不是**union**
 - 利用索引扫描来避免排序



Top-N SQL语句

```
SQL> SELECT *  
  2  FROM (SELECT prod_id  
  3           ,      prod_name  
  4           ,      prod_list_price  
  5           ,      prod_min_price  
  6           FROM products  
  7           ORDER BY prod_list_price DESC)  
  8  WHERE ROWNUM <= 5;
```



连接术语

- 连接语句
- 连接条件，非连接条件
- 单行条件
 - 有唯一性索引的列等值判断

```
SQL> SELECT  c.cust_last_name, c.cust_first_name,
2           co.country_id, co.country_name
3 FROM      customers c , countries co
4 WHERE     c.country_id = co.country_id
5 AND       co.country_id = 'JP'
6 OR        c.cust_id = 205;
```



连接术语

- 等值连接 (equijoin)

```
SQL> SELECT c.cust_last_name, co.country_name
2   FROM   customers c, countries co
3   WHERE  c.country_id = co.country_id;
```

- 不等连接 (nonequijoin)

```
SQL> SELECT  c.cust_last_name, c.cust_credit_limit
2   ,        cr.credit_limit_id
3   FROM    customers c, credit_limits cr
4   WHERE   c.cust_credit_limit
5   BETWEEN cr.credit_limit_min
6   AND     cr.credit_limit_max;
```



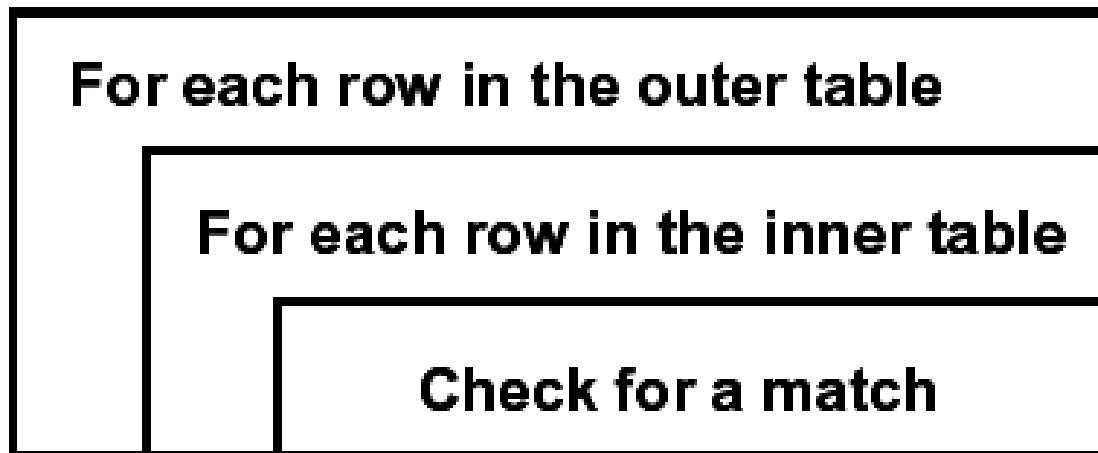
连接操作

- 连接操作从两个数据源得到并返回单一数据集
- 连接操作类型包括：
 - **Nested loop**连接
 - **Sort/merge**连接
 - **Hash**连接
 - **Cluster**连接
 - **Full outer**连接



Nested loop连接

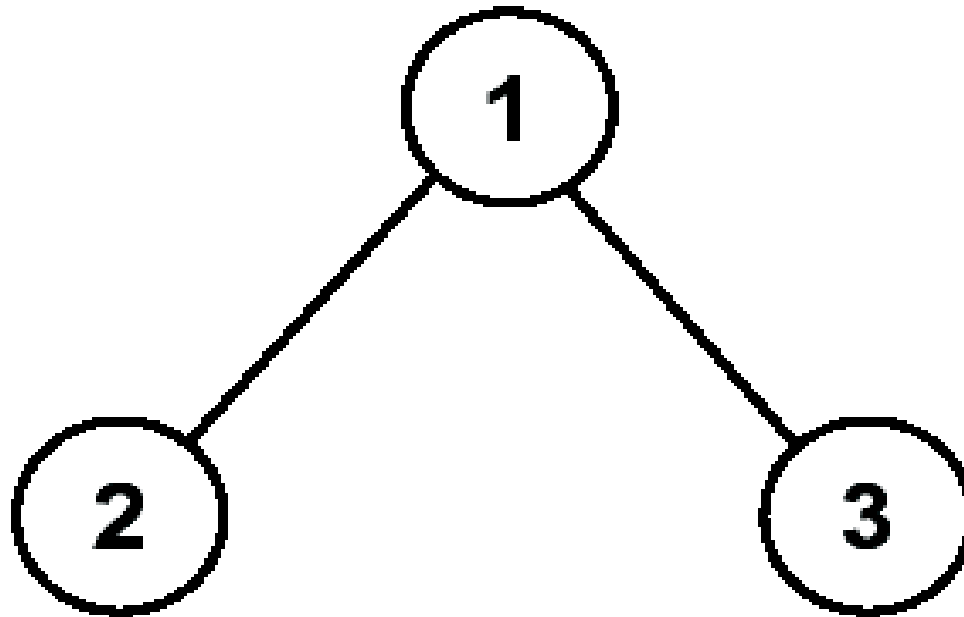
- 两个表中的一个被定义为**outer**表（或称**driving**表）
- 另一个表称为**inner**表
- 对于**outer**表的每一个记录，需取得所有对应的**inner**表的记录





Nested loop连接执行计划

Nested loops



**Table access
(Outer/driving table)**

**Table access
(Inner table)**

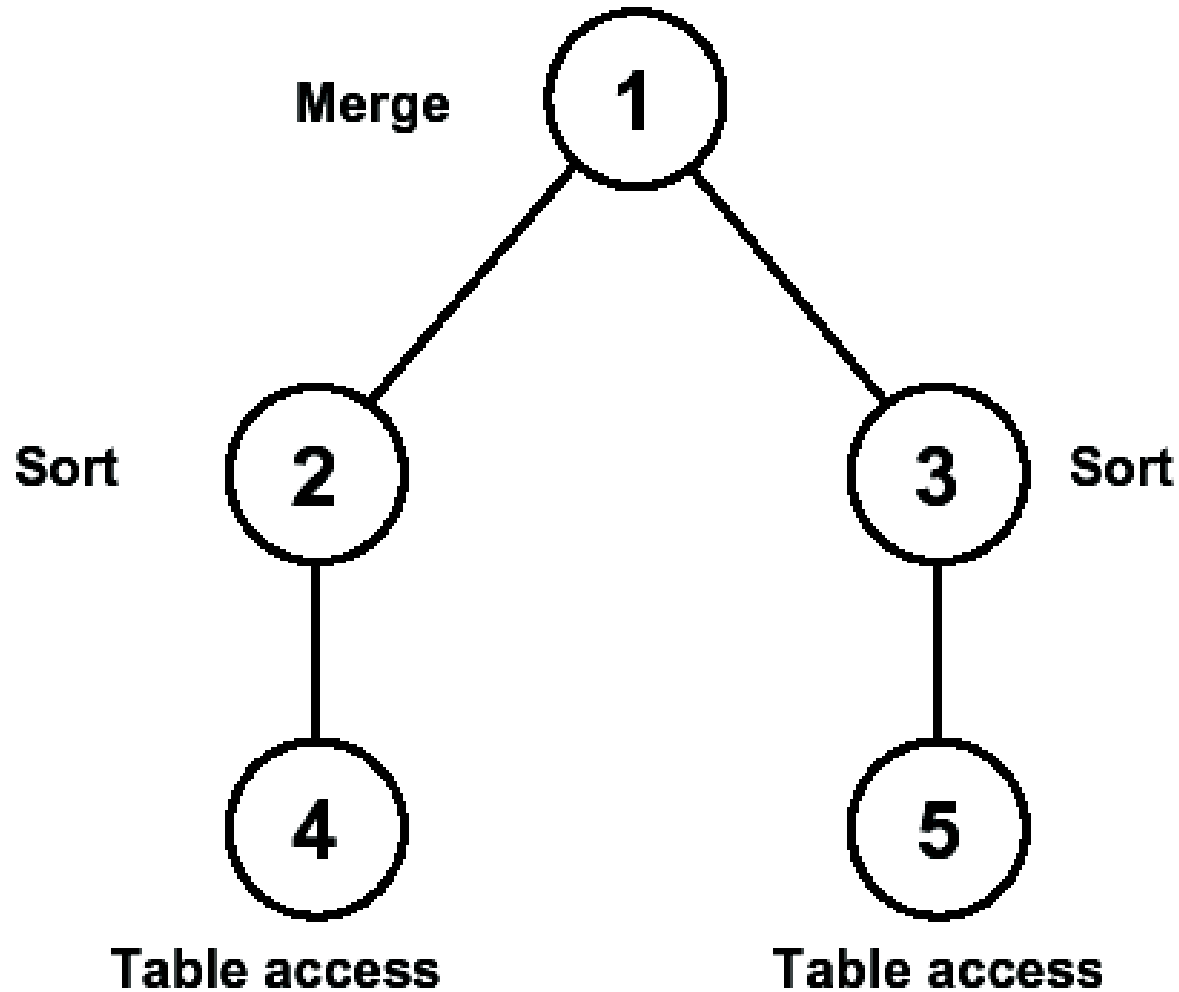


Sort/merge连接

- 每个数据源的行按照连接条件列排序
- 两个数据源然后融合到一起形成结果数据集合



Sort/merge连接执行计划





Hash连接

- 两个表被分成多个分区，使用全表扫描
- 对两表每一对分区，根据较小的分区在内存中建立一个**hash**表
- 用另一个分区来探查**hash**表



Hash连接执行计划

Hash join

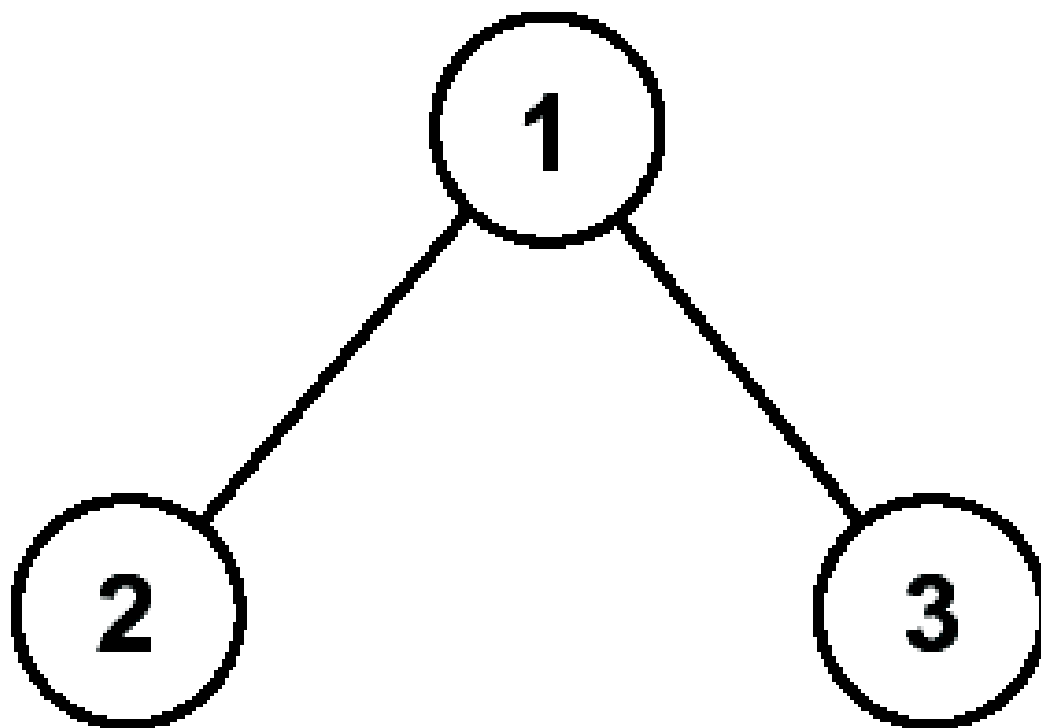


Table access

Table access

ORACLE



连接多个表

- 首先两个表连接，结果形成一个数据源
- 该数据源与下一个表做连接
- 重复上述步骤直至全部表完成连接



Outer连接

- 连接条件使用一个 (+) 号
- 非连接条件使用一个 (+) 号
- 条件中不使用 (+) 号则禁用outer连接

```
SQL> SELECT s.time_id, t.time_id  
2 FROM sales s, times t  
3 WHERE s.time_id (+) = t.time_id;
```



Outer连接的执行

- 可以使用索引
- 在连接条件无法使用索引时，做不等连接的外连接表可以使用索引



连接与优化器

- 优化器决定
 - 连接表的顺序
 - 连接操作类型
 - 每个数据源的访问路径



连接顺序规则

- 规则1
 - 单行条件使它的数据源放在前面
- 规则2
 - 对于**outer**连接，（+）表必须放在另一个表的后面



RBO连接优化

- 第一步，生产所有可能的连接顺序
- 这决定了下列组合

Number of Tables	Join Orders
-----	-----
2	$2! = 2$
3	$3! = 6$
4	$4! = 24$

- 增加连接表导致parse时间以几何级数增长



RBO连接优化

- 下列规则按顺序实行
 - 当**inner**表做全表扫描时，避免**nested loop**连接
 - 减少**sort/merge**连接
 - 选择访问路径级别最高的表作为**driving**表
 - 选择出现在**from**子句靠后的表做**driving**表



CBO连接优化

- 生成一系列可能的执行计划，由下列参数约束：
 - **Optimizer_search_limit**
 - **Optimizer_max_permutations**
- 然后优化器估算每个计划的成本，选择花费最少的执行计划



估算连接成本

- **Nested loop连接**
 - 成本为 $\text{read}(A) + \text{card}(A) * \text{read}(B)$
 - 影响因素—optimizer_index_caching参数
- **Sort/merge连接**
 - 成本为 $\text{read}(A) + \text{sort}(A) + \text{read}(B) + \text{sort}(B)$



连接顺序使用的hint

```
SQL> select /*+ ORDERED */...  
2   from    T1, T2, ...  
3   where   ...
```

```
SQL> select /*+ LEADING (T1) */...  
2   from    T1, T2, ...  
3   where   ...
```

```
SQL> select /*+ STAR */...  
2   from    T1, T2, ...  
3   where   ...
```



连接顺序使用的hint

```
SQL> select /*+ USE_NL(T2) */...  
2   from    T1, T2, ...  
3   where   ...
```

```
SQL> select /*+ USE_MERGE(T2) */...  
2   from    T1, T2, ...  
3   where   ...
```

```
SQL> select /*+ USE_HASH(T2) */...  
2   from    T1, T2, ...  
3   where   ...
```



其他连接hint

```
SQL> select /*+ STAR_TRANSFORMATION */ ...  
2   from    T1, T2, ...  
3   where   ...
```

```
SQL> select /*+ DRIVING_SITE(T1) */ ...  
2   from    T1, T2, ...  
3   where   ...
```



子查询与连接

- 与连接类似，子查询涉及到多个表
- 子查询类型
 - 非关联子查询
 - 关联子查询
 - **Not in**子查询 (**anti-joins**)
 - **Exists**子查询 (**semijoins**)



影响连接的参数

- **Hash连接参数**
 - **Hash_join_enabled**
 - **Hash_area_size**
 - **Hash_multiblock_io_count**
- **排序参数**
 - **Sort_area_size**
 - **Sort_area_retained_size**
 - **Sort_multiblock_read_count**



弃置行

- 取得的但是无用的行
- 计算弃置的行数，比较
 - 连接操作行数
 - 输入行总数



减少行弃置

- 使用索引
- 考虑使用**hint**强制使用**nested loop**连接，而不使用**sort/merge**连接



减少处理

- 将**nested loop**连接转换成**sort/merge**连接可导致：
 - 不使用索引
 - 增加排序的额外负担
- 理论上讲，**hash**连接最有效
- **Cluster**连接比相应的**nested loop**连接需要更少的I/O



小结

- 使用**top-N SQL**特性
- 描述连接操作
- 根据不同需要优化连接性能
- 影响连接顺序
- 发现调整连接比调整单表操作更复杂



执行计划的稳定性



目的

- 认识执行计划稳定性的目的与好处
- 建立存储的**outlines**
- 使用存储**outlines**
- 编辑存储**outlines**
- 维护存储**outlines**



优化计划稳定性

- 使经过调整的应用强行使用既定的**SQL**执行计划
- 在数据库变化中维持一致的执行计划
- 由使用了**hint**的存储**outline**实现
- 存储**outline**组成不同类型



计划的等效

- **SQL**语句文本必须一致
- 计划由下列维护
 - 新的**Oracle**版本
 - 数据库对象上的新统计信息
 - 初始参数的变化
 - 数据库重组
 - **Schema**变化
- 计划等效可以控制第三方应用的执行计划



建立存储outline

- 为session中的全部语句

```
SQL> ALTER SESSION
      2  SET CREATE_STORED_OUTLINES = TRAIN;
SQL> SELECT ... ;
SQL> SELECT ... ;
```

- 为具体一个语句

```
SQL> CREATE OR REPLACE OUTLINE CU_CO_JOIN
      2  FOR CATEGORY TRAIN ON
      3      SELECT co.country_name,
      4      cu.cust_city, cu.cust_last_name
      5      FROM    countries co , customers cu
      6      WHERE   co.country_id = cu.country_id
      ...
```



使用存储outline

- 设置 **user_stored_outlines = true** 或类型名

```
SQL> ALTER SESSION  
2   SET USE_STORED_OUTLINES = TRAIN;  
SQL> SELECT ...
```

- 可以在两个级别上设定参数 **create_stored_outlines** 和 **use_stored_outlines**
 - Alter system
 - Alter session



数据字典信息

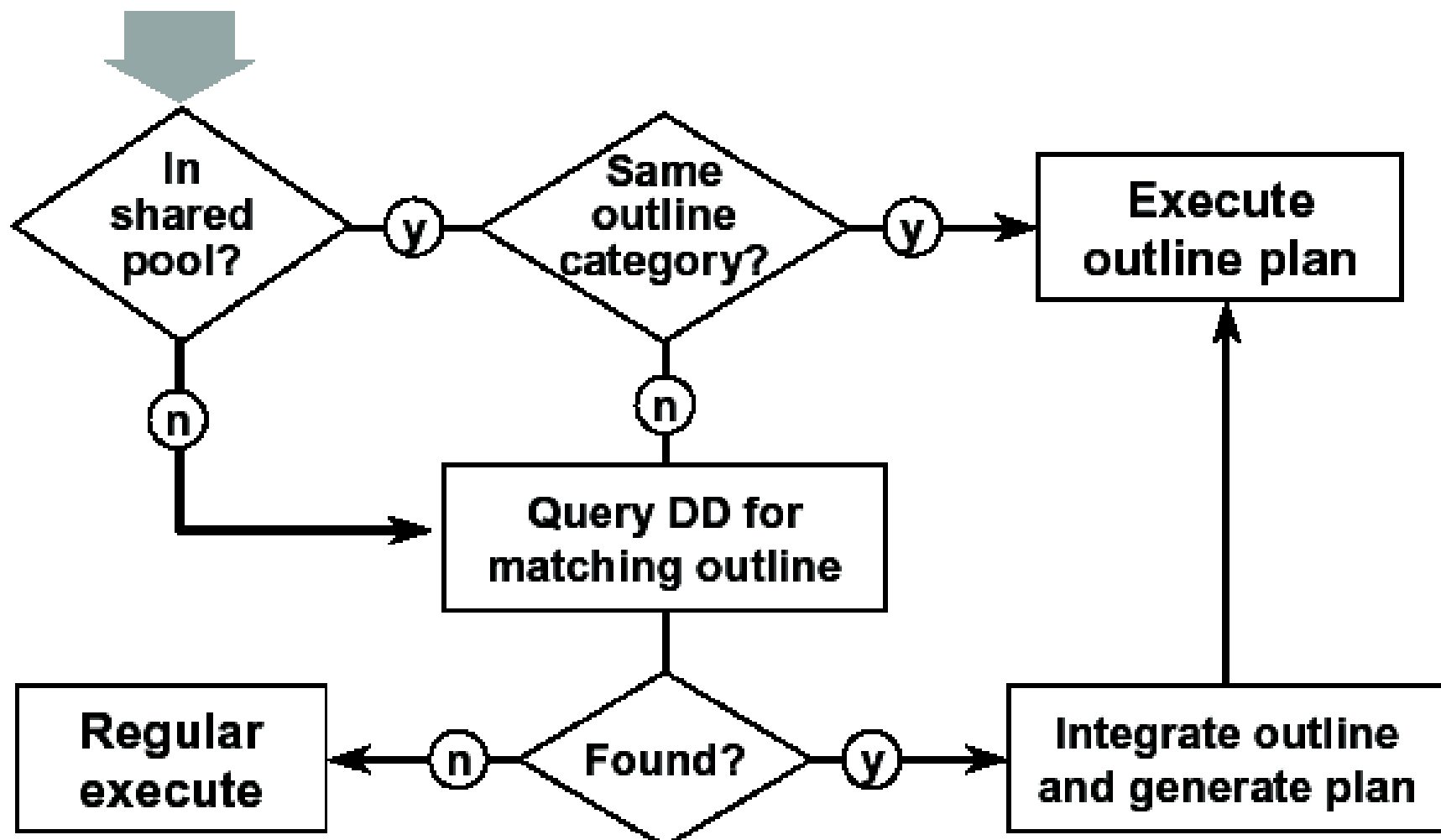
```
SQL> SELECT NAME, CATEGORY, USED  
2      ,      SQL_TEXT  
3      FROM    USER_OUTLINES;
```

```
SQL> SELECT NODE, HINT  
2      FROM    USER_OUTLINE_HINTS  
3      WHERE   NAME = ...;
```

```
SQL> SELECT SQL_TEXT, OUTLINE_CATEGORY  
2      FROM    V$SQL  
3      WHERE   ...;
```



执行计划逻辑





维护存储outline

- 使用**outln_pkg**
 - 删除不使用的**outline**
 - 删除**outline**类型
 - 重新命名**outline**类型
- 使用**alter outline**
 - 重命名**outline**
 - 重建**outline**
 - 为**outline**转换类型
- **Outline**存储在用户**outln**名下



维护存储outline

```
SQL> begin
  2     outln_pkg.DROP_UNUSED;
  3     outln_pkg.UPDATE_BY_CAT
  4         ('DEFAULT', 'TRAIN');
  5     outln_pkg.DROP_BY_CAT('TRAIN');
  6 end;
```



Outline配置参数

- **Use_private_outlines**是session参数

```
ALTER SESSION SET USE_PRIVATE_OUTLINES =  
TRUE | FALSE | category_name ;
```

- **True**允许在缺省的类型下使用私有outline
- **False**禁止使用私有outline
- 类型名运行使用该类型名下的私有outline



建立outline语法

```
CREATE [OR REPLACE]
  [PUBLIC | PRIVATE] OUTLINE [outline_name]
  [FROM [PUBLIC | PRIVATE] source_outline_name]
  [FOR CATEGORY category_name] [ON statement]
```



小结

- 使用**outline**确保执行计划的一致性
- 为**session**或语句建立**outline**
- 在类型中组织**outline**
- 使用或禁用**outline**
- 用**outln_pkg**和**alter outline**命令维护**outline**



高级索引



目的

- 建立**bitmap**索引
- 了解**bitmap**索引操作
- 设定**bitmap**索引**hint**
- 使用星型转换
- 建立**function-based**索引
- 查看数据字典信息



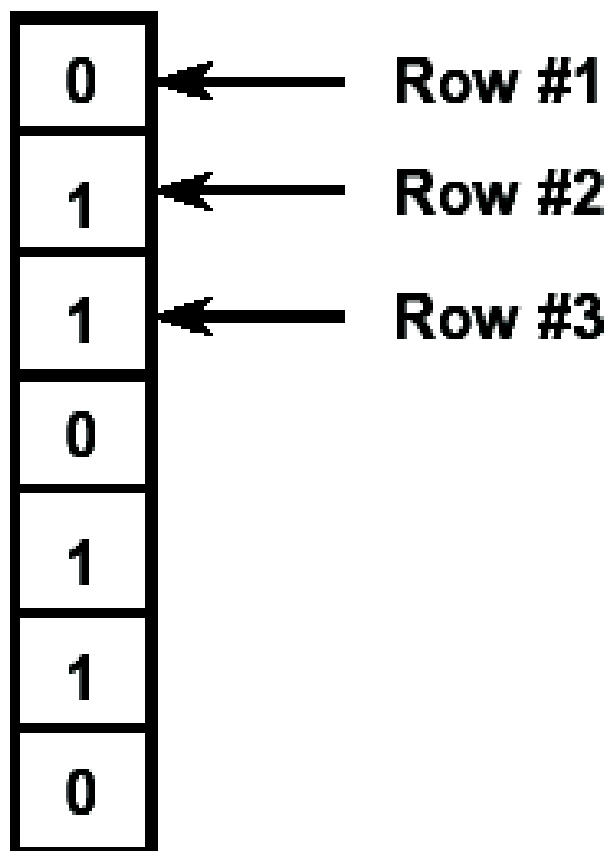
Bitmap索引

- 对于不同值少的列，使用**bitmap**索引性能更好并节省存储空间
- 每个**bitmap**索引存储的位图组成
- 每个位图包含索引的特定信息
- 位图被压缩并以**B*-tree**结构存储



Bitmap索引结构

- 位图中的每个位置存储一个特定行的信息





建立bitmap索引

```
SQL> CREATE BITMAP INDEX prod_supplier_id  
2 ON products (supplier_id);
```

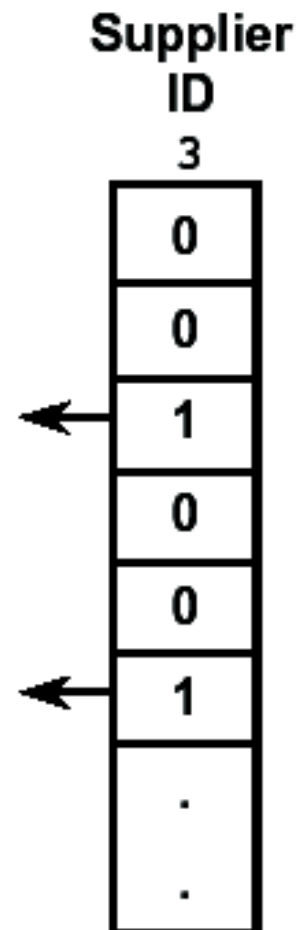
Row value	Supplier ID 1	Supplier ID 2	Supplier ID 3	Supplier ID 4	...
'1'	1	0	0	0	
'2'	0	1	0	0	
'3'	0	0	1	0	
'4'	0	0	0	1	
'2'	0	1	0	0	
'3'	0	0	1	0	
.	
.	



使用bitmap索引查询

```
SQL> SELECT *  
2 FROM products  
3 WHERE supplier_id = 3;
```

Rows
returned

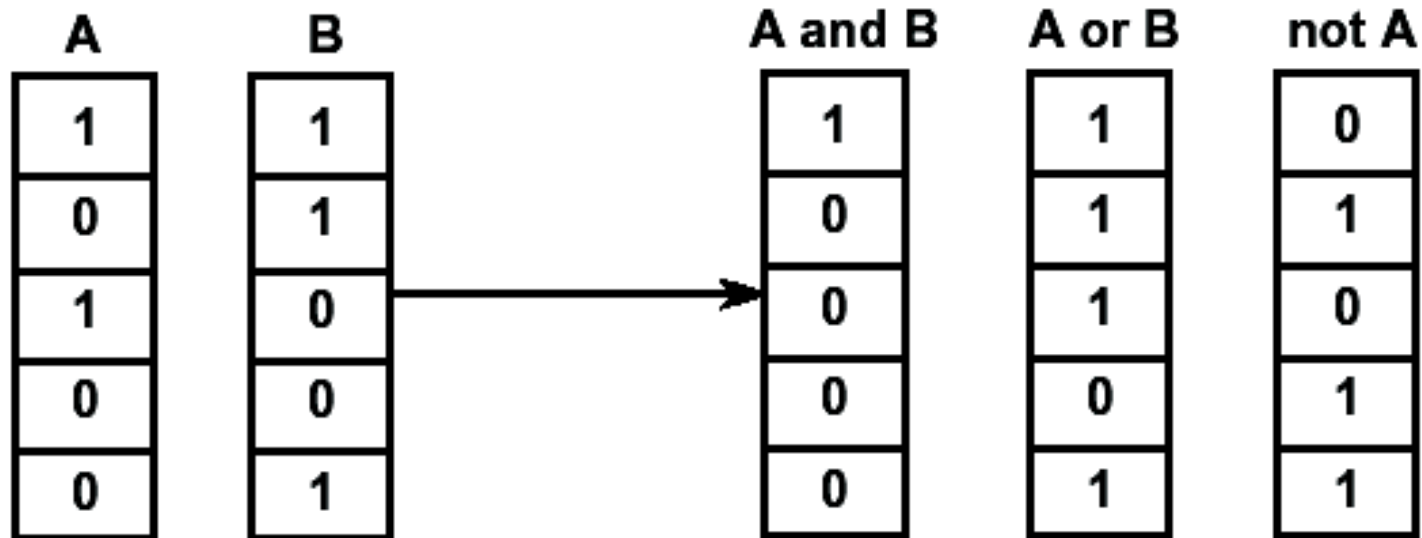




Bitmap索引的结合

由于快速的位图**and**, **minus**, **or**操作,
bitmap索引是高效的

- 当使用 **in** (... ..)





何时使用bitmap索引

- 当列值多数相同时
- 列被频繁地使用在
 - 复杂的where条件中
 - Group函数（count, sum等）
- 非常大的表
- DSS系统



Bitmap索引的优点

- 如果使用得当
 - 减少查询反应时间
 - 明显的存储空间节省
 - 显著的性能改善



Bitmap索引使用指导

- 减少存储空间
 - 可能的话尽量使用**not null**约束
 - 使用定长数据类型
 - 使用**alter table ... minimize records_per_block**命令
- 增加下列参数以增加索引性能
 - **Create_bitmap_area_size**（缺省8MB）
 - **Bitmap_merge_area_size**（缺省1MB）



Function-based索引|

```
SQL> CREATE INDEX FBI_UPPER_LASTNAME  
2 ON CUSTOMERS (upper(cust last name));
```

```
SQL> ALTER SESSION  
2 SET QUERY_REWRITE_ENABLED = TRUE;
```

```
SQL> SELECT *  
2 FROM customers  
3 WHERE upper(cust last name) = 'SMITH';
```



Function-based索引的使用

- 方便频繁计算的表达式
- 方便大小写敏感的查询
- 提高一种简单的数据压缩
- 可以用作**nls**排序索引



数据自动信息

```
SQL> SELECT i.index_name, i.index_type
2      ,      ic.column_name, i.status
3 FROM      user_indexes i
4      ,      user_ind_columns ic
5 WHERE     i.index_name = ic.index_name
6 AND       i.table_name='SALES';
```



小结

- **Bitmap索引**
- **Function-based索引**



Materialized视图与临时表



目的

- 了解**materialized**视图的目的和优势
- 建立**materialized**视图
- 查询重写
- 建立**dimension**
- 了解临时表的优势



Materialized视图

- 是一次SQL执行的结果
- 是一种segment
 - 使用常规空间管理
 - 使用自己的索引
- 主要用于
 - 频繁而复杂的连接
 - 统计求和数据



建立materialized视图

```
SQL> CREATE MATERIALIZED VIEW
  2  fweek_pscat_costs_mv
  3  AS
  4  SELECT      t.week_ending_day
  5             , t.calendar_year
  6             , p.prod_subcategory
  7             , sum(c.unit_cost) AS dollars
  8  FROM        costs c
  9             , times t
 10             , products p
 11  WHERE        c.time_id = t.time_id
 12  AND          c.prod_id = p.prod_id
 13  GROUP BY    t.week_ending_day
 14             , t.calendar_year
 15             , p.prod_subcategory;
```

Materialized view created.



刷新materialized视图

- 刷新类型
 - 完全
 - 快速
 - 强制
 - 从不
- 为快速刷新建立materialized视图log

```
SQL> CREATE MATERIALIZED VIEW LOG ON ...
```



刷新materialized视图

- 手动刷新
 - 使用dbms_mview包
- 自动刷新
 - 同步：在底层表改变提交时
 - 异步：定义刷新时间间隔



查询重写

- 如果访问**materialized**视图而非表，查询必须重写
- 查询重写对应用透明
- 不需要特殊权利
- 可以**enable**或**disable**



查询重写

- 优化器为使用**materialized**视图而重写查询
 - **Query rewrite**权限运行用户**enable**视图
 - **Dbms_olap**包可以使用视图



查询重写

- 使用**explain plan**或**autotrace**验证查询重写的发生
- 检查查询反应
 - 访问更少的数据块
 - 反应时间应会有很大提高



建立materialized视图句法

```
CREATE MATERIALIZED VIEW mview_name
  [TABLESPACE ts_name]
  [PARALLEL (DEGREE n)]
  [BUILD {IMMEDIATE|DEFERRED}]
  [REFRESH {FAST|COMPLETE|FORCE
            |NEVER|ON COMMIT}]
  [{ENABLE|DISABLE} QUERY REWRITE]

AS SELECT ... FROM ...
```



控制查询重写

- 查询重写只在**CBO**中使用

```
QUERY_REWRITE_ENABLED = {true|false}  
QUERY_REWRITE_INTEGRITY =  
{enforced|trusted|stale_tolerated}
```

- 优化hint为
 - Rewrite
 - Norewrite



查询重写例子

```
SQL> explain plan for
  2  SELECT      t.week_ending_day
  3      ,        t.calendar_year
  4      ,        p.prod_subcategory
  5      ,        sum(c.unit_cost) AS dollars
  6  FROM        costs c
  7      ,        times t
  8      ,        products p
  9  WHERE       c.time_id = t.time_id
...

```


OPERATION	NAME

SELECT STATEMENT	
TABLE ACCESS FULL	fweek_pscat_costs_mv



查询重写例子

```
SQL> SELECT      t.week_ending_day
  2      ,        t.calendar_year
  3      ,        p.prod_subcategory
  4      ,        sum(c.unit_cost) AS dollars
  5  FROM          costs c, times t, products p
  6  WHERE         c.time_id = t.time_id
  7  AND          c.prod_id = p.prod_id
  8  AND          t.calendar_year = '1999'
  9  GROUP BY     t.week_ending_day, t.calendar_year
 10      ,        p.prod_subcategory
 11  HAVING        sum(c.unit_cost) > 10000;
```



```
SQL> SELECT      week_ending_day
  2      ,        prod_subcategory
  3      ,        dollars
  4  from          fweek_pscat_costs_mv
  5  where         calendar_year = '1999'
  6  and          dollars > 10000;
```



临时表

- 包含的数据存在周期为**transaction**或**session**
- 定义始终存在于数据字典中
- 数据只对**session**可见
- 使用内存中的排序空间
- 如果需要，可以分配临时**extent**



建立临时表

```
SQL> CREATE OR REPLACE VIEW sales_detail
 2  AS
 3  SELECT cu.cust_last_name, cu.cust_email
 4         ,      cu.cust_income_level
 5         ,      pr.prod_name, ch.channel_desc
 6         ,      pm.promo_name, sa.amount_sold
 7  FROM    customers cu, products pr
 8         ,      channels ch, promotions pm, sales sa
 9  WHERE   sa.cust_id = cu.cust_id
10  AND     sa.prod_id = pr.prod_id
11  AND     sa.channel_id = ch.channel_id
12  AND     sa.promo_id = pm.promo_id
13  AND     sa.time_id between
14         '01-DEC-1999' and '31-DEC-1999'
```

View created.



建立临时表

```
SQL> CREATE GLOBAL TEMPORARY TABLE sales_detail_temp
  2  ( cust_last_name      VARCHAR2(50)
  3    , cust_income_level VARCHAR2(30)
  4    , cust_email        VARCHAR2(30)
  5    , prod_name         VARCHAR2(50)
  6    , channel_desc      VARCHAR2(20)
  7    , promo_name        VARCHAR2(20)
  8    , amount_sold       NUMBER)
  9  ON COMMIT PRESERVE ROWS;
```

Table created.

```
SQL> INSERT INTO sales_detail_temp
  2  SELECT * FROM sales_detail;
```

37830 rows created.



建立临时表

```
SQL> SELECT table_name, temporary, duration
2  FROM    dba_tables
3  WHERE   table_name = 'SALES_DETAIL_TEMP';
```

TABLE_NAME	T	DURATION
-----	-	-----
SALES_DETAIL_TEMP	Y	SYS\$SESSION



小结

- 建立**materialized**视图
- 使用视图**enable**查询重写
- 使用临时表



其他的存储技术



目的

- **Index-organized表**
- **外部表 (external table) ***



存储用户数据



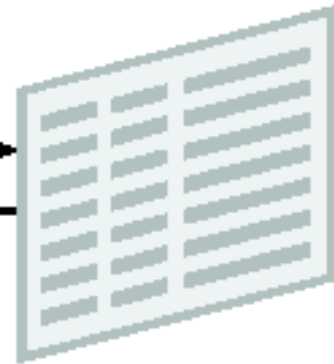
**Regular
table**



**Index-organized
table**



**External
table**



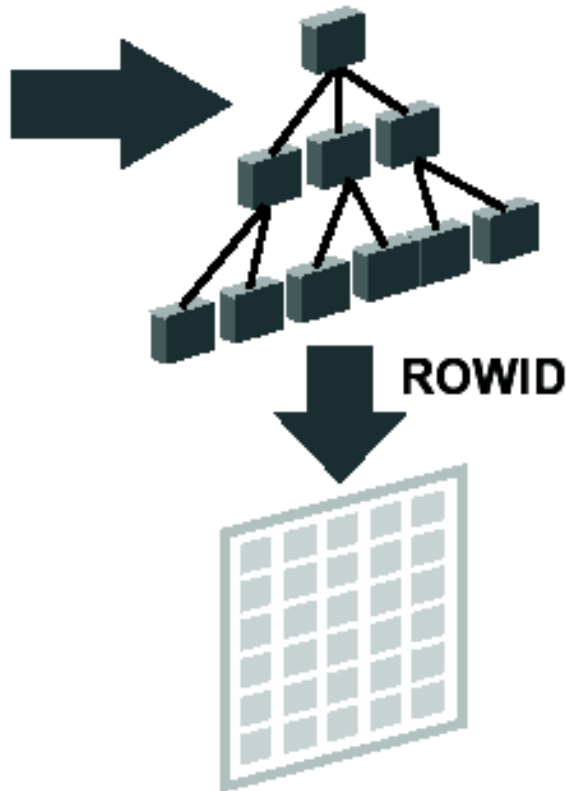
OS file



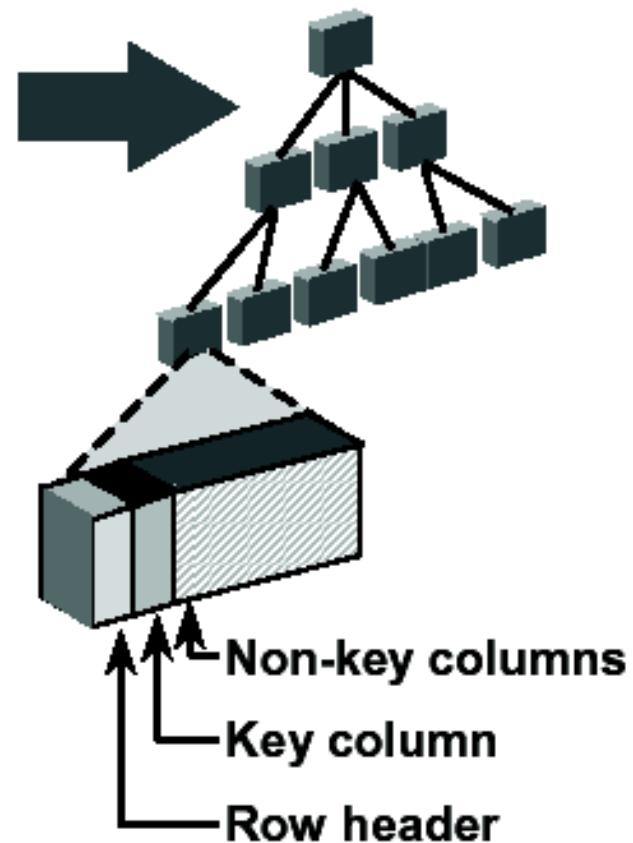


Index-organized表

Indexed access
on table



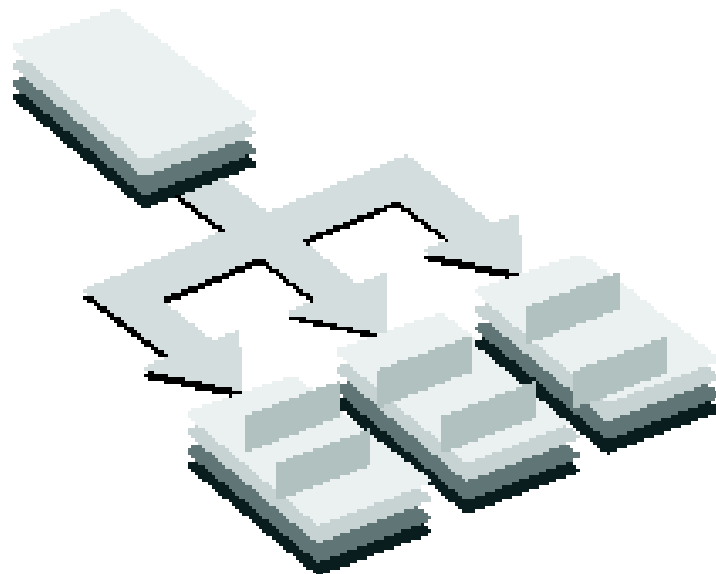
Accessing index-
organized table





IOT性能特性

- **B*-tree**存储全行
- 经过排序的行
- 逻辑rowid
- 快速的通过键值访问表数据





IOT的局限

- 必须有主键
- 不能是索引**cluster**或**hash cluster**的一部分
- 不能包含**long**型数据列
 - 但**lob**列可以



何时使用IOT

- 注意IOT对用户和应用透明
- 用于文本、声音、图象等的存取



建立IOT

```
CREATE TABLE table-name  
( column_definitions  
[,constraint_definitions] )  
ORGANIZATION INDEX  
[ block_util_parameters ]  
[ storage_clause ]  
[ TABLESPACE tablespace ]  
[ PCTTHRESHOLD integer  
  [ INCLUDING column_name ] ]  
[ OVERFLOW segment_attr_clause ]
```



获得IOT信息

USER_TABLES

TABLE_NAME
IOT_TYPE
IOT_NAME
TABLESPACE_NAME

USER_INDEXES

TABLE_NAME
INDEX_NAME
INDEX_TYPE
PCT_THRESHOLD
INCLUDE_COLUMN



小结

- **IOT**



谢谢