# 一 查询转换

子查询解嵌套 在where语句之后

视图合并在 where语句之前

1.子查询解嵌套[/\*+ no\_unnest\*/]

第一种情形：将子查询拆开，把该子查询中的表，视图从子查询中拿出来，然后和外部查询中的表，视图做表联接。不考虑SQL成本

第二种情形:对于那种不拆开子查询，但是会把该子查询转换成一个内嵌视图的子查询展开。考虑SQL成本

注意：

等价改写sql语义 == 原sql语义 直接展开属于第一种情形

等价改写sql语义 != 原sql语义 转换成视图在展开第二种情形

2.视图合并[/\*+ no\_merge()\*/]

简单视图合并：1.不使用外联接，2.视图SQL中不含[distinct,rownum, group by]等聚合函数，集合运算符 [union,minus,intersect] 做了简单视图合并后等价改写SQL的成本值一定小于或等于原SQL的成本值。不考虑SQL成本

外联接视图合并：1.使用了外联接，2.视图SQL中不含[distinct,rownum, group by]等聚合函数，集合运算符 [union,minus,intersect] 外连接视图合并的前提条件是：要么视图被作为外连接的驱动表，要么该视图虽作为外连接的被驱动表但它的视图定义SQL语句中只包含一个表。不考虑SQL成本

复杂视图合并：1.视图定义SQL语句中含有group by 或 distinct 的复杂视图

复杂视图对应的group by 或 distinct操作的延迟不一定能带来效率的提升，

1.group by 或 distinct操作能过滤大部分数据且表连接不能过滤，不合并 视图效率更好

2.group by 或 distinct不能能过滤大部分数据且表连接能过滤，合并视图 效率更好。

考虑SQL成本

注意：1.[外链接]是指外部查询表和视图的联接方式

2.解析函数，聚合函数，集合运算，order by ,rownum也可以防止视图合并与/\*+ no\_merge()\*/ 作用一样

3.星形转换

星型转换的核心是将原星形连接中针对各个维度表的限制条件通过等价改写的方式以额外的子查询施加到事实表上，然后在通过对事实表上的各个连接列上已存在的位图索引间的位图操作(如按位与，按位或等)，来达到有效减少事实表上待访问的数据量，避免对事实表做全表扫描的目的。适合事实表数据量很大，维度表数据小

4.连接谓词推入 [/\*+ no\_push\_pred()\*/]

优化器将视图的定义SQL语句当做一个独立的单元处理，但此时优化器会把原本处于该视图外部查询中和该视图之间的连接条件推入到视图的定义SQL语句内部，这样是为了能使用上该视图内部相关基表上的索引，进而能走出基于索引的嵌套循环。考虑SQL成本

Oracle支持对如下类型的视图做连接谓词推入

视图定义SQL 语句中包含 union/union all 的视图

视图定义SQL 语句中包含 DISTINCT 的视图

视图定义SQL 语句中包含 GROUP BY 的视图

和外部查询之间的连接类型是外链接视图

和外部查询之间的连接方法是反连接的视图(反连接使用了 not exists,not in ,<> 关键字)

和外部查询之间的连接方法是半连接的视图(反连接使用了 exists, in 关键字)

5.连接因式分解

优化器处理带UNION ALL 的目标SQL的一种优化，它是指优化器处理以UNION ALL 连接的目标SQL 的各个分支时，不在原封不动的分别重复执行每个分支，而是会把各个分支中的公共部分提出来作为一个单独的结果集，然后在和原UNION ALL 中剩下的部分做连接。使用UNION ALL的不能进行视图合并，但有可能会因式分解

6.SQL中的IN

在Oracle中，IN 和 OR 是等价的，优化器处理带 IN 的目标SQL时实际上会将其转换成带 OR 的等价改写SQL。

优化器在处理带 IN 的目标SQL时，通常会采用如下四种方法：

IN-List Iterator：1.针对目标SQL的 IN 后面是常量集合的处理方法，通常比 IN-List Expansion高，2.IN-List Iterator 处理IN的前提条件是IN所在列上一定要有索引

IN-List Expansion：针对目标SQL的 IN 后面是常量集合的另一种处理方法，它是指优化器把目标SQL中IN后面的常量集合拆开，把里面的每个常量提出来一个分支，各分支之间用UNION ALL 来连接。改写后各个分支就各自可以走各自的索引，表连接等相关执行计划。等价改写后的目标SQL的解析时间会随着UNION ALL 分支递增而递增，意味着IN后面集合包含元素过多时，解析时间就会很长。考虑SQL成本

IN-List Filter：目标SQL要满足如下两个条件：

1.目标SQL 的 IN 后面是子查询而不是常量集合

2.Oracle [未对] 目标SQL 的 IN 后面的子查询做子查询展开

对 IN 做子查询展开/视图合并：目标SQL要满足如下前提条件：

1.目标SQL 的 IN 后面是子查询而不是常量集合

2.Oracle [对]目标SQL 的 IN 后面的子查询做子查询展开

# 二 表的联接方式

排序-合并联接：在条件为非等式的时候 ，还有就是数据大到内存无法保存，需要外存排序

嵌套循环联接：

1./\*+ ordered use\_nl(A表(有别名必须使用别名) B表)\*/ (from 的表顺序必须和 use\_nl 中表顺序一样，提示才能生效)

2.[嵌套循环联接]适用驱动表返回结果集较小约1万行以内，并且联接的列上带有索引(指被驱动表关联列带有索引)，性能最好

3.filter操作是netloop的优化(关联列，有重复数据,子查询受制外表驱动不能视图合并)

散列联接：使用大表作为驱动表(大是指存储块大小)，大表只被全表扫描一次，小表被多次哈希探测

# 三 扫描方式(全表扫描,索引扫描)

\*表所选定的扫描方法被用来确定最终计划中所使用的[表联接方法和联接顺序]

全表扫描：此扫描是否高效取决于需要访问的数据块以及返回结果集行数

索引快速扫描：count(\*) 走快速扫描

索引唯一扫描：查询条件 列存在primary key ,unique 索引唯一索引，

索引范围扫描：查询条件 使用 < ,>,like,between甚至是 = 时使用范围扫描，

范围越大就有可能使用全表扫描替代它

索引跳跃扫描：复合索引

优化器：优化器的目的是按照一定的判断原则来得到它认为目标sql在当前情形下最高效的访问路径

基于成本的优化器CBO：成本指的是I/O和CPU消耗

基于规则的优化器RBO:

基于选择的优化器

一，存储中定义table和arr

TYPE MY\_TABLE IS TABLE OF VARCHAR(20) INDEX BY BINARY\_INTEGER;

--数组类型(定长数组)

TYPE MY\_ARR IS VARRAY(100) OF VARCHAR2(100) ;

二，存储过程返回结果集

CREATE OR REPLACE PACKAGE BODY PKG\_CODE\_COMPARE IS

--存储返回结果集游标

PROCEDURE TEST(O\_CUR OUT SYS\_REFCURSOR)IS

TYPE C\_CURREF IS REF CURSOR;

C\_CUR C\_CURREF;

BEGIN

OPEN O\_CUR FOR SELECT \* FROM SB\_YXZD;

END TEST;

--存储返回表结果集

PROCEDURE TEST2(O\_TAB OUT MY\_TABLE)IS

V\_ROW VARCHAR2(20);

BEGIN

O\_TAB(0):='test1';

O\_TAB(1):='test2';

END TEST2;

--返回流式结果集

TYPE IDS\_TAB IS TABLE OF VARCHAR2(20);

FUNCTION CONVERTOTABLE(ID\_GROUP IN VARCHAR2) RETURN IDS\_TAB PIPELINED IS

TYPE TYPE\_REFCUR IS REF CURSOR ;

C\_CURSOR TYPE\_REFCUR;

V\_ITEM VARCHAR2(30);

BEGIN

OPEN C\_CURSOR FOR SELECT REGEXP\_SUBSTR(ID\_GROUP, '[^,]+', 1, ROWNUM) AS ID FROM DUAL

CONNECT BY ROWNUM <=

LENGTH(REGEXP\_REPLACE(ID\_GROUP, '[^,]')) + 1;

LOOP

FETCH C\_CURSOR INTO V\_ITEM;

EXIT WHEN C\_CURSOR%NOTFOUND;

PIPE ROW(V\_ITEM);

END LOOP;

END CONVERTOTABLE;

END PKG\_CODE\_COMPARE;

三，列转行\_\_行转列

v\_nams:='0201,0203,0206'

--[列转行]

--LISTAGG-->列转行

SELECT LISTAGG('字段名称', ',') WITHIN GROUP(ORDER BY '字段名称') as zzmc

FROM 表名 t1

where 过滤条件;

--层级查询-->列转行

select max(sys\_connect\_by\_path('字段名称', ',')) as txt

from (select rownum no, '字段名称', rownum, lag(rownum) over(order by rownum) pid

from 表名 t)

start with pid is null

connect by prior no = pid;

--[行转列]

select regexp\_substr(v\_nams, '[^,]+', 1, rownum) from dual

connect by rownum<length(regexp\_replace(v\_nams,'[^,]'))+2

# JVM调优

Java堆：3~4倍Full GC后的老年代空间占用

-Xms 初始堆大小

-Xmx 最大堆大小

永久代：1.2~1.5倍Full GC后的永久代空间占用

-XX:PermSize

-XX:MaxPermSize

新生代：1.2~1.5倍Full GC后的老年代空间占用

-Xmn

老年代：2~3倍Full GC后的老年代空间占用

老年代 = Java堆-新生代

Survivor空间：survivor 空间大小 = -Xmn<value>/(-XX:SurvivorRatio=<ratio>+2)

-XX:SurvivorRatio=<ratio> 标识Survivor和Eden的比率

-XX:MaxTenuringThreshold=<n> 晋升阀值

-Xloggc:d:\\jvm.log 输出GC日志文件

-XX:+PrintGCDateStamps 或 -XX:+PrintGCTimeStamps 打印GC时间

-XX:+PrintGCDetails 打印GC详细信息

-XX:+PrintHeapAtGC 打印堆信息

-XX:+PrintAdaptiveSizePolicy 垃圾收集中Survivor空间的统计信息

-XX:+PrintCommandLineFlags 输出VM初始化堆大小

-XX:+PrintTenuringDistribution 监控晋升阀值（也就是对象经历MinitorGC次数后，提升至老年代）

-XX:+UseParallelGC Throughput收集器

-XX:+UseConcMarkSweepGC CMS收集器

注意：通用原则：从Throughput收集器切换到CMS收集器，将老年代空间增大20%-30%

垃圾收集调优基础

垃圾收集性能的三个主要属性：

吞吐量：不考虑垃圾收集引起的停顿和内存消耗，最高性能指标()

延迟：度量标准是缩短垃圾收集引起的停顿时间，避免程序抖动

内存占用：垃圾收集器流畅运行所需要的内存数量

注意：其中一个性能的提高都是以其他两个属性性能损失为代价的

垃圾收集器调优的三个基本原则：

MinorGC原则：每次MinorGC都尽可能多的收集垃圾对象，遵守这一原则可以减少应用程序触发FullGC，

FullGC的持续时间总是最长，是应用程序无法达到其延迟和吞吐量的罪魁祸首

GC内存最大化原则：处理吞吐量和延迟问题时，垃圾处理器能使用的内存越大，即JAVA堆空间越大，

垃圾收集效果越好，应用程序运行的也就越流畅

GC调优3选2原则：在这3个性能属性（吞吐量，延迟，内存）中任意选两个进行JVM垃圾收集调优

正常GC指标：

1.MinorGC执行时间不到50ms

2.MinorGC执行不频繁，约10s一次

3.FullGC执行时间不到1s；

4.FullGC执行不算频繁，不低于10分钟一次

垃圾调优目的：

减少对象从新生代提升老年代比率的方法

减少Eden的空间会导致频繁MinitorGC，频率越高对象老化速度越快

增大Survivor空间时保持Eden大小不变

jmap -histo:live <vmid> 触发Full Gc

Tomcat的catalina.bat文件中设置

set JAVA\_OPTS=-XX:+PrintGCDateStamps -XX:+PrintGCDetails

-XX:+PrintAdaptiveSizePolicy -Xloggc:d:\\jvm.log