深入理解 Oracle 的并行执行(上)

作者: 陈焕生

目录:

Oracle 的并行执行

Oracle 的并行执行是一种分而治之的方法. 执行一个 sql 时, 分配多个并行进程同时执行数据扫描, 连接以及聚合等操作, 使用更多的资源, 得到更快的 sql 响应时间. 并行执行是充分利用硬件资源, 处理大量数据时的核心技术.

在本文中, 在一个简单的星型模型上, 我会使用大量例子和 sql monitor 报告, 力求以最直观简单的方式, 向读者阐述并行执行的核心内容:

- Oracle 并行执行为什么使用生产者-消费者模型.
- 如何阅读并行执行计划.
- 不同的数据分发方式分别适合什么样的场景.
- 使用 partition wise join 和并行执行的组合提高性能.
- 数据倾斜会对不同的分发方式带来什么影响.
- 由于生产者-消费者模型的限制, 执行计划中可能出现阻塞点.
- 布隆过滤是如何提高并行执行性能的.
- 现实世界中, 使用并行执行时最常见的问题.

术语说明:

1. S: 时间单位秒.
2. K: 数量单位一千.
3. M: 数量单位一百万, 或者时间单位分钟.
4. DoP: Degree of Parallelism, 并行执行的并行度.
5. QC: 并行查询的 Query Coordinator.
6. PX 进程: Parallel Execution Slaves.
7. AAS: Average active session, 并行执行时平均的活动会话数.
8. 分发: pq distribution method, 并行执行的分发方式, 包括 replicate, broadcast, hash 和 adaptive 分发等 4 种方式, 其中 adaptive 分发是 12c 引入的的新特性, 我将在本篇文章中一一阐述.
9. Hash join 的左边: 驱动表, the build side of hash join, 一般为小表.
10. Hash join 的右边: 被驱动表, the probe side of hash join, 一般为大表.
11. 布隆过滤: bloom filter, 一种内存数据结构, 用于判断一个元素是否属于一个集合.

测试环境和数据

Oracle 版本为 12.1.0.2.2, 两个节点的 RAC, 硬件为 Exadata X3-8.
这是一个典型的星型模型, 事实表 lineorder 有 3 亿行记录, 维度表 part/customer 分别包含 1.2M 和 1.5M 行记录, 3 个表都没有进行分区, lineorder 大小接近 30GB.

表名	行数
lineorder	300005811
part	1200000
customer	1500000

```
select
   owner seg_owner,
   segment_name seg_segment_name,
   round(bytes/1048576,2) SEG_MB
from
```

```
   dba_segments
where
    owner = 'SID'
and segment_name in ('LINEORDER','PART','CUSTOMER')
/

OWNER  SEGMENT_NAME SEGMENT_TYPE   SEG_MB
------ ------------ ------------ --------
SID    LINEORDER    TABLE        30407.75
SID    CUSTOMER     TABLE             168
SID    PART         TABLE             120
```

本篇文章所有的测试, 除非特别的说明, 我关闭了 12c 的 adaptive plan 特性, 参数 optimizer_adaptive_features 被默认设置为 false. Adaptive 相关的特性如 cardinality feedback, adaptive distribution method, adaptive join 都不会启用. 如果检查执行计划的 outline 数据, 你会发现 7 个优化器相关的隐含参数被设置为关闭状态. 事实上, 12c 优化器因为引入 adaptive plan 特性, 比以往版本复杂得多, 剖析 12c 的优化器的各种新特性, 我觉得非常具有挑战性, 或许我会在另一篇文章里尝试一下☺

```
select * from table(dbms_xplan.display_cursor('77457qc9a324k',0,'outline'));

...

Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('12.1.0.2')
      DB_VERSION('12.1.0.2')
      OPT_PARAM('_optimizer_use_feedback' 'false')
      OPT_PARAM('_px_adaptive_dist_method' 'off')
      OPT_PARAM('_optimizer_dsdir_usage_control' 0)
      OPT_PARAM('_optimizer_adaptive_plans' 'false')
      OPT_PARAM('_optimizer_strans_adaptive_pruning' 'false')
      OPT_PARAM('_optimizer_gather_feedback' 'false')
      OPT_PARAM('_optimizer_nlj_hj_adaptive_join' 'false')
      OPT_PARAM('optimizer_dynamic_sampling' 11)
      ALL_ROWS
      ……
      END_OUTLINE_DATA
  */
```

并行初体验

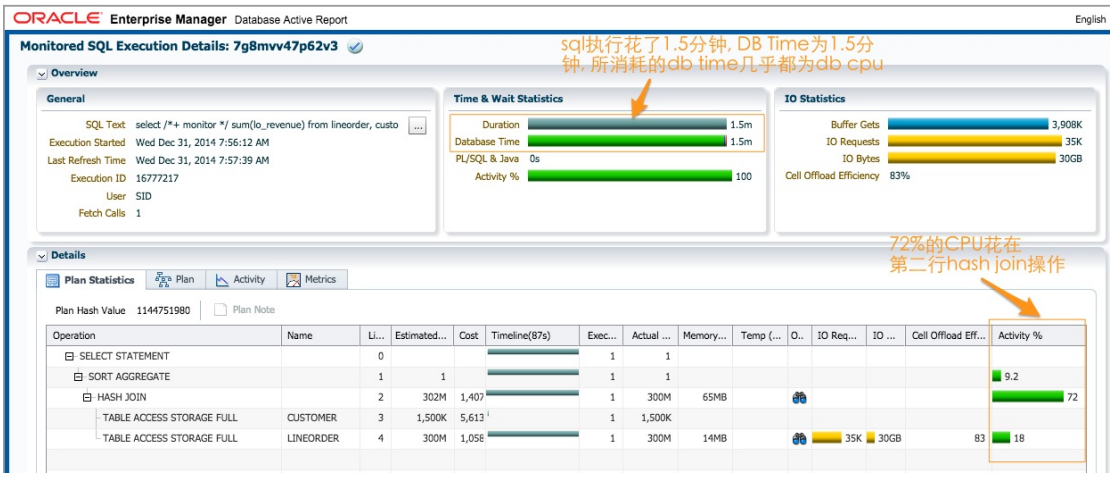
串行执行

以下 sql 对 customers 和 lineorder 连接之后, 计算所有订单的全部利润. 串行执行时不使用 parallel hint:

```
select /*+ monitor */
    sum(lo_revenue)
from
    lineorder, customer
where
    lo_custkey = c_custkey;
```

串行执行时, sql 执行时间为 1.5 分钟, db time 为 1.5 分钟. 执行计划有 5 行, 一个用户进程工作完成了对 customer, lineorder 两个表的扫描, hash join, 聚合以及返回数据的所有操作. 此时 AAS(average active sessions)为 1, sql 执行时间等于 db time. 几乎所有的 db time 都为 db cpu, 72% 的 cpu 花在了第二行的 hash join 操作. 因为测试机器为一台 Exadata X3-8, 30GB 的 IO 请求在一秒

之内处理完成. Cell offload Efficiency 等于 87%意味着经过存储节点扫描, 过滤不需要的列, 最终返回计算节点的数据大小只有 30GB 的 13%.



并行执行

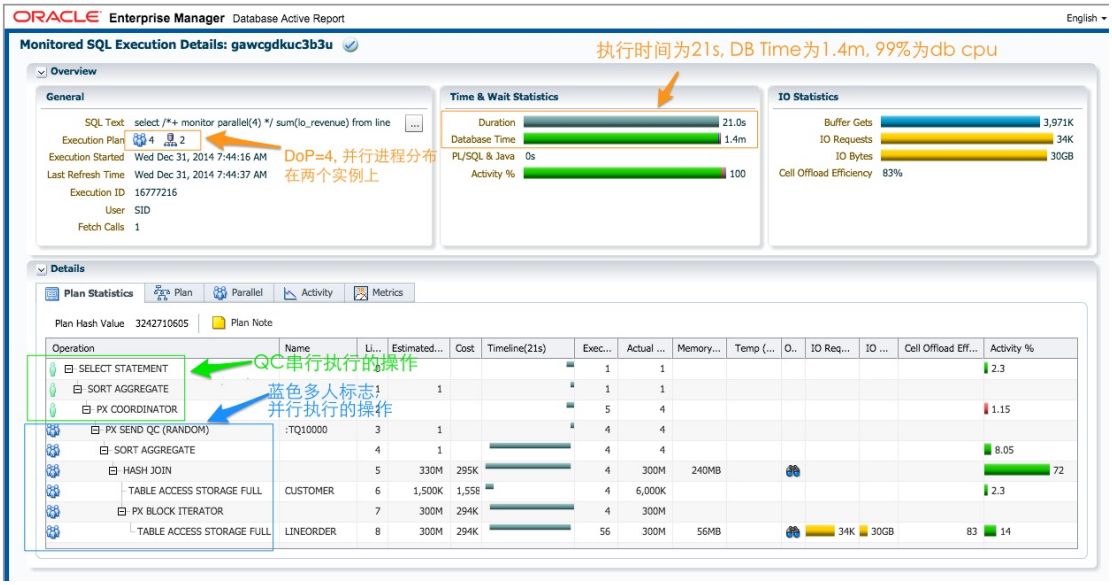
使用 hint parallel(4), 指定 DoP=4 并行执行同样的 sql:

```
select /*+ monitor parallel(4)*/
    sum(lo_revenue)
from
    lineorder, customer
where
    lo_custkey = c_custkey;
```

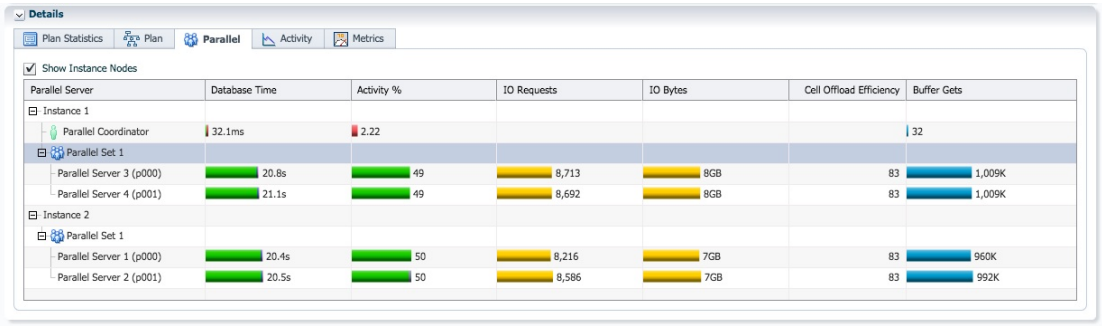
SQL 执行时间为 21s, db time 为 1.4 分钟. DoP=4, 在两个实例上执行. 执行计划从 5 行增加为 9 行, 从下往上分别多了'PX BLOCK ITERATOR', 'SORT AGGREGATE', 'PX SEND QC(RANDOM)'和'PX COORDINATOR'这四个操作.

其中 3 到 8 行的操作为并行处理, sql 的执行顺序为: 每个 PX 进程扫描维度表 customer(第 6 行), 以数据块地址区间作为单位(第 7 行)扫描四分之一的事实表 lineorder(第 8 行), 接着进行 hash join(第 5 行), 然后对连接之后的数据做预先聚合(第 4 行), 最后把结果给 QC(第三行). QC 接收数据(第 2 行)之后, 做进一步的汇总(第 1 行), 最后返回数据(第 0 行).

SQL 执行时间比原来快了 4 倍, 因为最消耗时间的操作, 比如对 lineorder 的全表扫描, hash join 和聚合, 我们使用 4 个进程并行处理, 因此最终 sql 执行时间为串行执行的 1/4. 另一方面, db time 并没有明显下降, 并行时 1.4m, 串行时为 1.5m, 从系统的角度看, 两次执行消耗的系统资源是一样的.



DoP=4 时, 因为没有涉及数据的分发(distribution), QC 只需分配一组 PX 进程, 四个 PX 进程分别为实例 1 和 2 的 p000/p0001. 我们可以从系统上查看这 4 个 PX 进程. 每个 PX 进程消耗大致一样的 db time, CPU 和 IO 资源. AAS=4, 这是最理想的情况, 每个 PX 进程完成同样的工作量, 一直保持活跃. 没有串行点, 没有并行执行倾斜.



AAS=4, 查看活动信息时, 为了更好的展示活动信息, 注意点掉”CPU Cores”这个复选框.

在 Linux 系统上显示这四个 PX 进程.

```
[oracle@exa01db01 sidney]$ ps -ef | egrep "p00[01]_SSB"
oracle    20888      1  4  2014 ?        18:50:59 ora_p000_SSB1
oracle    20892      1  4  2014 ?        19:01:29 ora_p001_SSB1
[oracle@exa01db01 sidney]$ ssh exa01db02 'ps -ef | egrep "p00[01]_SSB"'
oracle    56910      1  4  2014 ?        19:01:03 ora_p000_SSB2
oracle    56912      1  4  2014 ?        18:53:30 ora_p001_SSB2
```

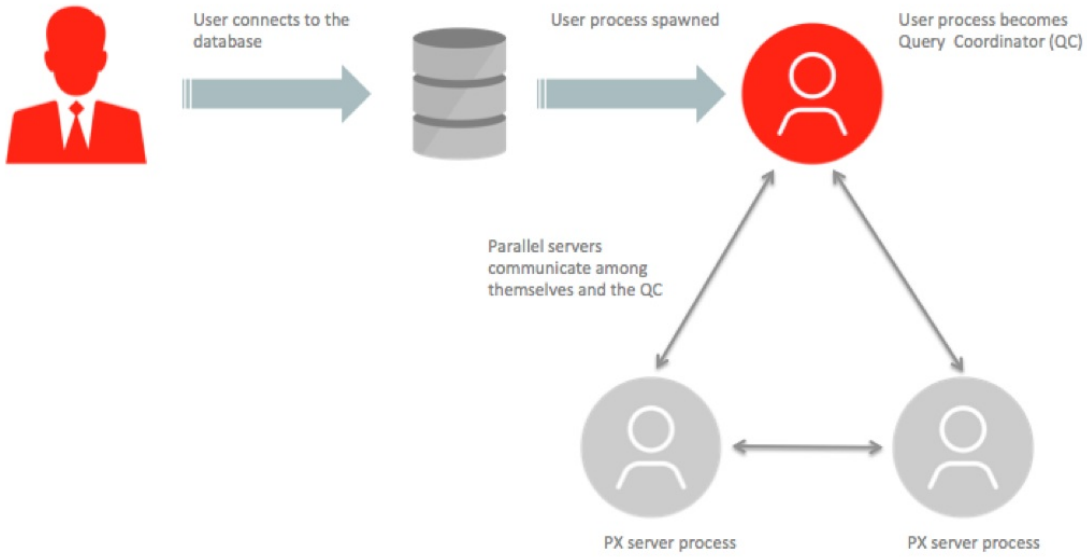
小结

本节的例子中, DoP=4, 并行执行时分配了 4 个 PX 进程, 带来 4 倍 的性能提升. SQL monitor 报告包含了并行执行的总体信息和各种细节, 比如 QC, DoP, 并行执行所在的实例, 每个 PX 进程消耗的资源, 以及执行 SQL 时 AAS. 下一节, 我们将深入讨论并行执行的生产者-消费者模型.

生产者-消费者模型

在上面并行执行的例子中, 每个 px 进程都会扫描一遍维度表 customer, 然后扫描事实表 lineorder 进行 hash join. 这时没有数据需要进行分发, 只需要分配一组 px 进程. 这种 replicate 维度表的行为, 是 12c 的新特性, 由参数_px_replication_enabled 控制.

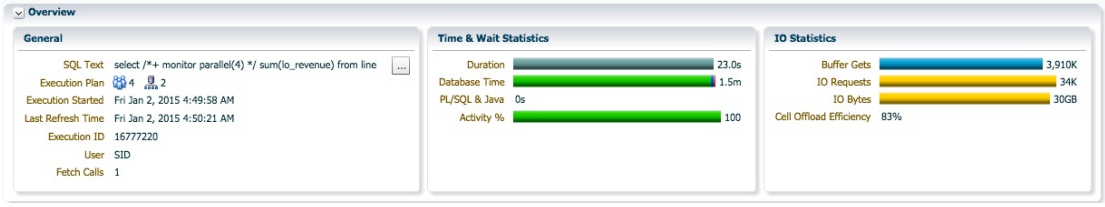
更常见情况是并行执行时, QC 需要分配两组 PX 进程, 互为生产者和消费者, 协同工作, 完成并行执行计划. 架构图[1]如下:



Broadcast 分发, 一次数据分发

为了举例说明两组 px 进程如何协作的, 设置_px_replication_enabled 为 false. QC 会分配两组 PX 进程, 一组为生产者, 一组为消费者.

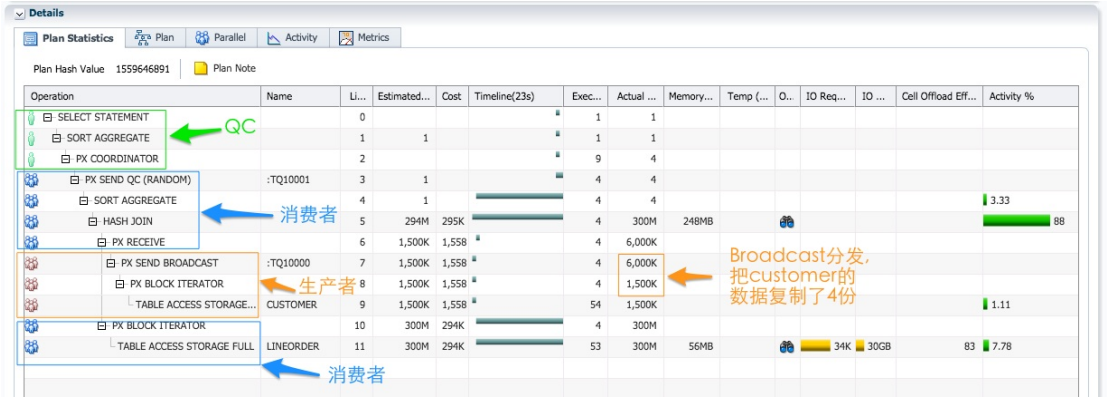
见下图, 此时 sql 执行时间为 23s, 执行时间变慢了 2s, db time 仍为 1.5 分钟.



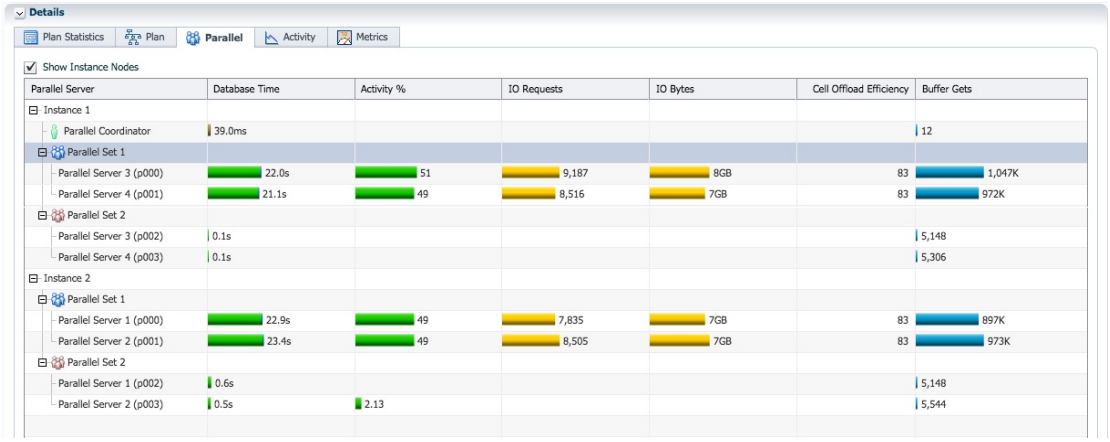
最大的变化来自执行计划, 现在执行计划有 12 行. 增加了对 customer 的并行扫描'PX BLOCK ITERATOR'(第 8 行), 分发'PX SEND BROADCAST'和接收'PX RECEIVE'. 执行计划中出现了两组 PX 进程, 除了之前蓝色的多人标志, 现在出现了红色的多人标志. 此时, SQL 的执行顺序为:

[1] 引用自白皮书 Parallel Execution with Oracle Database 12c Fundamentals: http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-bidw-parallel-execution-130766.pdf

1. 4 个红色的 PX 进程扮演生产者角色, 扫描维度表 customer, 把数据通过 broadcast 的方式分发给每一个扮演消费者的蓝色 PX 进程. 因为 DoP=4, 每一条被扫描出来的记录被复制了 4 份, 从 sql monitor 的第 9 行, customer 全表扫描返回 1.5m 行数据, 第 8 行的分发和第 7 行的接受之时, 变成了 6m 行记录, 每个作为消费者的蓝色 px 进程都持有了一份完整包含所有 custome 记录的数据, 并准备好第 5 行 hash join 的 build table.
2. 4 个作为消费者的蓝色 PX 进程, 以数据块地址区间为单位扫描事实表 lineorder(第 10/11 行); 同时和已经持有的 customer 表的数据进行 hash join(第 5 行), 然后对满足 join 条件的数据做预聚合(第 4 行), 因为我们查询的目标是对所有 lo_revenue 求和, 聚合之后每个 PX 进程只需输出一个总数.
3. 4 个蓝色的 PX 进程反过来作为生产者, 把聚合的数据发给消费者 QC(第 3 行和第 2 行). 由 QC 对接收到 4 行记录做最后的聚合, 然后返回给用户.
4. 使用 broadcast 的分发方式, 只需要把 customer 的数据广播给每个消费者. Lineorder 的数不需要重新分发. 因为 lineorder 的数据量比 customer 大的多, 应该避免对 lineorder 的数据进行分发, 这种执行计划非常适合星型模型的数据.



观察 sql monitor 报告中 Parallel 标签下的信息, 红色的 PX 进程为实例 1、2 上的 p002/p003 进程, 蓝色的 PX 进程为 p000/p001 进程, 因为蓝色的 PX 进程负责扫描事实表 lineorder, hash join 和聚合, 所以消耗几乎所有的 db time.



生产者-消费者模型工作原理

并行查询之后, 可以通过视图 V$PQ_TQSTAT, 验证以上描述的执行过程.

1. 实例 1、2 上的 p002/p003 进程作为生产者, 几乎平均扫描 customer 的 1/4 记录, 把每一条记录广播给 4 个消费者 PX 进程, 发送的记录数之和为 6m 行. 通过 table queue 0(TQ_ID=0), 每个作为消费者的 p000/p001 进程, 接收了完整的 1.5m 行 customer 记录, 接收的记录数之和为 6m 行.
2. 实例 1、2 上的 p000/p0001 进程作为生产者, 通过 table queue 1(TQ_ID=1), 把聚合的一条结果记录发给作为消费者的 QC. QC 作为消费者, 接收了 4 行记录.

```
SELECT
    dfo_number, tq_id, server_type, instance, process, num_rows
FROM
    V$PQ_TQSTAT
ORDER BY
    dfo_number DESC, tq_id, server_type desc, instance, process;

DFO_NUMBER      TQ_ID SERVER_TYPE      INSTANCE PROCESS     NUM_ROWS
---------- ---------- -------------- ---------- --------- ----------
         1          0 Producer                1 P002         1461932
         1          0 Producer                1 P003         1501892
         1          0 Producer                2 P002         1575712
         1          0 Producer                2 P003         1460464
         1          0 Consumer                1 P000         1500000
         1          0 Consumer                1 P001         1500000
         1          0 Consumer                2 P000         1500000
         1          0 Consumer                2 P001         1500000
         1          1 Producer                1 P000               1
         1          1 Producer                1 P001               1
         1          1 Producer                2 P000               1
         1          1 Producer                2 P001               1
         1          1 Consumer                1 QC                 4

13 rows selected.
```

那么, 以上的输出中, DFO_NUMBER 和 TQ_ID 这两列表示什么意思呢?

1. DFO 代表 Data Flow Operator, 是执行计划中可以并行执行的操作. 一个 QC 代表一棵 DFO 树 (tree), 包含多个 DFO; 同一个 QC 中所有并行操作的 DFO_NUMBER 是相同的, 此例中, 所有 DFO_NUMBER 为 1. 执行计划包含多个 QC 的例子也不少见, 比如使用 union all 的语句, union all 每个分支都是独立的 DFO 树, 不同的 DFO 树之间可以并行执行. 本篇文章仅讨论执行计划只有一个 QC 的情况.
2. TQ 代表 table queue, 用以 PX 进程之间或者和 QC 通信连接. 以上执行计划中, table queue 0 为 PX 进程之间的连接, table queue 1 为 PX 进程和 QC 之间的连接. 生产者通过 table queue 分发数据, 消费者从 table queue 接收数据. 不同的 table queue 编号, 代表了不同的数据分发. 通过 table queue, 我们可以理解 Oracle 并行执行使用生产者-消费者模型的本质:
    - **同一棵 DFO 树中, 最多只有两组 PX 进程.** 每个生产者进程都存在一个和每个消费者进程的连接, 每个 PX 进程和 QC 都存在一个连接. 假设 DoP=n, 连接总数为(n*n + 2*n), 随着 n 的增长, 连接总数会爆炸型增长. Oracle 并行执行设计时, 采用生产者和消费者模型, 考虑到连接数的复杂度, 每个 DFO 最多只分配两组 PX 进程. 假设 DoP=100 时, 两组 PX 进程之间的连接总数为 10000. 假设可以分配三组 PX 进程一起完成并行执行计划, 那么三组 PX 之间连接总数会等于 1 百万, 维护这么多连接, 是一个不可能的任务.
    - **同一棵 DFO 树中, 两组 PX 进程之间, 同一时间只存在一个活跃的数据分发.** 如果执行路径很长, 数据需要多次分发, 两组 PX 进程会变换生产者消费者角色, 相互协作, 完成所有并行操作. 每次数据分发, 对应的 table queue 的编号不同. 一个活跃的数据分发过程, 需要两组 PX 进程都参与, 一组为生产者发送数据, 一组为消费者接收数据. 因为一个 DFO 里最多只有两组 PX 进程, 意味着, PX 进程之间, 同一时间只能有一个活跃的数据分发. 如果 PX 进程在执行计划中需要多次分发数据, 可能需要在执行计划插入一些阻塞点, 比如 BUFFER SORT 和 HASH JOIN BUFFERED 这两个操作, 保证上一次的数据分发完成之后, 才开始下一次分发. 在后面的章节, 我将会说明这些阻塞点带来什么影响. 这个例子中, table queue 0 和 1 可以同时工作是因为: table queue 0 是两组 PX 进程之间的链接, table queue 1 为 PX 进程和 QC 之间的连接, table queue 0 与 table queue 1 是相互独立的, 因此可以同时进行.
    - PX 进程之间或者与 QC 的连接至少存在一个(单节点下至多三个, RAC 环境下至多四个)消息缓冲区用于进程间数据交互, 该消息缓冲区默认在 Large pool 中分配(如果没有配置 Large pool 则在 Shared pool 中分配). 多个缓冲区是为了实现异步通信, 提高性能.
    - 每个消息缓冲区的大小由参数 parallel_execution_message_size 控制, 默认为 16k。
    - 当两个进程都在同一个节点的时候, 通过在 Large pool(如果没有配置 Large pool 则 Shared pool)中传递和接收消息缓冲进行数据交互. 当两个进程位于不同节点时. 通过

RAC 心跳网络进行数据交互, 其中一方接收的数据需要缓存在本地 Large pool(如果没有配置 Large pool 则 Shared pool)里面.

小结

为了说明并行执行的生产者-消费者模型是如何工作的, 我使用了 broadcast 分发, QC 分配两组 PX 进程, 一组为生产者, 一组为消费者. QC 和 PX 进程之间, 两组 PX 进程之间通过 table queue 进行数据分发, 协同完成整个并行执行计划. 视图 V$PQ_TQSTAT 记录了并行执行过程中, 数据是如何分发的. 通过对 DFO, table queue 的描述, 我阐述生产者-消费者模型的工作原理和通信过程, 或许有些描述对你来说过于突然, 不用担心, 后面的章节我会通过更多的例子来辅助理解.

如何阅读并行执行计划

Table queue 的编号代表了并行执行计划中, 数据分发的顺序. 理解执行计划中的并行操作是如何被执行的, 原则很简单: **跟随 Table queue 的顺序**.

通过 sql monitor 报告判断 sql 的执行顺序, 需要结合 name 列的 table queue 名字比如:TQ10000(代表 DFO=1, table queue 0),:TQ10001(代表 DFO=1, table queue 1), 还有 PX 进程的颜色, 进行确定.

下面的例子为 dbms_xplan.display_cursor 的输出. 对于并行执行计划, 会多出来三列:

1. TQ 列: 为 Q1:00 或者 Q1:01, 其中 Q1 代表第一个 DFO, 00 或者 01 代表 table queue 的编号.
    a. ID 7~9 的操作的 TQ 列为 Q1,00, 该组 PX 进程, 作为生产者首先执行, 然后通过 broadcast 的分发方式, 把数据发给消费者.
    b. ID 10~11, 3~6 的操作的 TQ 列为 Q1,01, 该组 PX 进程作为消费者接受 customer 的数据之后, 扫描 lineorder, hash join, 聚合之后, 又作为生产者通过 table queue 2 把数据发给 QC.
2. In-out 列: 表明数据的流动和分发.
    - PCWC: parallel combine with child.
    - PCWP: parallel combine with parent.
    - P->P: parallel to parallel.
    - P->S: parallel to Serial.
3. PQ Distribute 列: 数据的分发方式. 此执行计划中, 我们使用了 broadcast 的方式, 下面的章节我会讲述其他的分发方式.

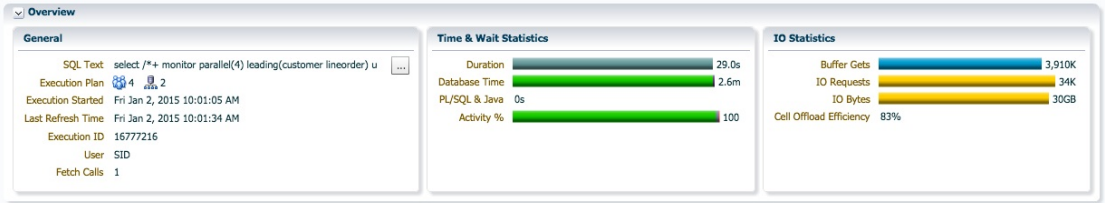
```
SQL_ID  gawcgdkuc3b3u, child number 0
-------------------------------------
select /*+ monitor parallel(4) */     sum(lo_revenue) from
lineorder, customer where     lo_custkey = c_custkey

Plan hash value: 1559646891

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name      | Rows  | Bytes | Cost (%CPU)| Time     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |           |       |       |   295K(100)|          |        |      |            |
|   1 |  SORT AGGREGATE               |           |     1 |    18 |            |          |        |      |            |
|   2 |   PX COORDINATOR              |           |       |       |            |          |        |      |            |
|   3 |    PX SEND QC (RANDOM)        | :TQ10001  |     1 |    18 |            |          | Q1,01  | P->S | QC (RAND)  |
|   4 |     SORT AGGREGATE            |           |     1 |    18 |            |          | Q1,01  | PCWP |            |
|* 5 |      HASH JOIN                |           |  294M|  5052M|   295K  (1)| 00:00:12 | Q1,01  | PCWP |            |
|   6 |       PX RECEIVE              |           | 1500K| 8789K|  1558   (1)| 00:00:01 | Q1,01  | PCWP |            |
|   7 |        PX SEND BROADCAST      | :TQ10000  | 1500K| 8789K|  1558   (1)| 00:00:01 | Q1,00  | P->P | BROADCAST  |
|   8 |         PX BLOCK ITERATOR     |           | 1500K| 8789K|  1558   (1)| 00:00:01 | Q1,00  | PCWC |            |
|* 9 |          TABLE ACCESS STORAGE FULL| CUSTOMER  | 1500K| 8789K|  1558   (1)| 00:00:01 | Q1,00  | PCWP |            |
|  10 |       PX BLOCK ITERATOR       |           |  300M|  3433M|   293K  (1)| 00:00:12 | Q1,01  | PCWC |            |
|* 11 |        TABLE ACCESS STORAGE FULL | LINEORDER |  300M|  3433M|   293K  (1)| 00:00:12 | Q1,01  | PCWP |            |
-----------------------------------------------------------------------------------------------------------------------

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

   5 - access("LO_CUSTKEY"="C_CUSTKEY")
   9 - storage(:Z>=:Z AND :Z<=:Z)
  11 - storage(:Z>=:Z AND :Z<=:Z)
```

HASH 分发方式, 两次数据分发

除了 broadcast 分发方式, 另一种常见的并行分发方式为 hash. 为了观察使用 hash 分发时 sql 的执行情况, 我对 sql 使用 pq_distribute hint.

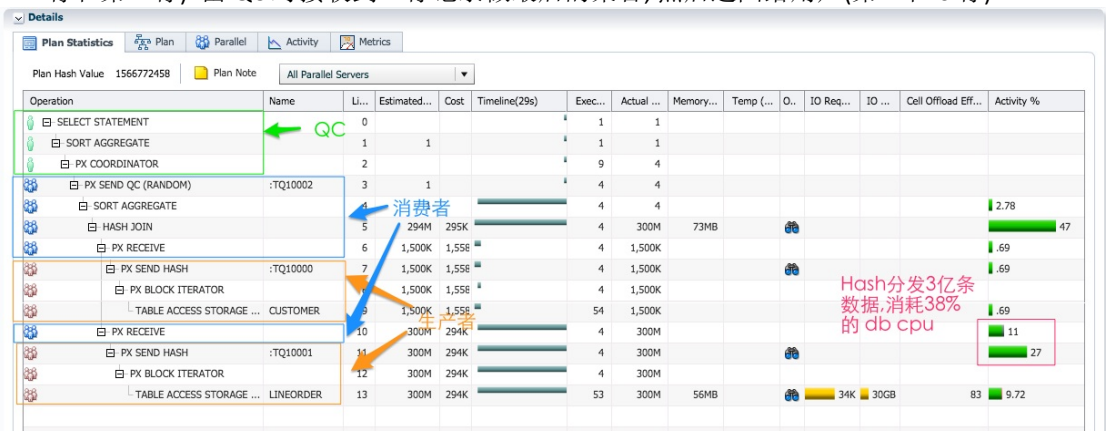
```
select /*+ monitor parallel(4)
        leading(customer lineorder)
        use_hash(lineorder)
        pq_distribute(lineorder hash hash) */
    sum(lo_revenue)
from
    lineorder, customer
where
    lo_custkey = c_custkey;
```

使用 hash 分发方式时, sql 的执行时间为 29s, db time 为 2.6m. 相对于 broadcast 方式, sql 的执行时间和 db time 都增加了大约 40%.

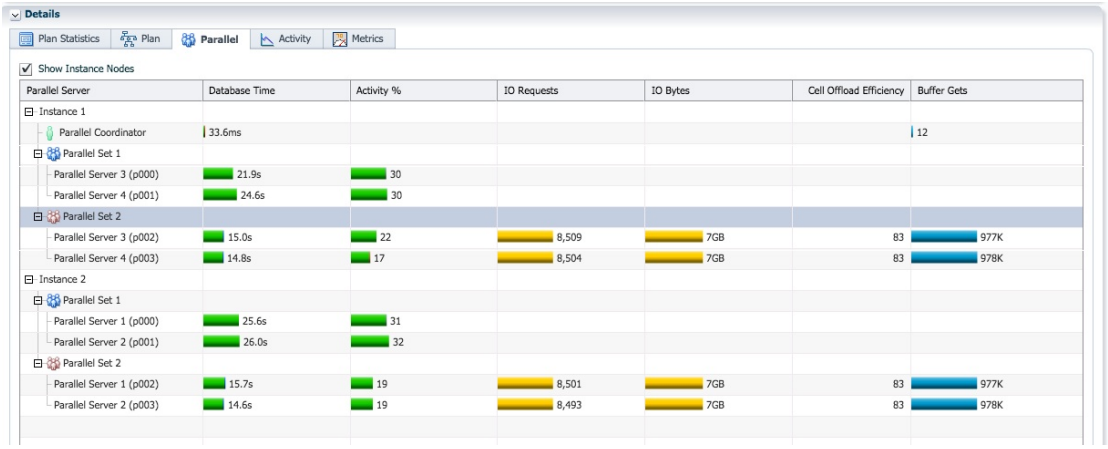


执行计划如下, 执行计划为 14 行, 增加了对 lineorder 的 hash 分发, 第 11 行的'PX SEND HASH'对 3 亿行数据通过 hash 函数分发, 第 10 行的'PX RECEIVE' 通过 table queue 1 接收 3 亿行数据, 这两个操作消耗了 38%的 db cpu. 这就是为什么 SQL 执行时间和 db time 变长的原因. 此时, SQL 的执行顺序为:

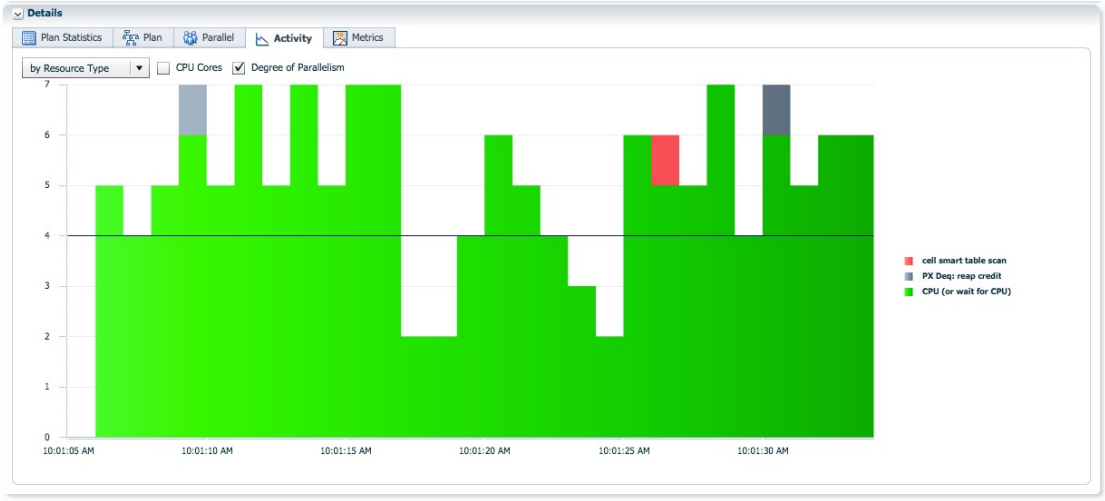
1. 红色的 PX 进程作为生产者, 并行扫描 customer(第 8~9 行), 对于连接键 c_custkey 运用 hash 函数, 根据每行记录的 hash 值, 通过 table queue 0, 发给 4 个蓝色消费者的其中一个(第 7 行). Hash 分发方式并不会复制数据, sql monitor 报告的第 6~9 行, actual rows 列都为 1.5m.
2. 红色的 PX 进程作为生产者, 并行扫描 lineorder(第 12~13 行), 对于连接键 lo_custkey 运用同样的 hash 函数, 通过 table queue 1, 发给 4 个蓝色消费者的其中一个(第 11 行). 同样的 hash 函数保证了 customer 和 lineorder 相同的连接键会发给同一个消费者, 保证 hash join 结果的正确. 因为 3 亿行数据都需要经过 hash 函数计算, 然后分发(这是进程间的通信, 或者需要通过 RAC 心跳网络通信), 这些巨大的额外开销, 就是增加 38% cpu 的原因.
3. 4 个蓝色的 PX 进程作为消费者接收了 customer 的 1.5M 行记录 (第 6 行), 和 lineorder 的 3 亿行记录(第 10 行), 进行 hash join(第 5 行), 预聚合(第 4 行).
4. 4 个蓝色的 PX 进程反过来作为生产者,通过 table queue 2, 把聚合的数据发给消费者 QC(第 3 行和第 2 行). 由 QC 对接收到 4 行记录做最后的聚合, 然后返回给用户(第 1 和 0 行).



观察 sql monitor 报告中 Parallel 标签下的信息, 红色的 px 进程为实例 1、2 上的 p002/p003 进程, 蓝色的 PX 进程为 p000/p001 进程. 作为生产者的红色 PX 进程负责扫描事实表 lineorder, 对 3 亿行数据进行 hash 分发, 占了超过 1/3 的 db time.



因为涉及 3 亿行数据的分发和接收, 作为生产者的红色 PX 进程和作为消费者的蓝色 PX 进程需要同时活跃, SQL monitor 报告中的 activity 信息显示大部分时间, AAS 超过并行度 4, 意味这两组 PX 进程同时工作. 不像 replicate 或者 broadcast 分发时, AAS 为 4, 只有一组 PX 进程保持活跃.



并行查询之后, 通过视图 V$PQ_TQSTAT, 进一步验证以上描述的执行过程. 并行执行过程涉及 3 个 table queue 0/1/2, V$PQ_TQSTAT 包含 21 行记录.

1. 实例 1、2 上的 p002/p003 进程作为生产者, 平均扫描 customer 的 1/4 记录, 然后通过 table queue 0(TQ_ID=0), 发给作为消费者的 p000/p001 进程. 发送和接收的 customer 记录之和都为 1.5m.
    - 发送的记录数: 1500000 = 365658 + 364899 + 375679 + 393764
    - 接收的记录数: 1500000 = 374690 + 374924 + 375709 + 374677
2. 实例 1、2 上的 p002/p0003 进程作为生产者, 平均扫描 lineorder 的 1/4 记录, 通过 table queue 1(TQ_ID=1), 发给作为消费者的 p000/p001 进程. 发送和接收的 lineorder 记录之和都为 300005811.
    - 发送的记录数: `300005811 = 74987629 + 75053393 + 74979748 + 74985041`
    - 接收的记录数: `300005811 = 74873553 + 74968719 + 75102151 + 75061388`
3. 实例 1、2 上的 p000/p0001 进程作为生产者, 通过 table queue 2(TQ_ID=2), 把聚合的一条结果记录发给作为消费者的 QC. QC 作为消费者, 接收了 4 行记录.

```
SELECT
    dfo_number, tq_id, server_type, instance, process, num_rows
FROM
    V$PQ_TQSTAT
ORDER BY
    dfo_number DESC, tq_id, server_type desc, instance, process;

DFO_NUMBER      TQ_ID SERVER_TYPE         INSTANCE PROCESS     NUM_ROWS
---------- ---------- ---------------- ---------- --------- ----------
         1          0 Producer                  1 P002          365658
```

```
         1          0 Producer                1 P003          364899
         1          0 Producer                2 P002          375679
         1          0 Producer                2 P003          393764
         1          0 Consumer                1 P000          374690
         1          0 Consumer                1 P001          374924
         1          0 Consumer                2 P000          375709
         1          0 Consumer                2 P001          374677
         1          1 Producer                1 P002        74987629
         1          1 Producer                1 P003        75053393
         1          1 Producer                2 P002        74979748
         1          1 Producer                2 P003        74985041
         1          1 Consumer                1 P000        74873553
         1          1 Consumer                1 P001        74968719
         1          1 Consumer                2 P000        75102151
         1          1 Consumer                2 P001        75061388
         1          2 Producer                1 P000               1
         1          2 Producer                1 P001               1
         1          2 Producer                2 P000               1
         1          2 Producer                2 P001               1
         1          2 Consumer                1 QC                 4

21 rows selected.
```

小结

我们观察 hash 分发时 sql 的并行执行过程. Hash 分发与 broadcast 最大的区分在于对 hash join 的两边都进行分发. 这个例子中, 对 lineorder 的 hash 分发会增加明显的 db cpu. 下一节, 我将使用另一个例子, 说明 hash 分发适用的场景.

Replicate, Broadcast 和 Hash 的选择

我们已经测试过 replicate, broadcast, 和 hash 这三种分发方式.

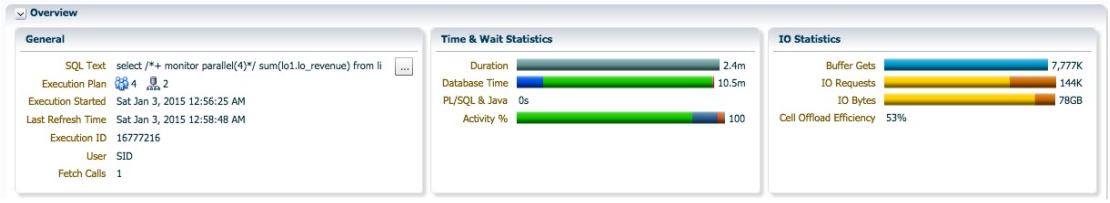
1. Replicate: 每个 PX 进程重复扫描 hash join 的左边, buffer cache 被用来缓存 hash join 左边的小表, 减少重复扫描所需的物理读. 相对于 broadcast 分发, replicate 方式只需一组 PX 进程. 但是 replicate 不能替换 broadcast 分发. 因为 replicate 仅限于 hash join 左边是表的情况, 如果 hash join 的左边的结果集来自其他操作, 比如 join 或者视图, 那么此时无法使用 replicate.
2. Broadcast 分发: 作为生产者的 PX 进程通过广播的方式, 把 hash join 左边的结果集分发给每个作为消费者的 PX 进程. 一般适用于 hash join 左边结果集比右边小得多的场景, 比如星型模型.
3. Hash 分发的本质: 把 hash join 的左边和右边(两个数据源), 通过同样 hash 函数重新分发, 切分为 N 个工作单元(假设 DoP=N), 再进行 join, 目的是减少 PX 进程进行 join 操作时, 需要连接的数据量. Hash 分发的代价需要对 hash join 的两边都进行分发. 对于 customer 连接 lineorder 的例子, 因为维度表 customer 的数据量比事实表 lineorder 小得多, 对 customer 进行 replicate 或者 broadcast 分发显然是更好的选择, 因为这两种方式不用对 lineorder 进行重新分发. 如果是两个大表 join 的话, join 操作会是整个执行计划的瓶颈所在, hash 分发是唯一合适的方式. 为了减低 join 的代价, 对 hash join 左边和右边都进行 hash 分发的代价是可以接受的.

Hash 分发, 有时是唯一合理的选择

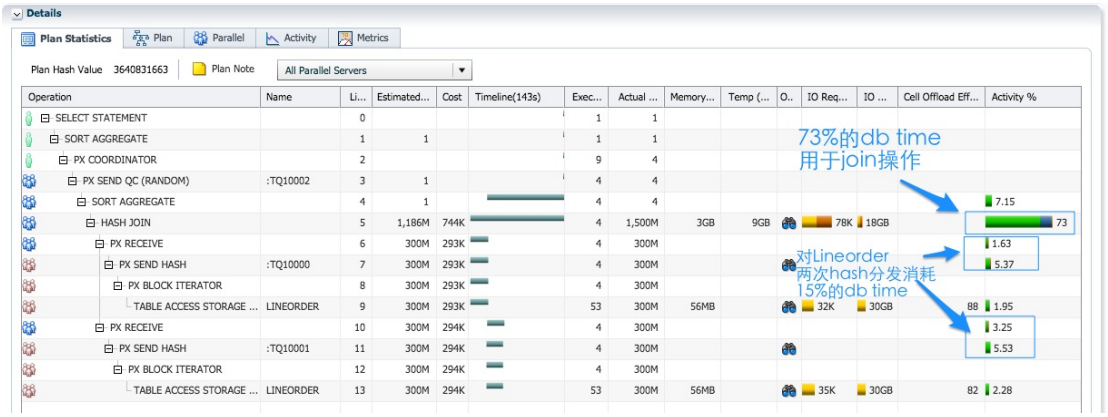
我们使用 lineorder 上的自连接来演示, 为什么有时 hash 分发是唯一合理的选择. 测试的 SQL 如下:

```
select /*+ monitor parallel(4)*/
    sum(lo1.lo_revenue)
from
    lineorder lo1, lineorder lo2
where
    lo1.lo_orderkey = lo2.lo_orderkey;
```

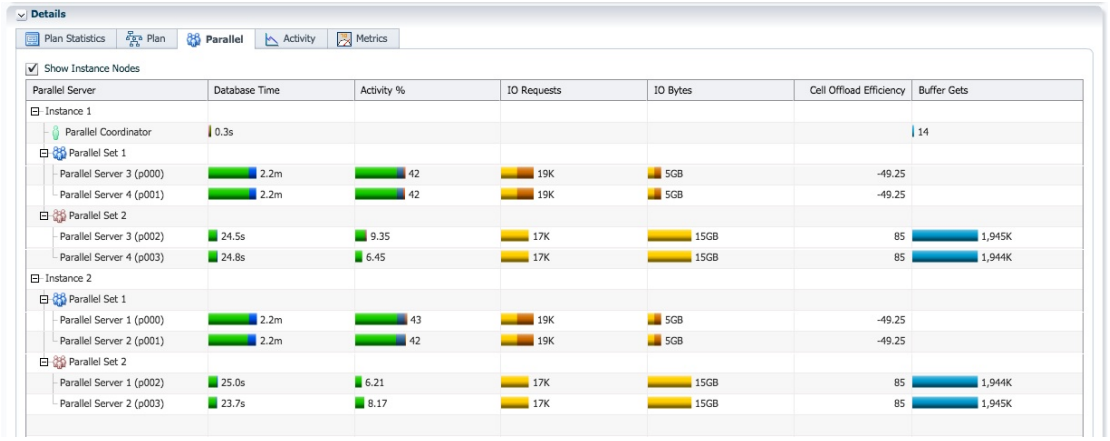
SQL 执行时间为 2.4 分钟, db time 为 10.5 分钟.



优化器默认选择 hash 分发方式, 执行计划为 14 行, 结构与之前的 Hash 分发的例子是一致的. 不同的是, 第 5 行的 hash join 消耗了 73%的 db time, 使用了 9GB 的临时表空间, 表空间的 IO 占 12%的 db time. 大约 15%的 db time 用于 Lineorder 的两次 hash 分发和接收, 相对上一个例子的占 38% 比例, 这两次 HASH 分发的整体影响降低了一倍多.



红色的 PX 进程为实例 1、2 上的 p002/p003 进程, 蓝色的 PX 进程为 p000/p001 进程.作为生产者的红色 PX 进程占总 db time 的 15%左右.



SQL 执行开始, 对 lineorder 两次 hash 分发时, AAS 大于 4, 分发完成之后, 只有蓝色的 PX 进程进行 hash join 操作, AAS=4.

从 V$PQ_TQSTAT 视图可以确认, 对于 lineorder 的存在两次分发, 通过 table queue 0 和 1, 作为消费者的 4 个 PX 进程接收到的两次数据是一样的, 保证重新分发不会影响 join 结果的正确性. 每个蓝色 PX 进程需要 hash join 的左边和右边均为 3 亿行数据的 1/4, 通过 hash 分发, 3 亿行记录连接 3 亿行记录的工作平均的分配四个独立 PX 进程各自处理, 每个 PX 进程处理 75M 行记录连接 75M 行记录.

```
SELECT
    dfo_number, tq_id, server_type, instance, process, num_rows
FROM
    V$PQ_TQSTAT
ORDER BY
    dfo_number DESC, tq_id, server_type desc, instance, process;

DFO_NUMBER      TQ_ID SERVER_TYPE      INSTANCE PROCESS      NUM_ROWS
---------- ---------- -------------- ---------- --------- ----------
         1          0 Producer                1 P002        75055725
         1          0 Producer                1 P003        74977459
         1          0 Producer                2 P002        74995276
         1          0 Producer                2 P003        74977351
         1          0 Consumer                1 P000        74998419
         1          0 Consumer                1 P001        74995836
         1          0 Consumer                2 P000        74976974
         1          0 Consumer                2 P001        75034582
         1          1 Producer                1 P002        74986798
         1          1 Producer                1 P003        74985268
         1          1 Producer                2 P002        74984883
         1          1 Producer                2 P003        75048862
         1          1 Consumer                1 P000        74998419
         1          1 Consumer                1 P001        74995836
         1          1 Consumer                2 P000        74976974
         1          1 Consumer                2 P001        75034582
         1          2 Producer                1 P000               1
         1          2 Producer                1 P001               1
         1          2 Producer                2 P000               1
         1          2 Producer                2 P001               1
         1          2 Consumer                1 QC                 4

21 rows selected.
```

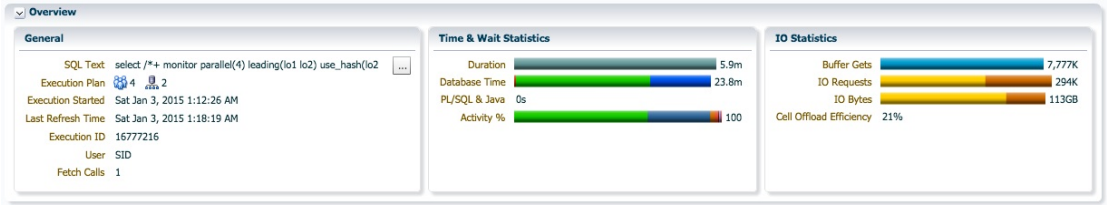
使用 broadcast 分发, 糟糕的性能

对于 lineorder, lineorder 的自连接, 如果我们使用 broadcast 分发, 会出现什么情况呢? 我们测试一下:

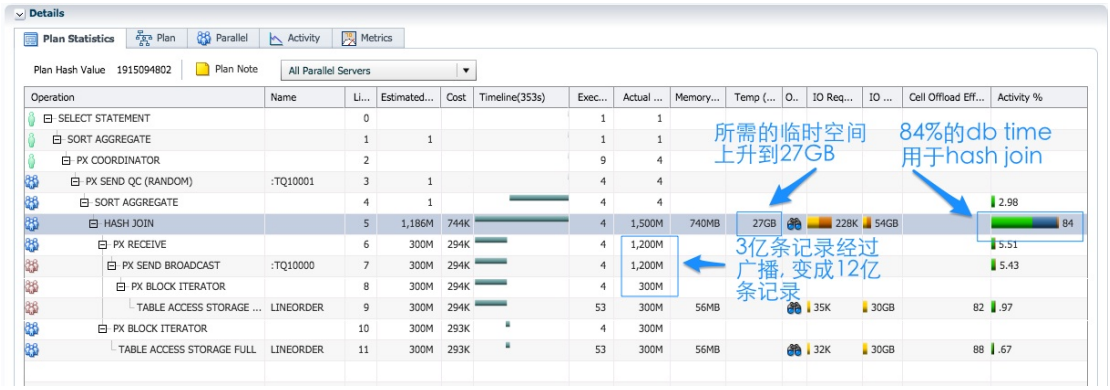
```
select /*+ monitor parallel(4)
    leading(lo1 lo2)
    use_hash(lo2)
    pq_distribute(lo2 broadcast none) */
```

```
sum(lo1.lo_revenue)
from
    lineorder lo1, lineorder lo2
where
    lo1.lo_orderkey = lo2.lo_orderkey;
```

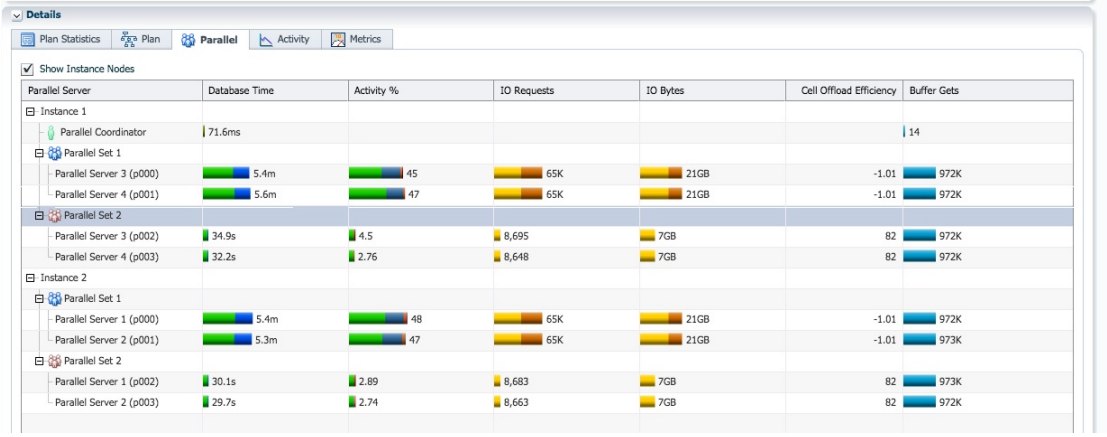
使用 broadcase 分发, SQL 的执行时间为 5.9 分钟, db time 为 23.8 分钟. 相比 hash 分发, 执行时间和 db time 都增加了接近 1.5 倍.



红色的 PX 进程作为生产者, 对 lineorder 进行并行扫描之后, 3 亿行记录通过 table queue 0 广播给 4 个作为消费者的蓝色 PX 进程(第 6~9 行), 相当于复制了 4 份, 每个蓝色的 PX 进程都接收了 3 亿行记录. 这次 broadcast 分发消耗了 11%的 db time, 因为需要每行记录传输给每个蓝色 PX 进程, 消耗的 db cpu 比使用 hash 分发时两次 hash 分发所消耗的还多.
第 5 行的 hash join 的所消耗的临时表空间上升到 27GB, 临时表空间 IO 占的 db time 的 38%. 因为每个蓝色 PX 进程进行 hash join 的数据变大了, hash join 的左边为 3 亿行数据, hash join 的右边为 3 亿行记录的 1/4.



蓝色 PX 进程为消费者负责 hash join, 所消耗的 db time 都大幅增加了.



hash join 时, 临时表空间读等待事件'direct path read temp'明显增加了.

V$PQ_TQSTAT 的输出中, 实例 1、2 上的 p000/p001 进程作为消费者, 都接收了 3 亿行数据, 造成后续 hash join 的急剧变慢. Broadcast 分发对 hash join 左边进行广播的机制, 决定了它不适合 hash join 两边都为大表的情况.

```
SELECT
    dfo_number, tq_id, server_type, instance, process, num_rows
FROM
    V$PQ_TQSTAT
ORDER BY
    dfo_number DESC, tq_id, server_type desc, instance, process;

DFO_NUMBER      TQ_ID SERVER_TYPE      INSTANCE PROCESS     NUM_ROWS
---------- ---------- -------------- ---------- --------- ----------
         1          0 Producer                1 P002       299928364
         1          0 Producer                1 P003       299954384
         1          0 Producer                2 P002       300188788
         1          0 Producer                2 P003       299951708
         1          0 Consumer                1 P000       300005811
         1          0 Consumer                1 P001       300005811
         1          0 Consumer                2 P000       300005811
         1          0 Consumer                2 P001       300005811
         1          1 Producer                1 P000               1
         1          1 Producer                1 P001               1
         1          1 Producer                2 P000               1
         1          1 Producer                2 P001               1
         1          1 Consumer                1 QC                 4

13 rows selected.
```

小结, Broadcast 和 Hash 分发的陷阱

通过前一节和本节的例子, 我们知道, 如果选择了不合理的分发方式, SQL 执行时性能会明显下降

1. 对于 broadcast 分发: 只对 hash join 的左边进行分发, 但是采用广播分发, hash join 时左边的数据量并没有减少, 如果 hash join 左边的包含大量数据, 并行对 hash join 性能改善有限. 对大量数据的 broadcast 分发也会消耗额外的 db cpu, 比如本节中 lineorder 自连接的例子. Replicate 同理.
2. 对于 hash 分发: 对 hash join 的两边都进行分发, 使每个 PX 进程进行 hash join 时, 左边和右边的数据量都为原始的 1/N, N 为并行度. Hash 分发的潜在陷阱在于:
    - 两次分发, 尤其对大表的分发, 可能带来明显的额外开销, 比如前一节 customer 连接 lineorder 的例子. 使用 Partition wise join 可以消除分发的需要, 后面会举例说明.
    - 如果数据存在倾斜, 连接键上的少数值占了大部分的数据, 通过 hash 分发, 同一个键值的记录会分发给同一个 PX 进程, 某一个 PX 进程会处理大部分数据的 hash join, 引起并行执行倾斜. 我会在后面的章节说明这种情况和解决方法.

SQL 解析时, 优化器会根据 hash join 左边和右边估算的 cardinality, 并行度等信息, 选择具体何种分发方式. 维护正确的统计信息, 对于优化器产生合理的并行执行计划是至关重要的.

Partition Wise Join, 消除分发的额外开销

无论对于 broadcast 或者 hash 分发, 数据需要通过进程或者节点之间通信的完成传输, 分发的数据越多, 消耗的 db cpu 越多. **并行执行时, 数据需要分发, 本质上是因为 Oracle 采用 share-everything 的集中存储架构, 任何数据对每个实例的 PX 进程都是共享的. 为了对 hash join 操作分而治之, 切分为 N 个独立的工作单元(假设 DoP=N), 必须提前对数据重新分发, 数据的分发操作就是并行带来的额外开销.**
使用 full 或者 partial partition wise join 技术, 可以完全消除分发的额外开销, 或者把这种开销降到最低. 如果 hash join 有一边在连接键上做 hash 分区, 那么优化器可以选择对分区表不分发, 因为 hash 分区已经对数据完成切分, 这只需要 hash 分发 hash join 的其中一边, 这是 partial partition wise join. 如果 hash join 的两边都在连接键上做了 hash join 分区, 那么每个 PX 进程可以独立的处理对等的 hash 分区, 没有数据需要分发, 这是 full partition wise join. hash 分区时, hash join 的工作单元就是对等 hash 分区包含的数据量, 应该控制每个分区的大小, hash join 时就可能消除临时表空间的使用, 大幅减少所需的 PGA.

Partition Wise Join, 不需要数据分发.

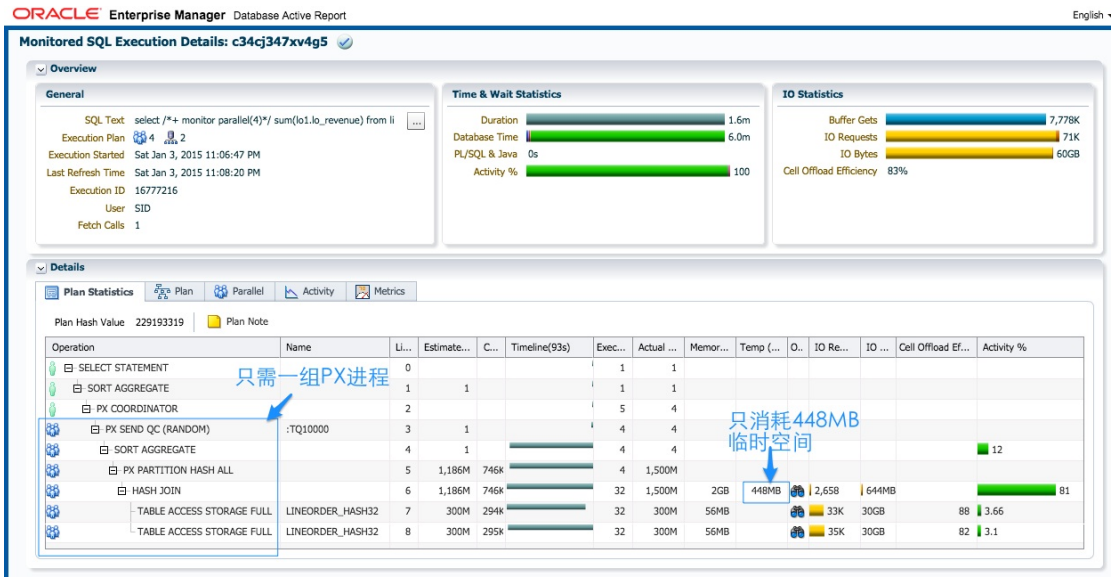
如果在 lineorder 的列 lo_orderkey 上做 hash 分区, 分区数为 32 个. 每个分区的大小接近 1G.

```
sid@SSB> @seg lineorder_hash32

SEGMENT_NAME       SEG_PART_NAME   SEGMENT_TYPE            SEG_MB
------------------ --------------- ------------------- ----------
LINEORDER_HASH32   SYS_P3345       TABLE PARTITION            960
LINEORDER_HASH32   SYS_P3344       TABLE PARTITION            960
...
LINEORDER_HASH32   SYS_P3315       TABLE PARTITION            960
LINEORDER_HASH32   SYS_P3314       TABLE PARTITION            960
                                                      ----------
                                                           30720

32 rows selected.
```

使用 lo_orderkey 连接时, lineorder 不需要再分发. 我们继续使用自连接的 sql, 演示 full partition wise join.

```
select /*+ monitor parallel(4)*/
    sum(lo1.lo_revenue)
from
    lineorder_hash32 lo1, lineorder_hash32 lo2
where
    lo1.lo_orderkey = lo2.lo_orderkey;
```

此时 sql 执行时间为 1.6 分钟, db time 6 分钟; 不分区使用 hash 分发时, 执行时间为 2.4 分钟, db time 10.5 分钟. 使用 Partition Wise join 快了三分之一. 执行计划中只有一组蓝色的 PX 进程, 不需要对数据进行分发. 因为 lineorder_hash32 的 3 亿行数据被切分为 32 个分区. 虽然并行度为 4, 每个 PX 进程 hash join 时, 工作单元为一对匹配的 hash 分区, 两边的数据量都为 3 亿的 1/32. 更小的工作单元, 使整个 hash join 消耗的临时表空间下降为 448MB. 每个 PX 进程消耗 8 对 hash 分区, 可以预见, 当我们把并行度提高到 8/16/32, 每个 PX 进程处理的 hash 分区对数, 应该分别为 4/2/1, sql 执行时间会线性的下降.



蓝色的 PX 进程为、的 p000/p001 进程. 每个 PX 进程消耗的 db time 是平均的, 每个 PX 进程均处理了 8 对分区的扫描和 hash join.

AAS 绝大部分时间都为 4.

唯一的数据连接为 table queue 0, 每个 PX 进程向 QC 发送一行记录.

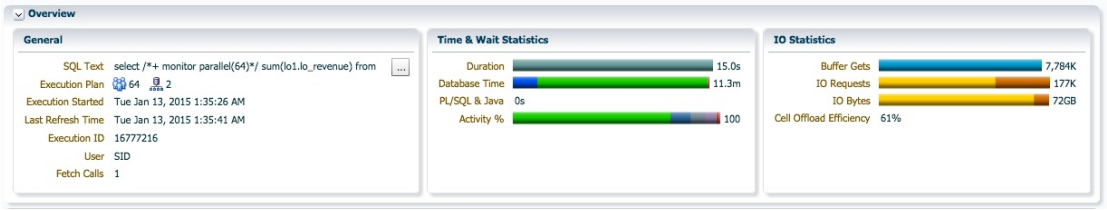
```
SELECT
    dfo_number, tq_id, server_type, instance, process, num_rows
FROM
    V$PQ_TQSTAT
ORDER BY
    dfo_number DESC, tq_id, server_type desc, instance, process;
DFO_NUMBER      TQ_ID SERVER_TYPE     INSTANCE PROCESS      NUM_ROWS
---------- ---------- ------------ ---------- ---------- ----------
         1          0 Producer              1 P000               1
         1          0 Producer              1 P001               1
         1          0 Producer              2 P000               1
         1          0 Producer              2 P001               1
         1          0 Consumer              1 QC                 4

5 rows selected
```

当 DoP 大于分区数时, Partition Wise Join 不会发生

当并行执行的 DoP 大于 hash 分区数时, partition wise join 不会发生, 这时优化器会使用 broadcast local 的分发. 使用 DoP=64 执行同样的 sql:

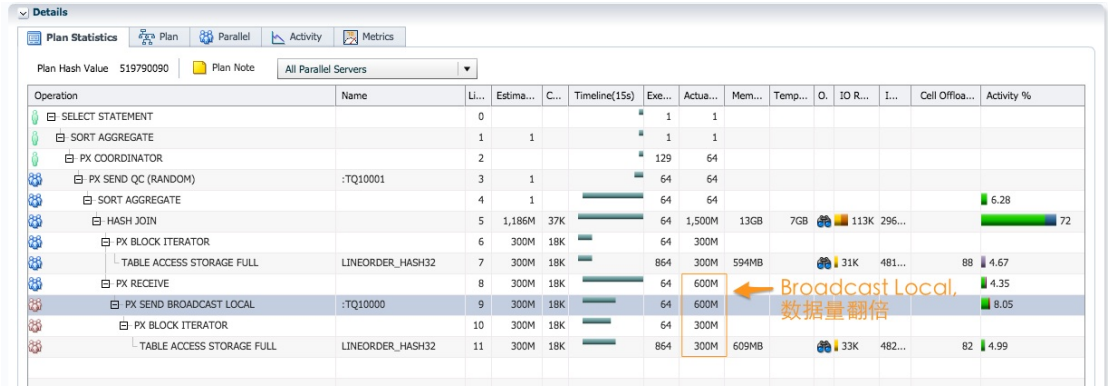
```
select /*+ monitor parallel(64)*/
    sum(lo1.lo_revenue)
from
    lineorder_hash32 lo1, lineorder_hash32 lo2
where
    lo1.lo_orderkey = lo2.lo_orderkey
```

DoP=64, 查询执行时间为 15 秒, db time 为 11.3 分钟.

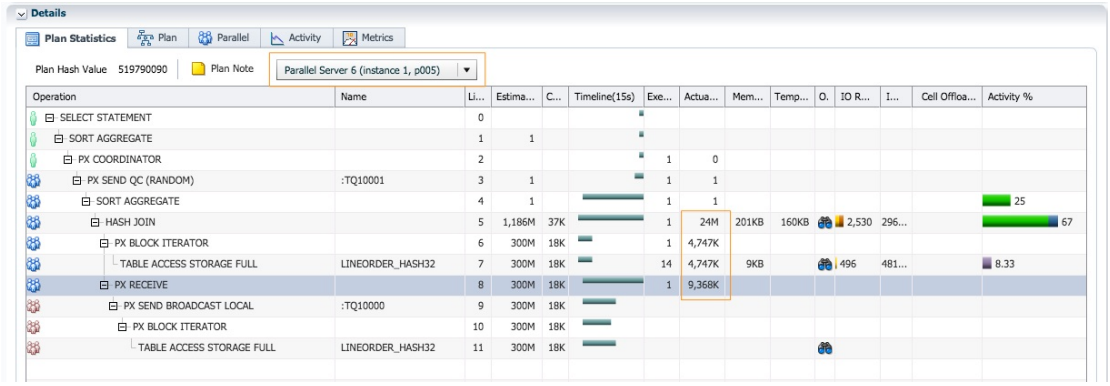


执行计划中出现了两组 PX 进程. 优化器选择对 hash join 的右边进行 broadcast local 分发. 如果 hash join 的左边比较小的话, broadcast local 会发生在 hash join 的左边. 因为 DoP 是分区数的两倍, hash join 两边的 lineorder_hash64 的每个分区, 由 2 个 PX 进程共同处理. 处理一对匹配分区的两个蓝色的 PX 进程和两个红色的 PX 进程, 会处在同一个实例上. 数据只会在同一个实例的 PX 进程之间, 不会跨实例传输, 降低数据分发成本, 这是 broadcast local 的含义. SQL 的执行顺序如下:

1. 以数据库地址区间为单位, 蓝色的 PX 进程并行扫描 hash join 左边的 lineorder_hash32(第 7 行), 因为 DoP 是分区数的两倍, 每个分区由两个蓝色 PX 进程共同扫描, 这两个 PX 进程在同一个实例上. 每个蓝色的 PX 进程大约扫描每个分区一半的数据, 大约 4.7M 行记录, 并准备好第 5 行 hash join 的 build table.
2. 红色的 PX 进程并行扫描 hash join 右边的 lineorder_hash32, 每个红色的 PX 进程大概扫描 4.7M 行记录, 然后 table queue 0, 以 broadcast local 的方式, 分发给本实例两个红色的 PX 进程(数据分发时, 映射到本实例某些 PX 进程, 避免跨节点传输的特性, 称为 slaves mapping, 除了 broadcast local, 还有 hash local, random local 等分发方式). 通过 broadcast local 分发, 数据量从 300M 行变成 600M 行.
3. 每个蓝色的 PX 进程通过 table queue 0 接收了大概 9.4M 行数据, 这是整个匹配分区的数据量. 然后进行 hash join, 以及之后的聚合操作. 每个蓝色的 PX 进程 hash join 操作时, 左边的数据量为 lineorder_hash32 的 1/64(=1/DoP), 右边的数据为 lineorder_hash32 的 1/32(=1/分区数). 如果继续提高 DoP, 只有 hash join 左边的数据量减少, 右边的数据量并不会减少; 同时, 更多的 PX 进程处理同一个分区, 会提高 broadcast 分发成本. 所以当 DoP 大于分区数时, 并行执行的随着 DoP 的提高, 扩展性并不好.



查看一个蓝色的 PX 进程, 实例 1 p005 进程的执行信息, 可以确认 hash join 的左边为 lineorder_hash32 的 1/64, hash join 的右边为 lineorder_hash32 的 1/32.



小结

数据仓库设计时，为了取得最佳的性能，应该使用 partition wise join 和并行执行的组合. 在大表最常用的连接键上，进行 hash 分区，hash join 时使优化器有机会选择 partition wise join. Range-hash 或者 list-hash 是常见的分区组合策略，一级分区根据业务特点，利用时间范围或者列表对数据做初步的切分，二级分区使用 hash 分区. 查询时，对一级分区裁剪之后，优化器可以选择 partition wise join.

设计 partition wise join 时，应该尽可能提高 hash 分区数，控制每个分区的大小. Partition wise join 时，每对匹配的分区由一个 PX 进程处理，如果分区数据太多，可能导致 join 操作时使用临时空间，影响性能. 另一方面，如果分区数太少，当 DoP 大于分区数时，partition wise join 会失效，使用更大的 DoP 对性能改善非常有限.

数据倾斜对不同分发方式的影响

数据倾斜是指某一列上的大部分数据都是少数热门的值(Popular Value). Hash join 时，如果 hash join 的右边连接键上的数据是倾斜的，数据分发导致某个 PX 进程需要处理所有热门的数据，拖长 sql 执行时间，这种情况称为并行执行倾斜. **如果优化器选择了 hash 分发，此时 join 两边的数据都进行 hash 分发，数据倾斜会导致执行倾斜.** 同值记录的 hash 值也是一样的，会被分发到同一 PX 进程进行 hash join. 工作分配不均匀，某个不幸的 PX 进程需要完成大部分的工作，消耗的 db time 会比其他 PX 进程多，SQL 执行时间会因此被明显延长. **对于 replicate 或者 broadcast 分发，则不存在这种执行倾斜的风险**，因为 hash join 右边(一般为大表)的数据不用进行分发，PX 进程使用基于数据块地址区间或者基于分区的 granule，平均扫描 hash join 右边的数据，再进行 join 操作.

为了演示数据倾斜和不同分发的关系，新建两个表，customer_skew 包含一条 c_custkey=-1 的记录，lineorder_skew 90%的记录，两亿七千万行记录 lo_custkey=-1.

```
sid@SSB> select count(*) from customer_skew where c_custkey = -1;

  COUNT(*)
----------
         1

sid@SSB> select count(*) from customer_skew;

  COUNT(*)
----------
   1500000

sid@SSB> select count(*) from lineorder_skew where lo_custkey = -1;

  COUNT(*)
----------
 270007612

sid@SSB> select count(*) from lineorder_skew;

  COUNT(*)
----------
```

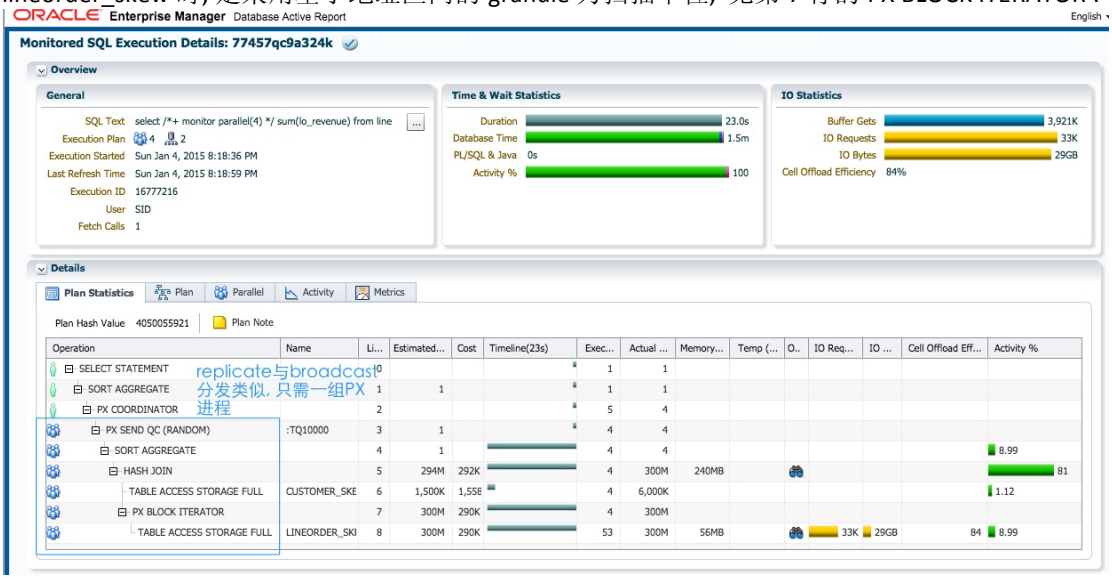
```
 300005811
```

Replicate 方式，不受数据倾斜的影响

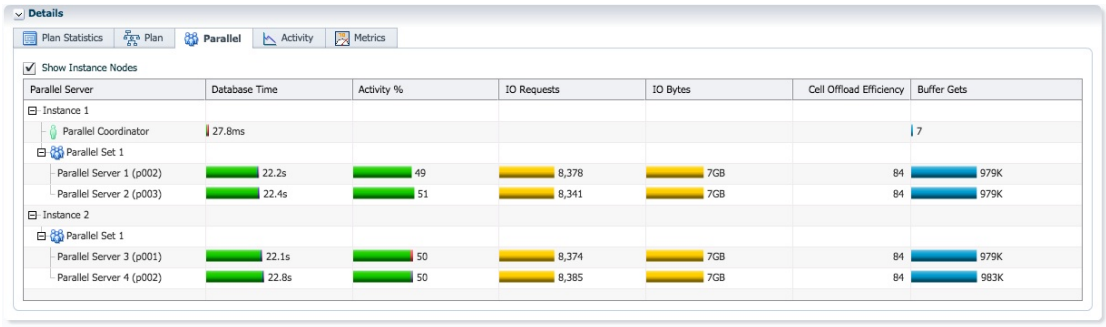
测试 sql 如下:

```
select /*+ monitor parallel(4) */
    sum(lo_revenue)
from
    lineorder_skew, customer_skew
where
    lo_custkey = c_custkey;
```

SQL 执行时间为 23 秒, db time 为 1.5m. 优化器默认的执行计划选择 replicate 的方式, 只需分配一组 PX 进程, 与 broadcast 分发的方式类似. 每个蓝色的 PX 进程重复扫描 customer, 并行扫描 lineorder_skew 时, 是采用基于地址区间的 granule 为扫描单位，见第 7 行的'PX BLOCK ITERATOR'.



4 个蓝色的 PX 进程消耗的 db time 是平均的, 对于 replicate 方式，lineorder_skew 的数据倾斜并没有造成 4 个 PX 进程的执行倾斜.



当优化器使用 replicate 方式时, 可以通过执行计划中 outline 中的 hint PQ_REPLICATE 确认. 以下部分 dbms_xplan.display_cursor 输出没有显示, 只显示 outline 数据.

```
select * from table(dbms_xplan.display_cursor('77457qc9a324k',0,'outline'));

Plan hash value: 4050055921

...

Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
```

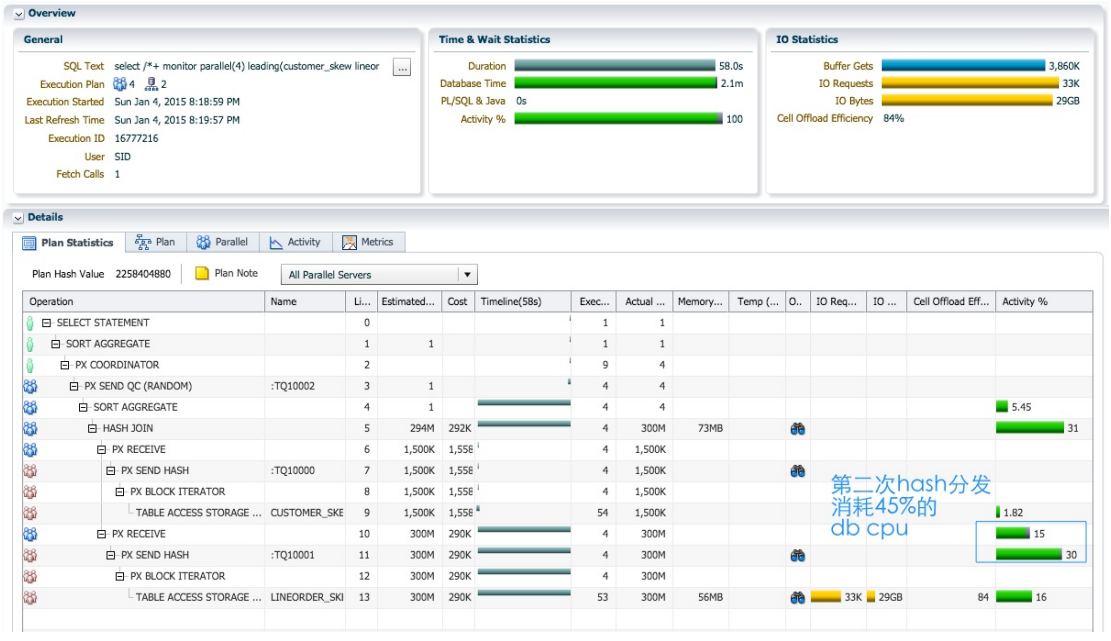
```
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('12.1.0.2')
      DB_VERSION('12.1.0.2')
      ……
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "CUSTOMER_SKEW"@"SEL$1")
      FULL(@"SEL$1" "LINEORDER_SKEW"@"SEL$1")
      LEADING(@"SEL$1" "CUSTOMER_SKEW"@"SEL$1" "LINEORDER_SKEW"@"SEL$1")
      USE_HASH(@"SEL$1" "LINEORDER_SKEW"@"SEL$1")
      PQ_DISTRIBUTE(@"SEL$1" "LINEORDER_SKEW"@"SEL$1" BROADCAST NONE)
      PQ_REPLICATE(@"SEL$1" "LINEORDER_SKEW"@"SEL$1")
      END_OUTLINE_DATA
  */
```

Hash 分发, 数据倾斜造成执行倾斜

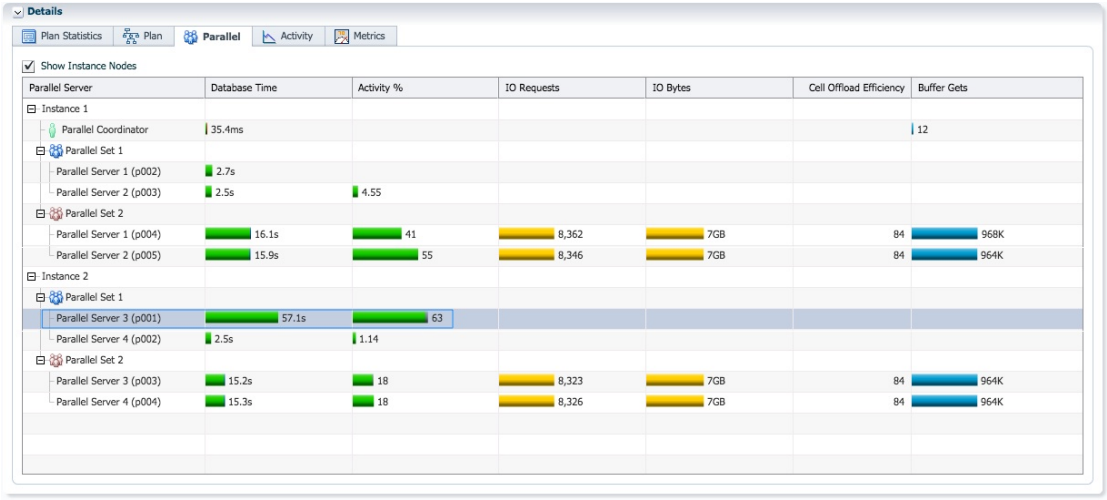
通过 hint 使用 hash 分发, 测试 sql 如下:

```
select /*+ monitor parallel(4)
    leading(customer_skew lineorder_skew)
    use_hash(lineorder_skew)
    pq_distribute(lineorder_skew hash hash) */
    sum(lo_revenue)
from
    lineorder_skew, customer_skew
where
    lo_custkey = c_custkey;
```

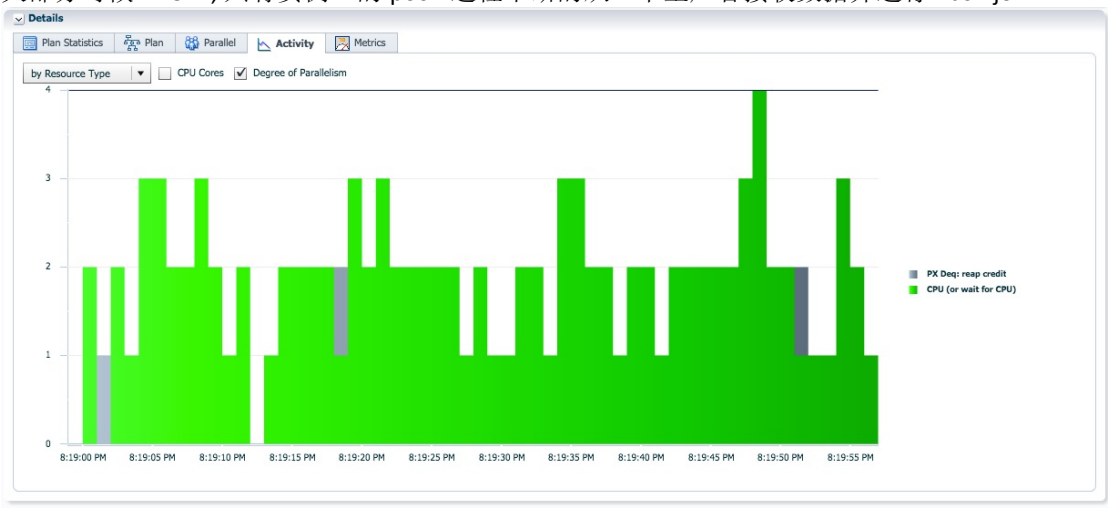
使用 hash 分发, SQL 执行时间为 58 秒, db time 2.1 分钟. 对于 replicate 时 sql 执行时间 23 秒, db time 1.5 分钟. 有趣的是, 整个 sql 消耗的 db time 只增加了 37 秒, 而执行时间确增加了 35 秒, 意味着所增加的 db time 并不是平均到每个 PX 进程的. 如果增加的 db time 平均到每个 PX 进程, 而且并行执行没有倾斜的话, 那么 sql 执行时间应该增加 37/4, 约 9 秒, 而不是现在的 35 秒.
红色的 PX 进程作为生产者, 分别对 customer_skew 和 lineorder_skew 完成并行扫描并通过 table queue0/1, hash 分发给蓝色的 PX 进程. 对 lineorder_skew 的分发, 占了 45%的 db cpu.



实例 2 的蓝色 PX 进程 p001 消耗了 57.1 秒的 db time, sql 执行时间 58 秒, 这个 PX 进程在 sql 执行过程中一直是活跃状态. 可以预见, lineorder_skew 所有 lo_custkey=-1 的数据都分发到这个进程处理. 而作为生产者的红色 PX 进程, 负责扫描 lineorder_skew 并进行分发, 它们的工作量是平均的.



大部分时候 AAS=2, 只有实例 2 的 p001 进程不断的从 4 个生产者接收数据并进行 hash join.

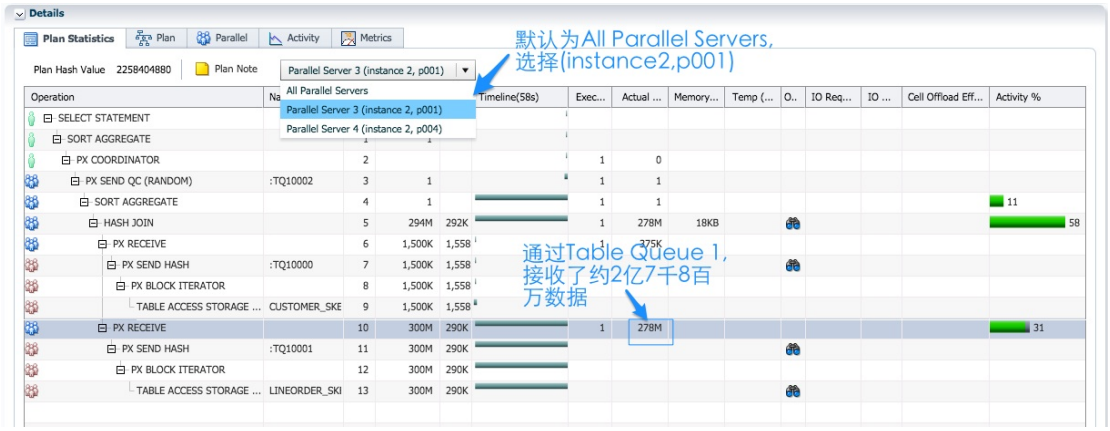


从 V$PQ_TQSTAT 视图我们可以确认, 对 hash join 右边分发时, 通过 table queue 1, 作为消费者的实例 2 的 P001, 接收了两亿七千多万的数据. 这就是该 PX 进程在整个 sql 执行过程中一直保持活跃的原因.

```
SELECT
    dfo_number, tq_id, server_type, instance, process, num_rows
FROM
    V$PQ_TQSTAT
ORDER BY
    dfo_number DESC, tq_id, server_type desc, instance, process;
DFO_NUMBER      TQ_ID SERVER_TYPE     INSTANCE PROCESS      NUM_ROWS
---------- ---------- ------------ ---------- ---------- ----------
         1          0 Producer              1 P004          375754
         1          0 Producer              1 P005          365410
         1          0 Producer              2 P003          393069
         1          0 Producer              2 P004          365767
         1          0 Consumer              1 P002          375709
         1          0 Consumer              1 P003          374677
         1          0 Consumer              2 P001          374690
         1          0 Consumer              2 P002          374924
         1          1 Producer              1 P004        75234478
         1          1 Producer              1 P005        74926098
         1          1 Producer              2 P003        74923913
         1          1 Producer              2 P004        74921322
         1          1 Consumer              1 P002         7497409
         1          1 Consumer              1 P003         7467378
         1          1 Consumer              2 P001       277538575
         1          1 Consumer              2 P002         7502449
```

```
         1          2 Producer              1 P002                  1
         1          2 Producer              1 P003                  1
         1          2 Producer              2 P001                  1
         1          2 Producer              2 P002                  1
         1          2 Consumer              1 QC                    4

21 rows selected.
```

12c 的 sql monitor 报告作了增强, 并行执行倾斜时, 包含了消耗最大的 PX 进程的采样信息. 在 plan statistics 页面, 下拉菜单选择'Parallel Server 3(instance 2, p001)', 从执行计划的第 10 行, 'PX RECEIVE', 以及 Actual Rows 列的数据 278M, 也可以确认实例 2 的 p001 进程接收了两亿七千多万数据.



小节

对于实际的应用, 处理数据倾斜是一个复杂的主题. 比如在倾斜列上使用绑定变量进行过滤, 绑定变量窥视(bind peeking)可能造成执行计划不稳定. 本节讨论了数据倾斜对不同分发方式的带来影响:

1. 通常, replicate 或者 broadcast 分发不受数据倾斜的影响.
2. 对于 hash 分发, hash join 两边连接键的最热门数据, 会被分发到同一 PX 进程进行 join 操作, 容易造成明显的并行执行倾斜.
3. 12c 引入 adaptive 分发, 可以解决 hash 分发时并行执行倾斜的问题. 我将在下一篇文章"深入理解 Oracle 的并行执行倾斜(下)"演示 adaptive 分发这个新特性.

HASH JOIN BUFFERED, 连续 hash 分发时执行计划中的阻塞点

到目前为止, 所有的测试只涉及两个表的连接. 如果多于两个表, 就需要至少两次的 hash join, 数据分发次数变多, 生产者消费者的角色可能互换, 执行计划将不可避免变得复杂. 执行路径变长, 为了保证并行执行的正常进行, 执行计划可能会插入相应的阻塞点, 在 hash join 时 , 把符合 join 条件的数据缓存到临时表, 暂停数据继续分发. 本节我使用一个三表连接的 sql 来说明连续 hash join 时, 不同分发方式的不同行为.

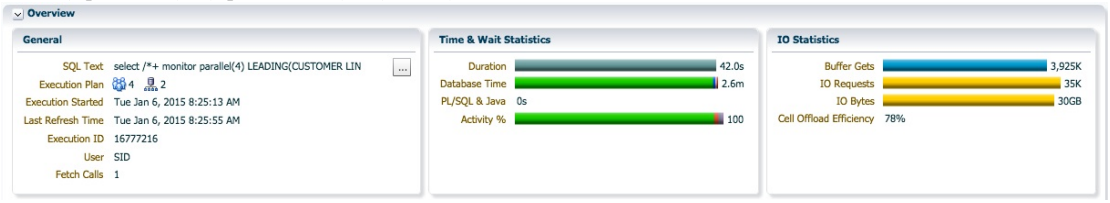
使用 Broadcast 分发, 没有阻塞点.

测试三个表连接的 sql 如下, 加入 part 表, 使用 hint 让优化器两次 hash join 都使用 broadcast 分发. Replicate SQL 查询性能类似.

```
select /*+ monitor parallel(4)
    LEADING(CUSTOMER LINEORDER PART)
    USE_HASH(LINEORDER)
    USE_HASH(PART)
    SWAP_JOIN_INPUTS(PART)
    PQ_DISTRIBUTE(PART NONE BROADCAST)
    NO_PQ_REPLICATE(PART)
    PQ_DISTRIBUTE(LINEORDER BROADCAST NONE)
    NO_PQ_REPLICATE(LINEORDER)
```

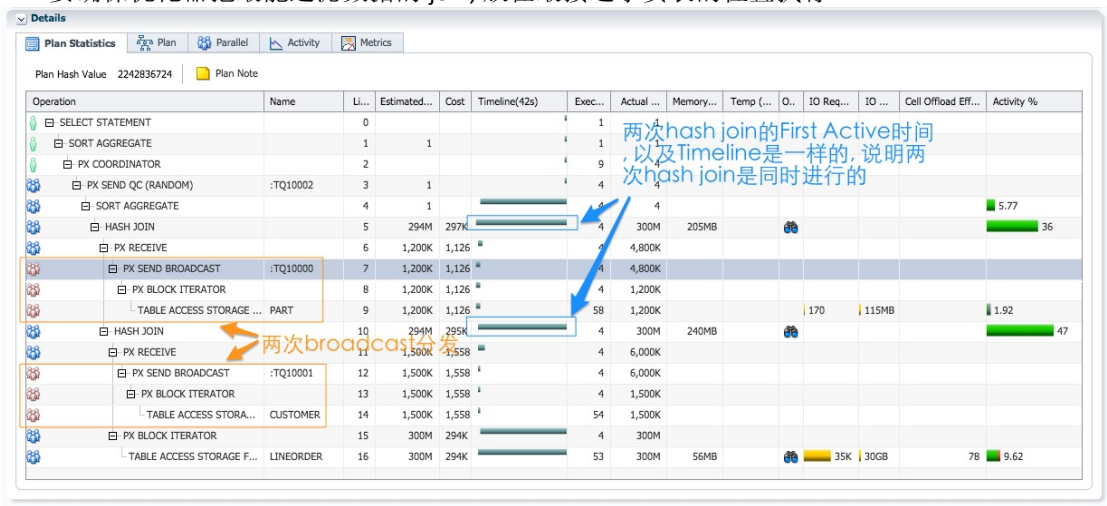
```
*/
    sum(lo_revenue)
from
    lineorder, customer, part
where
    lo_custkey = c_custkey
and lo_partkey = p_partkey;
```

SQL 执行时间为 42 秒, db time 为 2.6 分钟.
AAS=(sql db time) / (sql 执行时间) = (2.6*60) / 42 =3.7, 接近 4, 说明 4 个 PX 进程基本一直保持活跃.



执行计划是一颗完美的右深树, 这是星型模型查询时执行计划的典型形式. 生产者对两个维度进行 broadcast 分发, 消费者接受数据之后准备好两次 hash join 的 build table, 最后扫描事实表, 并进行 hash join. 我们通过**跟随 table queue 顺序的原则,** 阅读这个执行计划.

1. 红色 PX 进程作为生产者并行扫描 part, 通过 table queue 0 广播给每个蓝色的消费者 PX 进程(第 7~9 行). 每个蓝色的 PX 进程接收 part 的完整数据(第 6 行), 1.2M 行记录, 并准备好 第 5 行 hash join 的 build table.
2. 红色 PX 进程作为生产者并行扫描 customer, 通过 table queue 1 广播 broadcast 给每个蓝色的消费者 PX 进程(第 12~14 行). 每个蓝色的 PX 进程接收 customer 的完整数据(第 11 行), 1.5M 行记录, 并准备好第 10 行 hash join 的 build table.
3. 蓝色的 PX 进程并行扫描事实表 lineorder, 对每条符合扫描条件(如果 sql 语句包含对 lineorder 的过滤条件)的 3 亿行记录, 进行第 10 行的 hash join, 对于每一条通过第 10 行的 hash join 的记录, 马上进行第 5 行的 hash join, 接着再进行聚合. 从 sql monitor 报告的 Timeline 列信息, 对 lineorder 的扫描和两个 hash join 操作是同时进行的. 执行计划中没有阻塞点, 数据在执行路径上的流动不需要停下来等待. 大部分的 db cpu 消耗在两次 hash join 操作. 最优化的执行计划, 意味着经过每个 hash join 的数据越少越好. 对于这类执行计划, 你需要确保优化器把最能过滤数据的 join, 放在最接近事实表的位置执行.



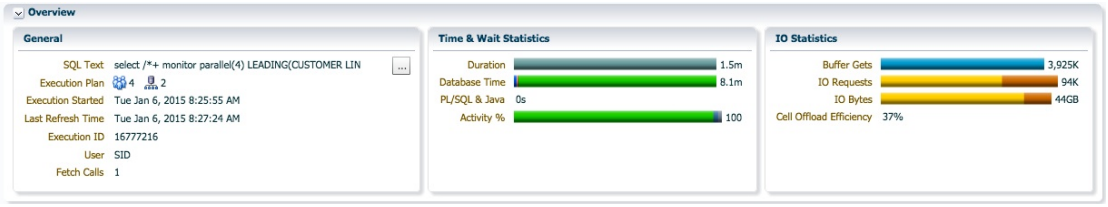
连续 hash 分发, 执行计划出现阻塞点

使用以下 hints, 强制 SQL 使用 hash 分发.

```
select /*+ monitor parallel(4)
    LEADING(CUSTOMER LINEORDER PART)
    USE_HASH(LINEORDER)
    USE_HASH(PART)
    SWAP_JOIN_INPUTS(PART)
    PQ_DISTRIBUTE(PART HASH HASH)
```

```
    PQ_DISTRIBUTE(LINEORDER HASH HASH)
*/
    sum(lo_revenue)
from
    lineorder, customer, part
where
    lo_custkey = c_custkey
and lo_partkey = p_partkey;
```

SQL 执行时间为 1.5 分钟, db time 为 8.1 分钟. 相对于增加了 14GB 的 IO 操作.



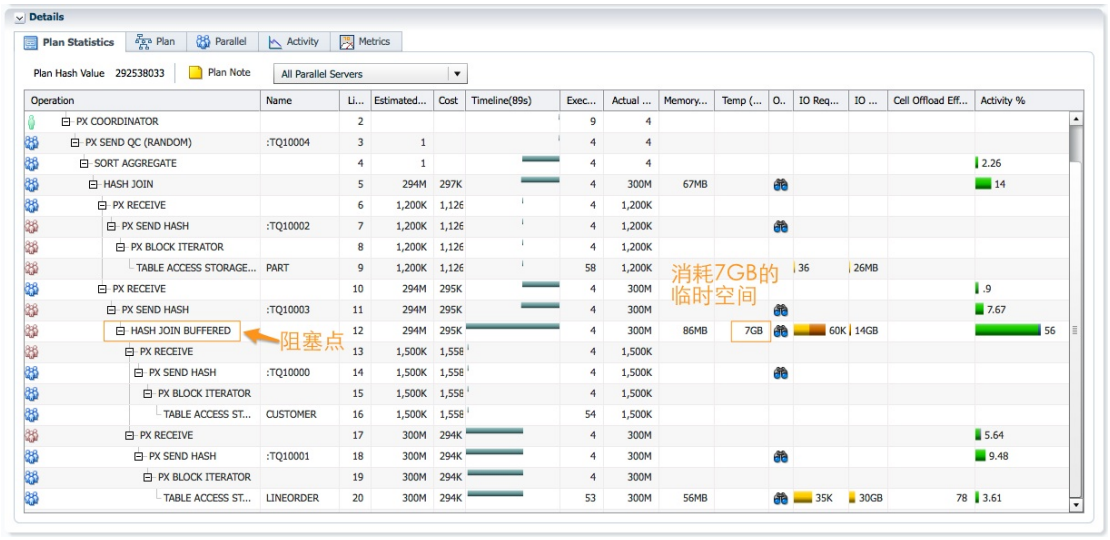
连续两次 hash join 都使用 HASH 分发, 每次 hash join 左右两边都需要分发, PX 进程之间发生 4 次数据分发. 执行计划中最显著的地方来自第 12 行的 HASH JOIN BUFFERED, 这是一个阻塞性的操作. 下面, 我们依然通过**跟随 table queue 顺序的原则**, 阅读执行计划, 并解析为什么出现 HASH JOIN BUFFERED 这个阻塞操作, 而不是一般的 HASH JOIN.

1. 蓝色的 PX 进程作为生产者, 并行扫描 customer, 通过 table queue 0, hash 分发给作为消费者的红色 PX 进程(第 14~16 行). 每个红色的 PX 进程接收了 1/4 的 customer 的数据(第 13 行), 大约为 370k 行记录, 并准备好第 12 行'HASH JOIN BUFFERED'的 build table. 与 broadcast 分发区别的是, 此时执行计划是从第 16 行, 扫描靠近 lineorder 的 customer 开始的, 而不是从第一个没有'孩子'的操作(第 9 行扫描 part)开始的. 这是 hash 分发和串行执行计划以及 broadcast 分发不同的地方.
2. 蓝色的 PX 进程作为生产者, 并行扫描 lineorder, 通过 table queue 1, hash 分发作为消费者的红色 PX 进程(第 18~20 行). 每个红色 PX 进程接收了 1/4 的 lineorder 数据(第 17 行), 大约 75M 行记录. 每个红色 PX 进程在接收通过 table queue 1 接收数据的同时, 进行第 12 行的 hash join, 并把 join 的结果集在 PGA 中作缓存, 使数据暂时不要继续往上流动. 如果结果集过大的话, 需要把数据暂存到临时空间, 比如我们这个例子, 用了 7GB 的临时空间. 你可以理解为把 join 的结果集暂存到一个临时表. 那么, 为什么执行计划需要在这里插入一个阻塞点, 阻止数据继续往上流动呢?

   这里涉及生产者消费者模型的核心: 同一棵 DFO 树中, 最多只能有两组 PX 进程, 一个数据分发要求两组 PX 进程协同工作; 这意味着同一时刻, 两组 PX 进程之间, 最多只能存在一个活跃的数据分发, 一组作为生产者发送数据, 一组作为消费者接收数据, 每个 PX 进程只能扮演其中一种角色, 不能同时扮演两种角色. 当红色的 PX 进程通过 table queue 1 向蓝色的 PX 进程分发 lineorder 数据, 同时, 蓝色的 PX 进程正在接收 lineorder 数据, 并进行 hash join. 观察 timeline 列的时间轴信息, 第 12, 17~20 行是同时进行的. 但是此时红色的 PX 进程不能反过来作为生产者, 把 hash join 的结果分发给蓝色进程, 因为此时有两个限制:
   - 蓝色的 PX 进程作为生产者, 正忙着扫描 lineorder; 此时, 无法反过来作为消费者, 接收来自红色 PX 进程的数据.
   - 第 5 行 hash jon 操作的 build table 还没准备好, 这时表 part 甚至还没被扫描.

   所以 Oracle 需要在第 12 行 hash join 这个位置插入一个阻塞点, 变成 HASH JOIN BUFFER 操作, 把 join 的结果集缓存起来. 当蓝色的 PX 进程完成对 lineorder 的扫描和分发, 红色的 PX 进程完成第 12 行的 hash join 并把结果完全暂存到临时空间之后. Table queue 2 的数据分发就开始了.
3. 红色的 PX 进程作为生产者, 并行扫描 part, 通过 table queue 2, 分发给作为消费者的蓝色 PX 进程(第 7~9 行). 每个蓝色 PX 进程接收了 1/4 的 part 数据(第 6 行), 大概 300k 行记录, 并准备好第 5 行 hash join 的 build table.
4. 红色的 PX 进程作为生产者, 把在第 12 行"HASH JOIN BUFFERED"操作, 存在临时空间的对于 customer 和 lineorder 连接的结果集, 读出来, 通过 table queue 3, 分发给蓝色的 PX 进程(第 11~12 行). "HASH JOIN BUFFERED"这个操作使用了 7GB 的临时空间, 写 IO 7GB, 读 IO 7GB, IO 总量为 14GB.

5. 每个蓝色的 PX 进程作为消费者, 接收了大约 75M 行记录. 对于通过 table queue 3 接收到的数据, 同时进行第 5 行的 hash join, 并且通过 join 操作的数据进行第 4 行的聚合操作. 当 table queue 3 上的数据分发结束, 每个蓝色的 PX 进程完成 hash join 和聚合操作之后, 再把各自的聚合结果, 一行记录, 通过 table queue 4, 分发给 QC(第 3~5 行). QC 完成最后的聚合, 返回给客户端.



小结

因为使用星型模型测试, 这个例子 使用 Broadcast 分发或者 replicate 才是合理的. 实际应用中, 连续的 hash 分发并不一定会出现 HASH JOIN BUFFERED 这个阻塞点, 如果查询涉及的表都较小, 一般不会出现 HASH JON BUFFERED. 即使执行计划中出现 BUFFER SORT, HASH JOIN BUFFERED 等阻塞操作, 也不意味着执行计划不是最优的. 如果 sql 性能不理想, HASH JOIN BUFFERED 操作消耗了大部分的 CPU 和大量临时空间, 通过 sql monitor 报告, 你可以判断这是否是合理的:

1. 检查 estimated rows 和 actual rows 这两列, 确定优化器对 hash Join 左右两边 cardinality 估算是否出现偏差, 所以选择 hash 分发.
2. 同样检查 hash join 操作的 estimated rows 和 actual rows 这两列, 优化器对 hash join 结果集 cardinality 的估算是否合理. 优化器会把 hash join 的两边视为独立事件, 对 join 结果集 cardinality 的估算可能过于保守, estimate rows 偏小. 对于星型模型的一种典型情况: 如果多个维度表参与连接, 执行路径很长, 一开始维度表的分发方式为 broadcast, 事实表不用分发, 经过几次 join 之后, 结果集 cardinality 下降很快, 后续 hash join 两边的 estimated rows 接近, 导致优化器选择 hash 分发.
3. 通过检查每个 join 所过滤的数据比例, 确定优化器是否把最有效过滤数据的 join 最先执行, 保证在执行路径上流动的数据量最少.

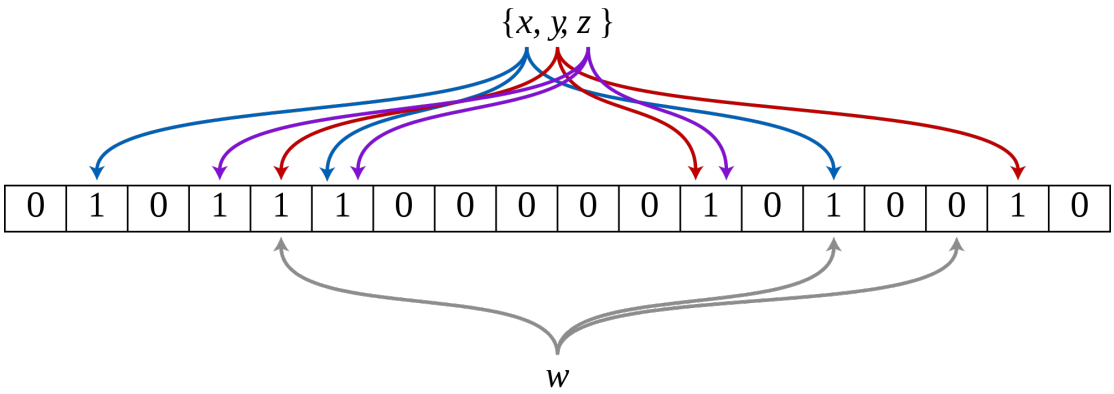
Hash join 和布隆过滤

布隆过滤在并行执行计划中的使用非常普遍, 我将在本章节解释这一数据结构及其作用. 从 11.2 版本开始, 串行执行的 sql 也可以使用布隆过滤.

关于布隆过滤.

布隆过滤是一种内存数据结构, 用于判断某个元素是否属于一个集合. 布隆过滤的工作原理图[2]如下:

[2] 引用自维基百科: http://en.wikipedia.org/wiki/Bloom_filter



如图, 布隆过滤是一个简单的 bit 数组, 需要定义两个变量:

1. m: 数组的大小, 这个例子中, m=18.
2. k: hash 函数的个数, 这个例子中, k=3,

一个空的布隆过滤所有 bit 都为 0. 增加一个元素时, 该元素需要经过三个 hash 函数计算, 得到 3 个 hash 值, 把数组中这三个位置都置为 1. 集合{x,y,z}的 3 个元素, 分布通过三次 hash 计算, 把数组 9 个位置设置为 1. 判断某个元素是否属于一个集合, 比如图中的 w, 只需对 w 进行三次 hash 计算产生三个值, 右边的位置在数组中不命中, 该位置为 0, 可以确定, w 不在{x,y,z}这个集合.
由于存在 hash 碰撞, 布隆过滤的判断会过于乐观(false positive), 可能存在元素不属于{x,y,z}, 但是通过 hash 计算之后三个位置都命中, 被错误认定为属于{x,y,z}. 根据集合元素的个数, 合理的设置数组大小 m, 可以把错误判断的几率控制在很小的范围之内.

布隆过滤对 hash join 性能的改进

布隆过滤的优势在于使用的很少内存, 就可以过滤大部分的数据. 如果 hash join 的左边包含过滤条件, 优化器可能选择对 hash join 左边的数据集生成布隆过滤, 在扫描 hash join 右边时使用这个布隆布隆作为过滤条件, 第一时间把绝大部分不满足 join 条件数据排除. 减少数据分发和 join 操作所处理的数据量, 提高性能.

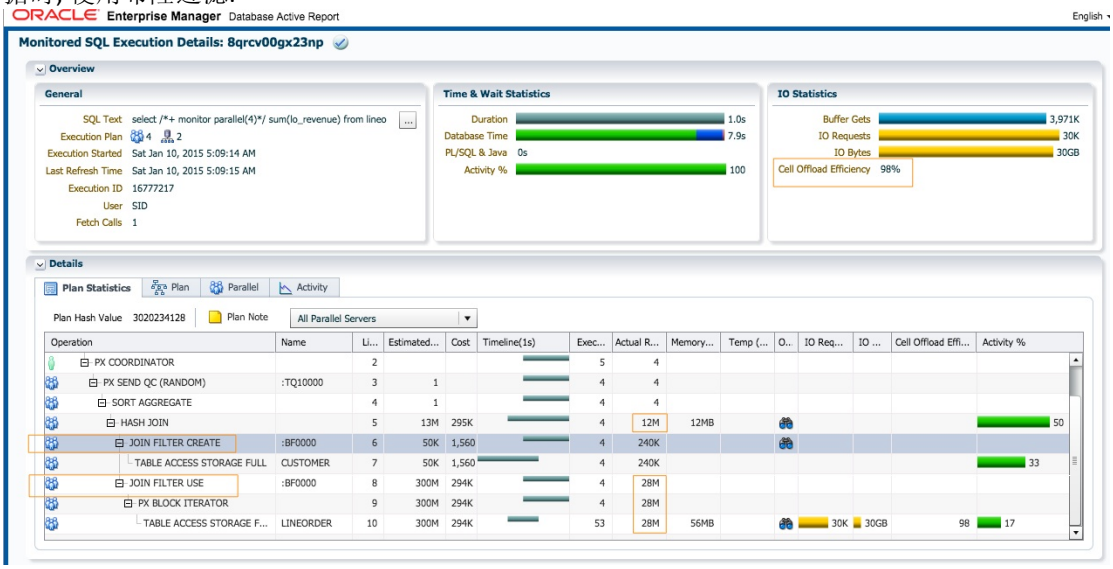
使用布隆过滤时的性能

对 customer 使用 c_nation='CHINA'条件, 只计算来自中国地区的客户订单的利润总和. 我们观察使用布隆过滤和不使用布隆过滤时性能的差别.

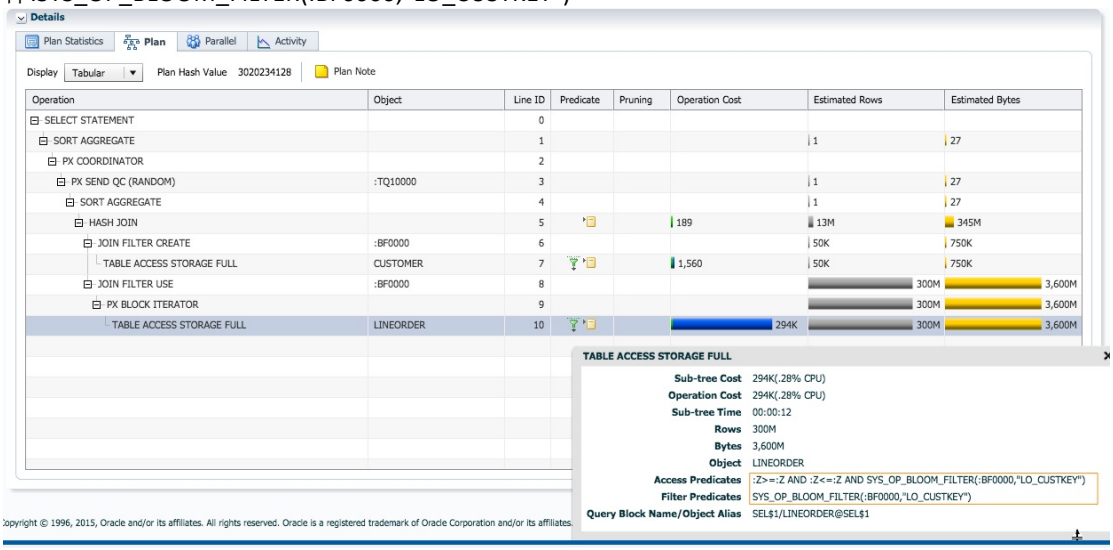
```
select /*+ monitor parallel(4)*/
    sum(lo_revenue)
from
    lineorder, customer
where
    lo_custkey   = c_custkey
and c_nation     = 'CHINA';
```

SQL 执行时间为 1 秒, db time 为 7.9 秒. 优化器默认选择 replicate 的方式. 执行计划中多了 JOIN FILTER CREATE 和 JOIN FILTER USE 这两个操作. SQL 的执行顺序为每个 PX 进程重复扫描 customer 表(第 7 行), 对符合 c_nation='CHINA'数据集, 60K(240K/4)行记录, 在 c_custkey 列生成布隆过滤:BF0000(第 6 行 JOIN FILTER CREATE). 在扫描 lineorder 时使用这个布隆过滤(第 8 行 JOIN FILTER USE). 虽然 lineorder 总行数为 300M, sql 没有过滤条件, 只使用布隆过滤, 扫描之后只返回 28M 行记录, 其他 272M 行记录被过滤掉了. 每个 PX 进程在 hash join 操作时, 只需处理 60K 行 customer 记录和 7M(28M/4)行 lineorder 记录的连接, 大大降低 join 操作的成本. 对于 Exadata, Smart Scan 支持布隆过滤卸载到存储节点, 存储节点扫描 lineorder 时, 使用布隆过滤排除 272M 行记录, 对于符合条件的数据, 把不需要的列也去掉. Cell offload Efficiency=98%, 意味着只有 30GB 的 2%从存储节点返回给 PX 进程. 如果不使用布隆过滤, Cell Offload Efficieny 不会高达 98%, 我们将在下个例子看到. 对于非 Exadata 平台, 由于没有 Smart Scan 特性, 数据的过滤操作需要由 PX 进程完成,

布隆过滤的效果不会这么明显. 12C 的新特性 Database In-memory, 支持扫描列式存储的内存数据时, 使用布隆过滤.



执行计划中出现第 10 行对 LINEORDER 的扫描时, 使用了布隆过滤条件:SYS_OP_BLOOM_FILTER(:BF0000,"LO_CUSTKEY")



不使用布隆过滤时的性能

接着, 我们通过 hint NO_PX_JOIN_FILTER, 禁用布隆过滤, 观察此时的 sql 执行性能.

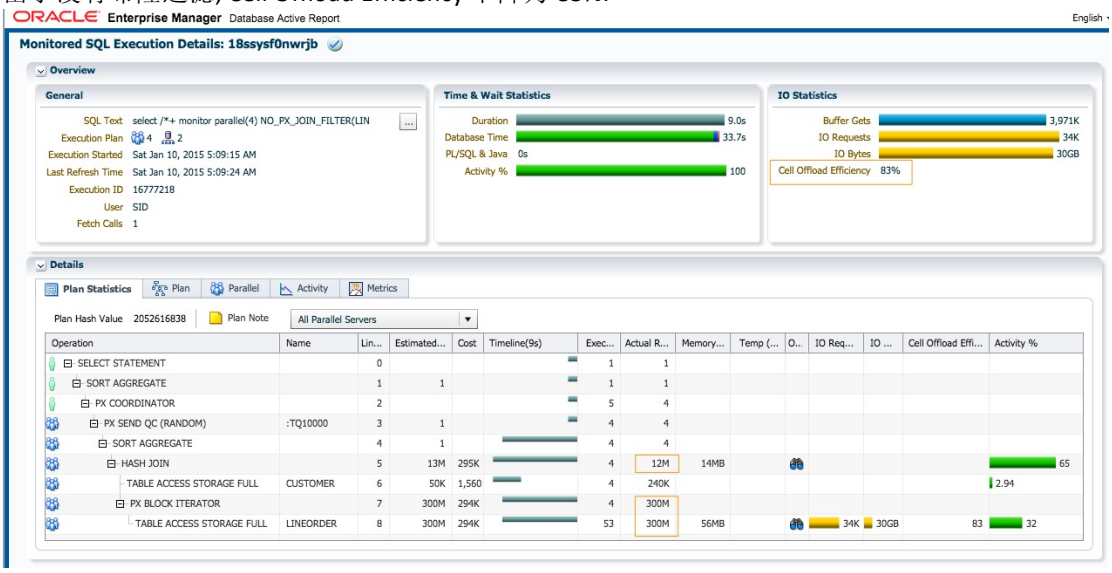
```
select /*+ monitor parallel(4)
        NO_PX_JOIN_FILTER(LINEORDER)*/
    sum(lo_revenue)
from
    lineorder, customer
where
    lo_custkey   = c_custkey
and c_nation     = 'CHINA';
```

SQL 执行时间为 9 秒, db time 为 33.7 秒. 比使用布隆过滤时, 性能下降明显. 优化器依然选择 replicate 的方式, 执行计划中没有 PX JOIN CREATE 和 PX JOIN USE 操作. db time 增加为原来 4 倍的原因:

1. 当 PX 扫描 lineorder 时, 返回 300M 行记录. 没有布隆过滤作为条件, 每个 PX 进程需要从存储节点接收 75M 行记录.

2. 进行第 5 行的 hash join 操作时, 每个 PX 进程需要连接 60k 行 customer 记录和 75M 行 lineorder 记录. Join 操作的成本大幅增加.

由于没有布隆过滤, Cell Offload Efficiency 下降为 83%.



HASH 分发时布隆过滤的生成, 传输, 合并与使用

我们通过 hint 强制使用 hash 分发, 观察此时 sql 执行计划中布隆过滤的生成和使用.

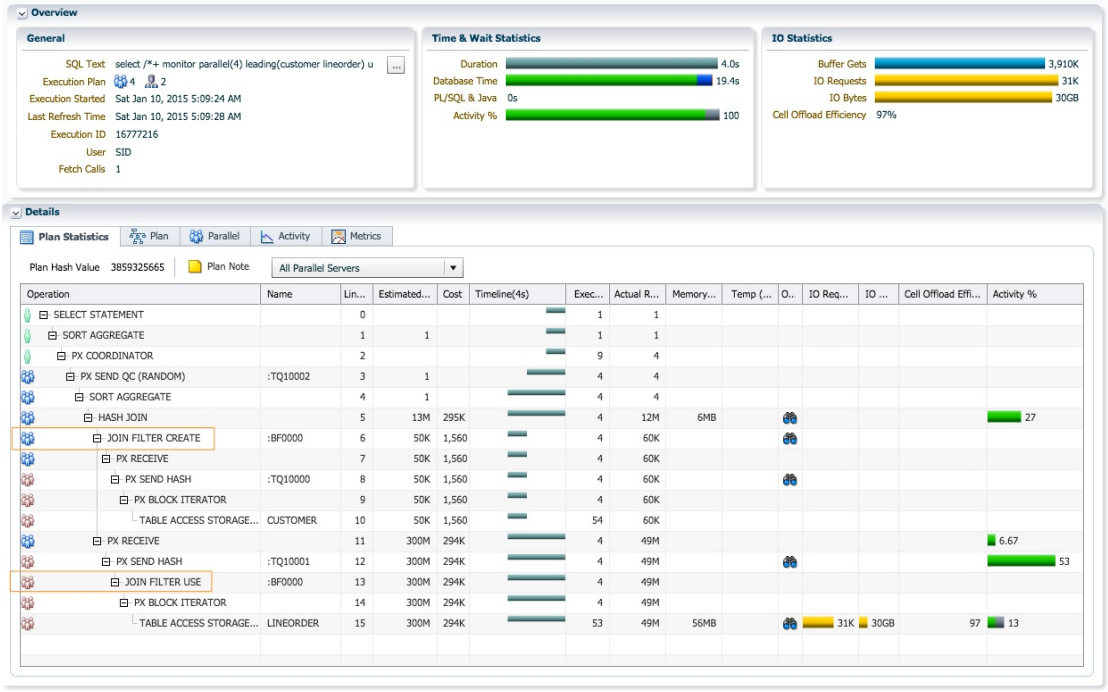
```
select /*+ monitor parallel(4)
        leading(customer lineorder)
        use_hash(lineorder)
        pq_distribute(lineorder hash hash) */
    sum(lo_revenue)
from
    lineorder, customer
where
    lo_custkey = c_custkey
and c_nation  = 'CHINA';
```

此时 sql 执行时间为 4 秒, db time 为 19.4 秒. 执行计划第 6 行为 JOIN FILTER CREATE; 第 13 行为 JOIN FILTER USE. 此例, PX 进程分布在多个 RAC 两个实例, Hash 分发时涉及布隆过滤的生成,传输, 合并和使用, 较为复杂, 具体过程如下:

1. **布隆过滤的产生:** 4 个蓝色的 PX 进程作为消费者, 通过 table queue 0 , 接收红色的 PX 进程 hash 分发的 customer 数据, 每个蓝色的 PX 进程接收 15K 行记录. 接收 customer 记录的同时, 实例 1 的两个蓝色 PX 进程在 SGA 共同生成一个布隆过滤, 假设为 B1; 实例 2 的两个蓝色 PX 进程在 SGA 共同生成一个布隆过滤, 假设为 B2. 因为位于 SGA 中, 布隆过滤 B1 对于实例 1 的两个红色的 PX 进程是可见的, 同样, B2 对于实例 2 的两个红色 PX 进程也是可见的.
2. **布隆过滤的传输:** 当红色的 PX 进程完成对 hash join 左边 customer 的扫描, 就会触发布隆过滤 B1/B2 的传输. 实例 1 的红色 PX 进程把 B1 发给实例 2 的蓝色 PX 进程; 实例 2 的红色 PX 进程把 B2 发给实例 1 的蓝色 PX 进程.
3. **布隆过滤的合并:** 实例 1 的蓝色 PX 进程合并 B1 和接收到的 B2; 实例 2 的蓝色 PX 进程合并 B2 和接收到的 B1. 合并之后, 实例 1 和 2 产生相同布隆过滤.
4. **布隆过滤的使用:** 实例 1 和 2 的 4 个红色的 PX 进程作为生产者, 并行扫描 lineorder 时使用合并之后的布隆过滤进行过滤. Lineorder 过滤之后为 49M 行记录, 此时的布隆过滤似乎没有 replicate 时的有效. Cell Offloadload Efficiency 为 97%.

如果并行执行只在一个实例, 则红色的 PX 进程不需要对布隆过滤进行传输, 蓝色的 PX 进程也无需对布隆过滤进行合并.

因为 hash join 的成本大大降低了, 对于 lineorder 49M 行记录的 hash 分发, 成为明显的平均, 占 53%的 db time.



小结

本节阐述了布隆过滤的原理, 以及在 Oracle 中的一个典型应用: 对 hash join 性能的提升. 布隆过滤的本质在于把 hash join 的连接操作提前了, 对 hash join 右边扫描时, 就第一时间把不符合 join 条件的大部分数据过滤掉. 大大降低后续数据分发和 hash join 操作的成本.

不同的分布方式, 布隆过滤的生成和使用略有不同:

- 对于 broadcast 分发和 replicate, 每个 PX 进程持有 hash join 左边的完整数据, 对连接键生成一个完整的布隆过滤, 扫描 hash join 右边时使用. 如果 sql 涉及多个维度表, 维度表全部使用 broadcast 分发, 优化器可能对不同的维度表数据生成多个的布隆过滤, 在扫描事实表时同时使用.
- 对于 hash 分发, 作为消费者的 PX 进程接收了 hash join 左边的数据之后, 每个 PX 进程分别对各自的数据集生成布隆过滤, 再广播给作为生产者的每个 PX 进程, 在扫描 hash join 右边时使用.

真实世界中, 优化器会根据统计信息和 sql 的过滤条件自动选择布隆过滤. 通常使用布隆过滤使都会带来性能的提升. 某些极端的情况, 使用布隆过滤反而造成性能下降, 两个场景:

- 当 hash join 左边的数据集过大, 比如几百万行, 而且连接键上的唯一值很多, 优化器依然选择使用布隆过滤. 生成的布隆过滤过大, 无法在 CPU cache 中完整缓存. 那么使用布隆过滤时, 对于 hash join 右边的每一行记录, 都需要到内存读取布隆过滤做判断, 导致性能问题.
- 如果 Join 操作本身无法过滤数据, 使用布隆过滤时 hash join 右边的数据都会命中. 优化器可能无法意识到 join 操作无法过滤数据, 依然选择使用布隆布隆. 如果 hash join 右边数据集很大, 布隆过滤可能会消耗明显的额外 cpu.

并行执行计划中典型的串行点

现实世界中, 由于使用不当, 并行操作无法并行, 或者并行执行计划效率低下, 没有获得期望的性能提升. 本节举几个典型例子.

1. 在 sql 中使用 rownum, 导致出现 PX SEND 1 SLAVE 操作, 所有数据都需要分发到一个 PX 进程, 以给每一行记录赋值一个唯一的 rownum 值, 以及 BUFFER SORT 等阻塞操作.
2. 使用用户自定义的 pl/sql 函数, 函数没有声明为 parallel_enable, 导致使用这个函数的 sql 无法并行.
3. 并行 DML 时, 没有 enable parallel dml, 导致 DML 操作无法并行.

Rownum, 导致并行执行计划效率低下

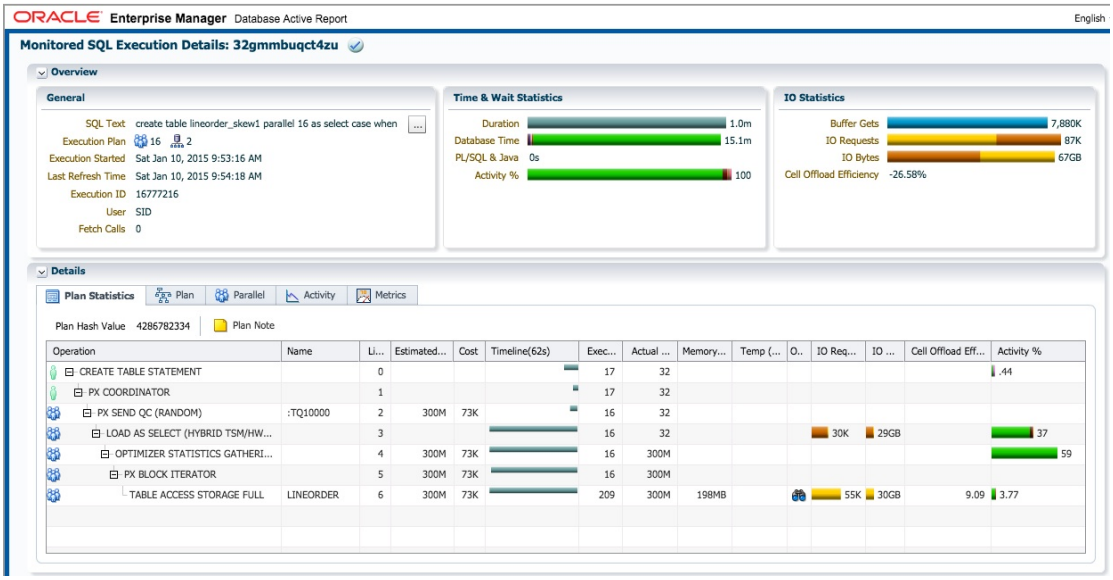
在'数据倾斜对不同分发方式的影响'小节中, 我们新建一个表 lineorder_skew 把 lineorder 的 lo_custkey 列 90%的值修改为-1. 因为 lo_custkey 是均匀分布的, 我们可以通过对 lo_custkey 列求模, 也可以通过对 rownum 求模, 把 90%的数据修改为-1. 使用如下的 case when 语句:

```
1. case when mod(lo_orderkey, 10) > 0 then -1 else lo_orderkey end
lo_orderkey
2. case when mod(rownum, 10) > 0 then -1 else lo_orderkey end lo_orderkey
```

通过以下的建表 sql 来测试两种用法时的 sql 执行性能, 并行度为 16.

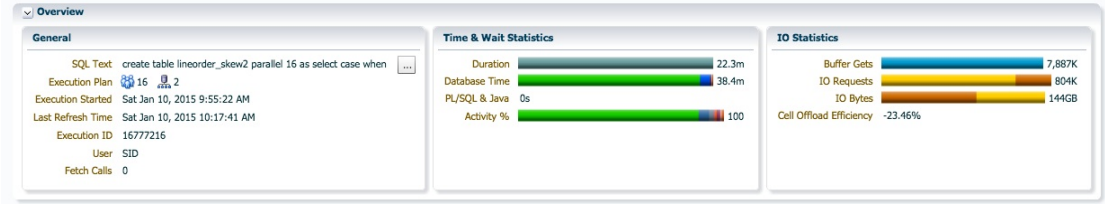
```
create table lineorder_skew1 parallel 16
as select
    case when mod(lo_orderkey, 10) > 0 then -1 else lo_orderkey end
lo_orderkey,
    lo_linenumber,
    lo_custkey,
    lo_partkey,
    lo_suppkey,
    lo_orderdate,
    lo_orderpriority,
    lo_shippriority,
    lo_quantity,
    lo_extendedprice,
    lo_ordtotalprice,
    lo_discount,
    lo_revenue,
    lo_supplycost,
    lo_tax,
    lo_commitdate,
    lo_shipmode,
    lo_status
from lineorder;
```

不使用 rownum 时, create table 执行时间为 1 分钟, db time 为 15.1 分钟. QC 只分配了一组 PX 进程, 每个蓝色的 PX 进程以基于数据块地址区间为单位, 并行扫描 lineorder 表, 收集统计信息, 并加载到 lineorder_skew1 表. 没有数据需要分发, 每个 PX 进程一直保持活跃, 这是最有效率的执行路径.



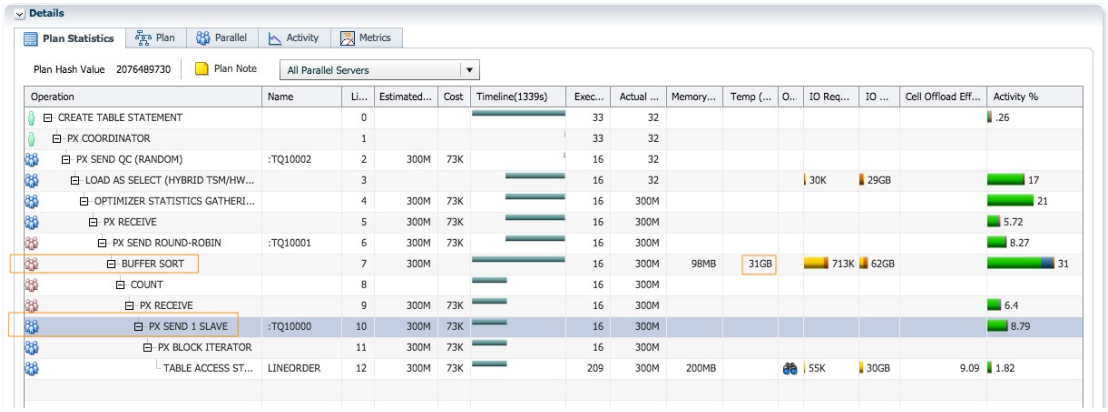
大部分时间, AAS=16.

使用 rownum 时, create table 执行时间为 22.3 分钟, db time 为 38.4 分钟. SQL 的执行时间为使用 lo_orderkey 时的 22 倍.



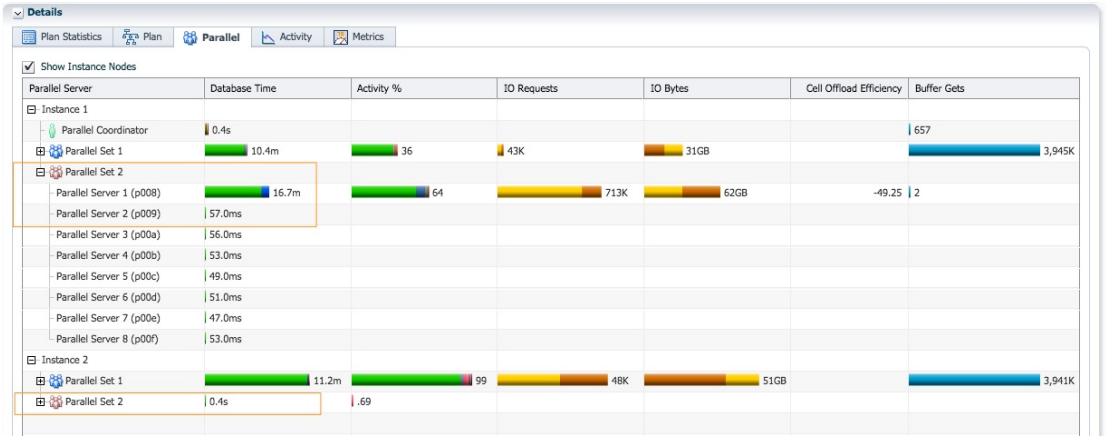
执行计划中出现两组 PX 进程, PX SEND 1 SLAVE 和 BUFFER SORT 两个操作在之前的测试没有出现过. 根据**跟随 table queue 顺序**的原则, 我们来阅读这个执行计划:

1. 蓝色的 PX 进程并行扫描 lineorder, 通过 table queue 0 把所有数据分发给一个红色的 PX 进程(第 10~12 行). **因为 rownum 是一个伪列, 为了保证每一行记录拥有一个唯一行号, 对所有数据的 rownum 赋值这个操作只能由一个进程完成, 为 rownum 列赋值成为整个并行执行计划的串行点. 这就是出现 PX SEND 1 SLAVE 操作, 性能急剧下降的原因.** 这个例子中, 唯一活跃的红色 PX 进程为实例 1 p008 进程. Lineorder 的 300M 行记录都需要发送到实例 1 p008 进程进行 rownum 赋值操作, 再由这个进程分发给 16 个蓝色的 PX 进程进行数据并行插入操作.
2. 实例 1 p008 进程接收了 16 个蓝色 PX 进程分发的数据, 给 rownum 列赋值(第 8 行 count 操作)之后, 需要通过 table queue 1 把数据分发给蓝色的 PX 进程. 但是因为通过 table queue 0 的数据分发的还在进行, 所以执行计划插入一个阻塞点 BUFFER SORT(第 7 行), 把 rownum 赋值之后的数据缓存到临时空间, 大小为 31GB.
3. Table queue 0 的数据分发结束之后, 实例 1 p008 把 31GB 数据从临时空间读出, 通过 table queue 1 分发给 16 个蓝色的 PX 进程进行统计信息收集和插入操作.

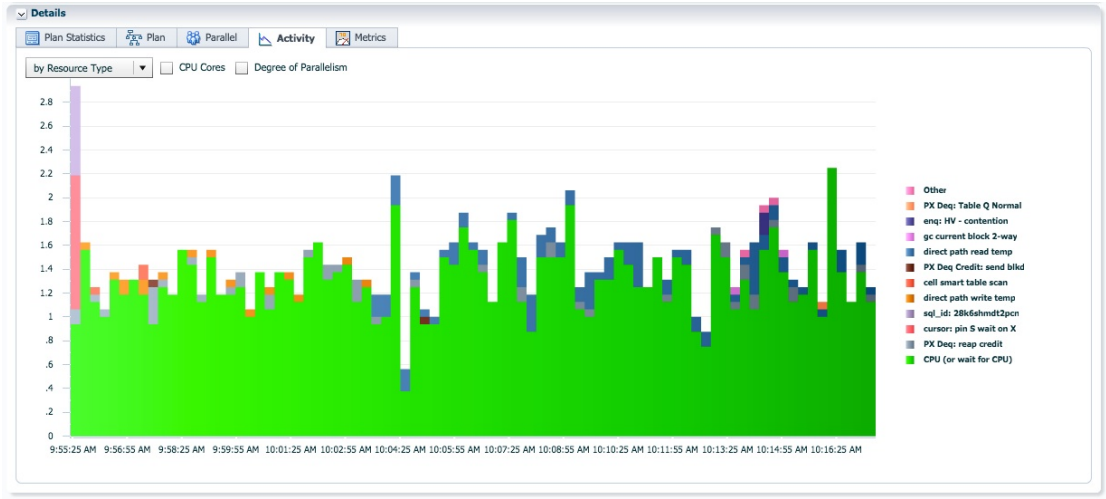


红色的 PX 进程只有实例 1 p008 是活跃的. 消耗了 16.7 分钟的 db time. 对于整个执行计划而言, 两次数据分发也消耗了大量的 db cpu. 通过 Table queue 0 把 300M 行记录从 16 个蓝色的 PX 进程分发给 1 个红色的 PX 进程. 通过 Table queue 1 把 300M 行记录从 1 个红色的 PX 进程分发给

16 个蓝色的 PX 进程.



虽然 DoP=16, 实际 AAS=1.5, 意味着执行计划效率低下.



现实世界中, 在应用中应该避免使用 rownum. Rownum 的生成操作会执行计划的串行点, 增加无谓的数据分发. 对于使用 rownum 的 sql, 提升并行度往往不会改善性能, 除了修改 sql 代码, 没有其他方法.

自定义 PL/SQL 函数没有设置 parallel_enable, 导致无法并行

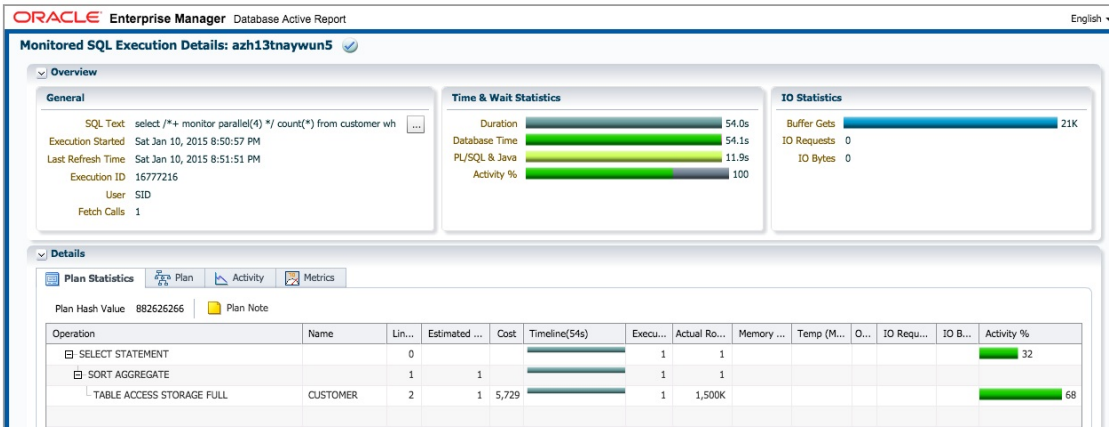
Rownum 会导致并行执行计划出现串行点, 而用户自定义的 pl/sql 函数, 如果没有声明为 parallel_enable, 会导致 sql 只能串行执行, 即使用 hint parallel 指定 sql 并行执行. 我们来测试一下, 创建 package pk_test, 包含函数 f, 返回和输入参数一样的值. 函数的声明中没有 parallel_enable, 不支持并行执行.

```
create or replace package pk_test authid current_user
as
  function f(p_n number) return number;
end;
/

create or replace package body pk_test
as
    function f(p_n number) return number
    as
    l_n1 number;
    begin
      select 0 into l_n1 from dual;
      return p_n - l_n1;
    end;
end;
/
```

以下例子中在 where 语句中使用函数 pk_test.f, 如果在 select 列表中使用函数 pk_test.f, 也会导致执行计划变成串行执行.

```
select /*+ monitor parallel(4) */
    count(*)
from
    customer
where
    c_custkey = pk_test.f(c_custkey);
```

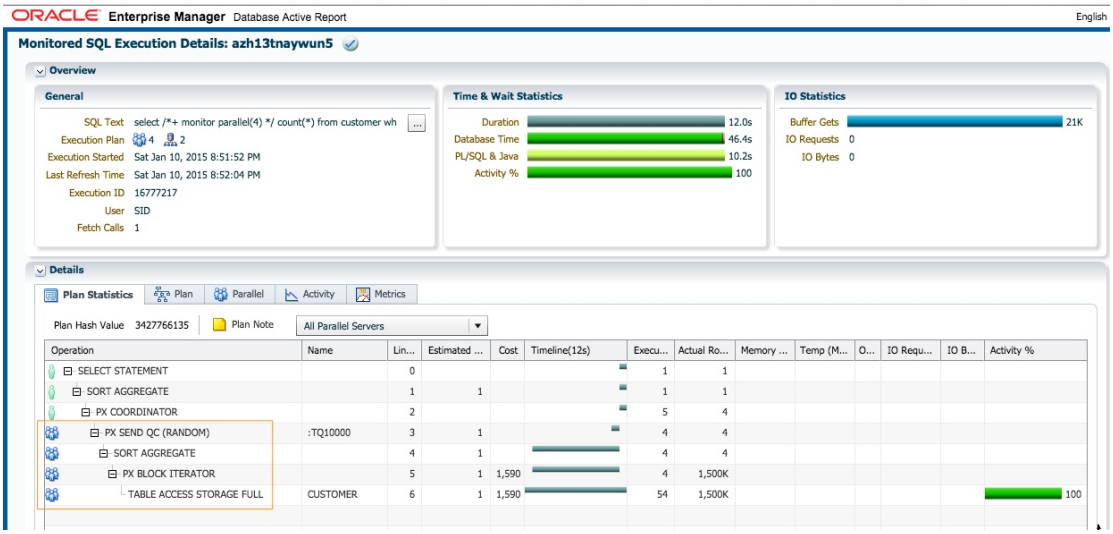
查询执行时间为 54 秒, db time 也为 54 秒. 虽然我们指定使用 Dop=4 并行执行, 执行计划实际是串行的.



在函数的声明时设置 parallel_enable, 表明函数支持并行执行, 再次执行 sql.

```
create or replace package pk_test authid current_user
as
  function f(p_n number) return number parallel_enable;
end;
/

create or replace package body pk_test
as
    function f(p_n number) return number parallel_enable
    as
    l_n1 number;
    begin
      select 0 into l_n1 from dual;
      return p_n - l_n1;
    end;
end;
/
```

此时查询的执行时间为 12 秒, db time 为 46.4 秒. 并行执行如期发生, 并行度为 4.



除非有特殊的约束, 创建自定义 pl/sql 函数时, 都应该声明为 parallel_enable. pl/sql 函数声明时没有设置 parallel_enable 导致无法并行是一个常见的问题, 我曾在多个客户的系统中遇到. 在 11g 中, 这种情况发生时, 执行计划中可能会出现 PX COORDINATOR FORCED SERIAL 操作, 这是一个明显的提示; 或者你需要通过 sql monitor 报告定位这种问题. 仅仅通过 dbms_xplan.display_cursor 检查执行计划是不够的, 这种情况执行计划的 note 部分, 还是会显示 DoP=4.

并行 DML, 没有 enable parallel dml, 导致 DML 操作无法并行.

这是 ETL 应用中常见的问题, 没有在 session 级别 enable 或者 force parallel dml, 导致 dml 操作无法并行. 使用 customer 的 1.5M 行数据演示一下.

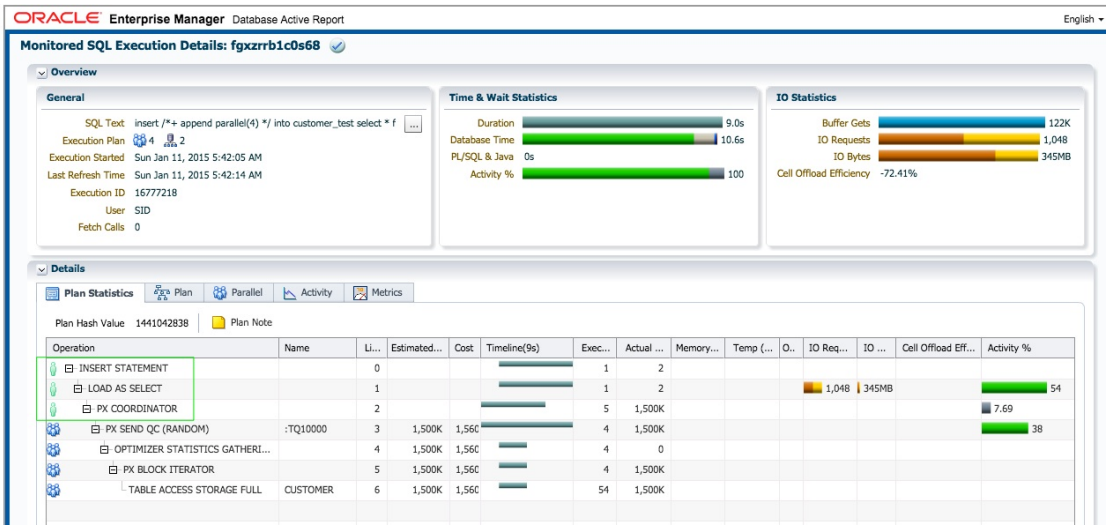
建一个空表 customer_test:

```
create table customer_test as select * from customer where 1=0;
```

我们使用并行直接路径插入的语句作为例子. 分别执行两次 insert, 第一次没有 enable parallel dml, insert 语句如下:

```
insert /*+ append parallel(4) */ into customer_test select * from customer;
```

Insert 语句执行时间 9 秒. 虽然整个语句的并行度为 4, 但是执行计划中, 第 2 行直接路径插入操作 LOAD AS SELECT 是串行执行的.



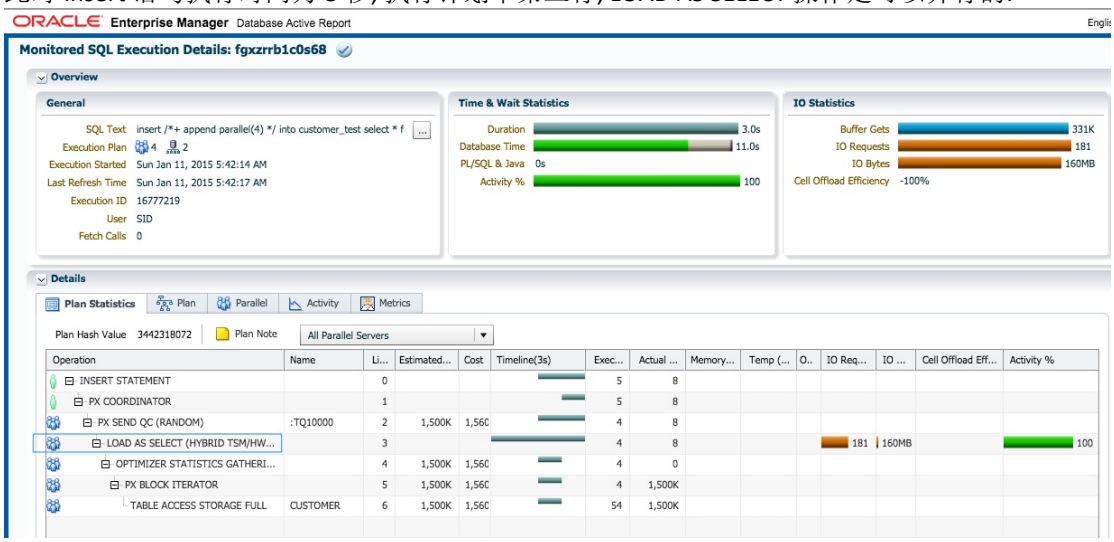
此时执行计划的 Note 部分会显示 PDML 没有启用:

```
Note
-----
   - PDML is disabled in current session
```

启用 parallel dml 之后, 重新执行 insert 语句.

```
alter session enable parallel dml;
```

此时 insert 语句执行时间为 3 秒, 执行计划中第三行, LOAD AS SELECT 操作是可以并行的.



小节

我列举了使用并行执行时, 常见的三种问题:

1. 使用 rownum.
2. 自定义 pl/sql 函数没有声明 parallel_enable.
3. 并行 DML 时没有 enable parallel dml.

希望通过以上三个例子, 希望读者对调试并行执行计划有一个更直观的感受. 处理并行执行的问题, sql monitor 报告是最好的分析工具. 对于并行 DDL 和 DML, Oracle 本身有一些限制, 可以参见官方文档[3], 比如:

1. 表上的触发器或者外键约束可能导致 DML 无法并行.
2. 包含 LOB 字段的非分区表, 不支持并行 DML 和 DDL; 包含 LOB 字段的分区表, 只支持分区间的并行 DML 和 DDL.
3. 远程表(通过 db link) 不支持并行 DML; 临时表不支持并行 update, merge, delete.

总结

这篇长长的文章更像是我在 Real-World Performance Group 的工作总结. 在大量实际项目中, 我们发现很多开发或者 DBA 并没有很好理解并行执行的工作原理, 设计和使用并行执行时, 往往也没取得最佳的性能. 对于并行执行, 已经有很多的 Oracle 书籍和网上文章讨论过, 在我看来, 这些内容更偏重于并行执行原理的解释, 缺乏实际的使用案例. 我希望在本文通过真实的例子和数据, 以最简单直接的方式, 向读者阐述 Oracle 并行执行的核心内容, 以及在现实世界中, 如果规避最常见的使用误区.也希望本文所使用 sql monitor 报告分析性能问题的方法, 对读者有所启示! 如果现在你对以下并行执行的关键点, 都胸有成竹的话, 我相信现实世界中 Oracle 的并行执行问题都不能难倒你.

- Oracle 并行执行为什么使用生产者-消费者模型.
- 如何阅读并行执行计划.
- 不同的数据分发方式分别适合什么样的场景.
- 使用 partition wise join 和并行执行的组合提高性能.

---

[3] Restrictions on Parallel DML
http://docs.oracle.com/cd/E11882_01/server.112/e25523/parallel003.htm#CACEJACE

- 数据倾斜会对不同的分发方式带来什么影响.
- 由于生产者-消费者模型的限制, 执行计划中可能出现阻塞点.
- 布隆过滤是如何提高并行执行性能的.
- 现实世界中, 使用并行执行时最常见的问题.

下一篇文章, 我将介绍 12c 的新特性, adaptive 分发.

致谢

本文目前的内容和质量, 源于对初稿的多次审校和迭代. 本文的两个难点, 1) 连续 hash 分发时出现阻塞点; 2) hash 分发时使用布隆过滤的具体过程, 得到了我英国同事 Mike Hallas 的解答和确认. 我的同事董志平对初稿的做了详细的审校, 指出多处纰漏. 本文的一些内容是在他的建议下增加的, 比如 Partition Wise Join 时, DoP 大于分区数时 partition wise join 会失效, 比如 replicate 方式为什么不能完全替代 broadcast 分发. 我的同事徐江和李常勇, 我的朋友蒋健阅读初稿之后, 也提供了诸多反馈, 在此一并感谢!

作者简介:

陈焕生, 英文名 Sidney
Oracle Real-World Performance Group 成员, senior performance engineer, 专注于 OLTP, OLAP 系统在 Exadata 平台和 In-Memory 特性上的最佳实践.

个人站点: dbsid.com
Email: sidney.chen@oracle.com

经过作者本人授权同意，本文发表在 OTN 网站上。