Oracle 12c In-Memory Option介绍

作者：陈志民
11g OCM，10g OCP , SHOUG，Oracle Young Expert
http://czmmiao.iteye.com
广发基金管理有限公司
数据架构师

CONTENTS

1

PART ONE

In-Memory Option 概述和使用

- 行式存储高效定位特定行，并且可以同时操作多列，针对DML操作为主的OLTP系统
- 列式存储提供高效的特定列聚合和扫描操作，多用于OLAP系统

In-Memory Option 概述和使用

In-Memory Option提供双模式混合数据存储方式：

- 数据文件和buffer cache中以行形式存放数据；
- SGA内的独立空间in memory area中以列示存放数据

IN Memory组件概述

- 提供行存储和列存储共存的混合存储方式
- Oracle 12.1.0.2或以上版本
- inmemory_size默认为0表示不开启，最小值为100MB
- 必须重启数据库才能生效

IN Memory组件概述

存在以下限制

- SYS用户存储在SYSTEM或 SYSAUX表空间的对象
- 索引组织表，簇表
- LONG字段
- Out of line LOBS
- 小于64KB的对象（IMCU的最小单位1MB）
- 无法在Active Data Guard上应用

使用IN Memory组件

```
SQL>  alter system set inmemory_size=200M scope=spfile;

System altered.

SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup;
ORACLE instance started.

Total System Global Area 4865392640 bytes
Fixed Size                  2934264 bytes
Variable Size             939526664 bytes
Database Buffers         3690987520 bytes
Redo Buffers               13840384 bytes
In-Memory Area            218103808 bytes
Database mounted.
Database opened.
```

使用IN Memory组件 ---- 相关参数

- INMEMORY_SIZE：设置IM column store的内存大小
- INMEMORY_FORCE：默认值为DEFAULT，如果设为OFF，即使开启了IN Memory属性，也不会有对象加载到in-memory area区域
- OPTIMIZER_INMEMORY_AWARE：控制CBO是否对IN Memory敏感的参数
- INMEMORY_QUERY：如果设置为DISABLE，查询语句将无法利用IM column store的特性
- INMEMORY_MAX_POPULATE_SERVERS：控制数据加载进程同时工作的最大数量
- INMEMORY_CLAUSE_DEFAULT：设置IN Memory的默认数据加载到内存的级别
- INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT：设置IMCU中’stale’数据所占的最大比例

使用IN Memory组件 ---- hint

- INMEMORY:当INMEMORY_QUERY参数设置为FALSE时，INMEMORY将令CBO考虑IN Memory扫描。
- NO_INMEMORY:当INMEMORY_QUERY参数设置为true时，NO_INMEMORY将禁用IN Memory扫描
- (NO_)INMEMORY_PRUNING：该hint会建议CBO使用或者禁用 In-Memory storage indexes进行数据裁剪，默认都是开启In-Memory storage indexes以减少数据扫描，多用于测试
- PX_JOIN_FILTER：建议CBO使用Bloom Filter。
- VECTOR_TRANSFORM：建议CBO使用VECTOR GROUP BY

2

PART TWO

数据加载和一致性保证

数据加载 ---- IN Memroy加载及压缩粒度

- IN Memroy可以针对列，表，分区或表空间进行压缩
- 不同对象数据加载的优先级别不同
- 存在不同的压缩方式
- 对单个对象中可能存在不同的压缩方式

数据加载 ---- IN Memroy数据加载的优先级别

Priority Level	Description
`CRITICAL`	Object is populated immediately after the database is opened
`HIGH`	Object is populated after all `CRITICAL` objects have been populated, if space remains available in the IM column store
`MEDIUM`	Object is populated after all `CRITICAL` and `HIGH` objects have been populated, and space remains available in the IM column store
`LOW`	Object is populated after all `CRITICAL`, `HIGH`, and `MEDIUM` objects have been populated, if space remains available in the IM column store
`NONE`	Objects only populated after they are scanned for the first time (Default), if space is available in the IM column store

数据加载 ---- IN Memroy压缩方式

COMPRESSION LEVEL	DESCRIPTION
NO MEMCOMPRESS	Data is populated without any compression
MEMCOMPRESS FOR DML	Minimal compression optimized for DML performance
MEMCOMPRESS FOR QUERY LOW	Optimized for query performance (default)
MEMCOMPRESS FOR QUERY HIGH	Optimized for query performance as well as space saving
MEMCOMPRESS FOR CAPACITY LOW	Balanced with a greater bias towards space saving
MEMCOMPRESS FOR CAPACITY HIGH	Optimized for space saving

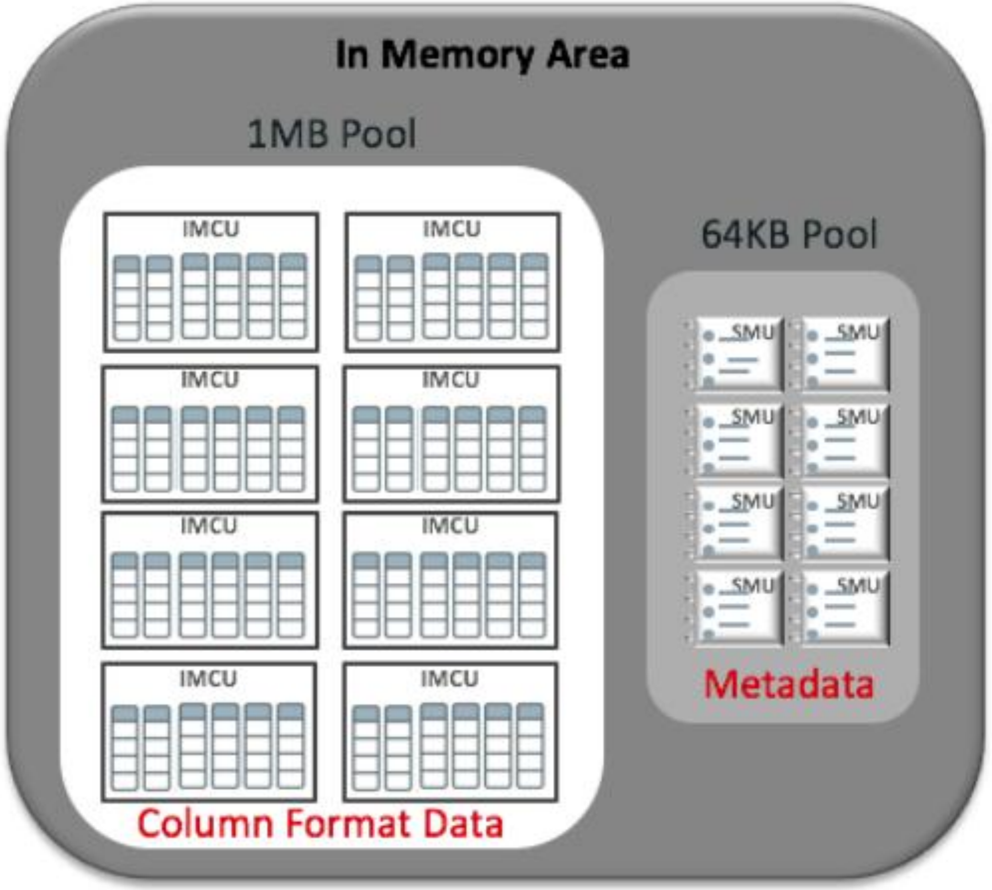
数据加载 ---- 应用IN Memroy

```
-- 对表使用IN Memory
alter table sales inmemory no memcompress priority critical;

-- 对表空间使用IN Memory
alter tablespace ts_data inmemory;

-- 对分区使用IN Memory
alter table sales modify partition sales_201501 inmemory;

-- 对列使用IN Memory策略
alter table sh.sales inmemory no inmemory(prod_id);
```

数据加载 ---- 应用IN Memroy

```
-- 创建压缩级别不同的IN Memory

CREATE TABLE t1
(
   c1 NUMBER,
   c2 NUMBER,
   c3 VARCHAR2(10),
   c4 CLOB
 )
INMEMORY MEMCOMPRESS FOR QUERY
NO INMEMORY(c4)
INMEMORY MEMCOMPRESS FOR CAPACITY HIGH(c2);
```

列式存储的内存结构



In-Memory Area是SGA的新组件，由1M pool和64K pool两部分构成

- 1M pool用于保存列格式数据, IMCU(in memory Compression Unit)为存储的基本单位
- 64K pool用于保存和IMCU相对的元数据信息，SMU(Snapshot Metadata Unit)是这部分内存的基本单位

列式存储的内存结构 ----- 加载IMCU

- 加载过程是通过工作进程（W00）进程实现
- 每个工作进程以IMCU为单位并行的进行数据加载工作
- 数据加载采用流机制进行加载，数据列式存储和压缩同时进行
- 数据的加载过程是根据行存储进行顺序加载，不会进行额外排序

列式存储的内存结构 ----- IMCU

每个IMCU包含IMCU header， 在IMCU header包含IMCU的元数据信息

```sql
select OBJD, TSN, IMCU_ADDR, IS_HEAD_PIECE, PREPOPULATED,COMPRESSION_LEVEL, NUM_DISK_EXTENTS
, HEADER_SIZE, NUM_COLS, NUM_ROWS, NUM_BLOCKS
from v$im_header;
```

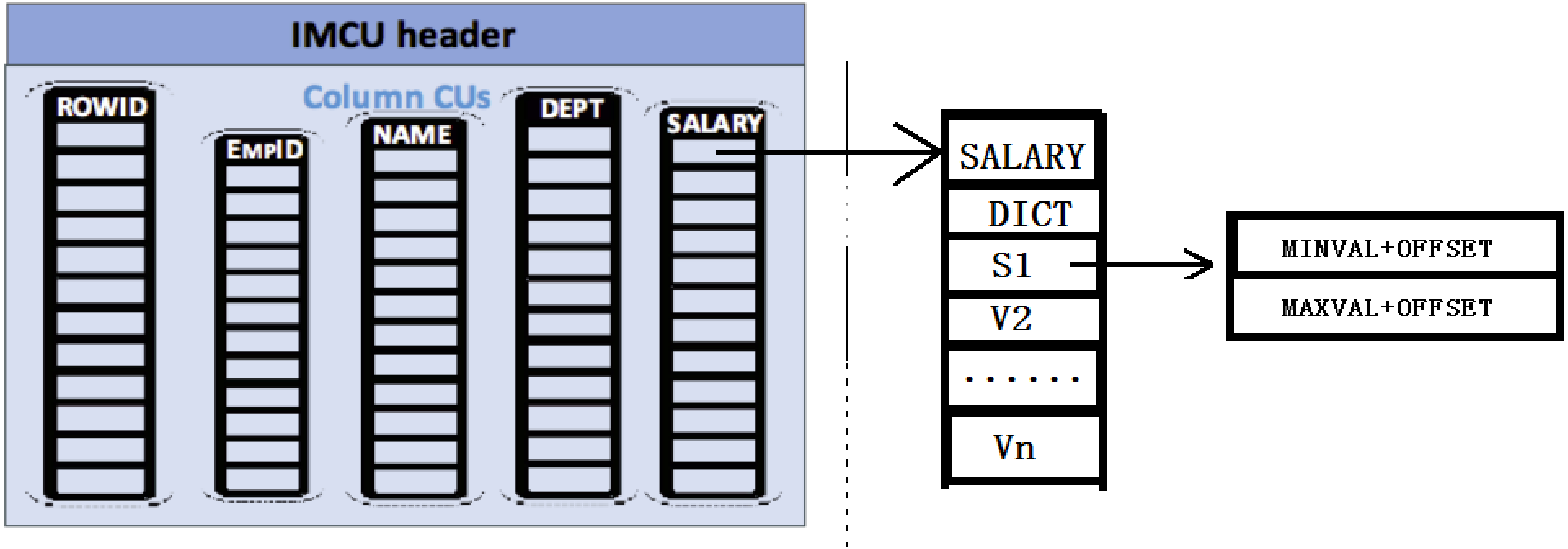
Script Output × | Query Result × | Query Result 1 ×

SQL | All Rows Fetched: 8 in 0.006 seconds

	OBJD	TSN	IMCU_ADDR	IS_HEAD_PIECE	PREPOPULATED	COMPRESSION_LEVEL	NUM_DISK_EXTENTS	HEADER_SIZE	NUM_COLS	NUM_ROWS	NUM_BLOCKS
1	91803	3	0000000065FFFFA0	1	1	5	28	288	18	90941	1527
2	91803	3	0000000069FFFFB8	1	1	5	54	496	18	466769	7802
3	91803	3	00000000663FFFA0	1	1	5	3	96	18	167816	2808
4	91803	3	0000000071FFFFE8	1	1	5	7	128	18	381327	6372
5	91803	3	00000000726FFFE8	1	1	5	10	144	18	549089	9180
6	91803	3	0000000062CFFF88	1	1	5	10	144	18	549004	9180
7	91803	3	00000000623FFF88	1	1	5	10	144	18	549135	9180
8	91803	3	0000000061FFFF88	1	1	5	3	96	18	156031	2808

列式存储的内存结构 ----- IMCU

IMCU中自动维护了In-Memory Storage Index，Storage Index中保存有每列的最大，最小值，distinct等元数据信息，这也就是列式存储可以加快扫描的关键所在



列式存储的内存结构 ---- IMCU测试案例

```
CREATE  TABLE t4
  (object_id     NUMBER,
    object_name   VARCHAR2(30),
    LAST_DDL_TIME DATE )
  INMEMORY PRIORITY LOW MEMCOMPRESS FOR QUERY
  NO INMEMORY  ( LAST_DDL_TIME  )
  INMEMORY MEMCOMPRESS FOR CAPACITY HIGH (object_name);

INSERT INTO t4
SELECT  object_id,  object_name,  LAST_DDL_TIME
FROM  dba_objects;
COMMIT;
```

列式存储的内存结构 ---- IMCU测试案例

查询in memory area的使用情况

```
SQL> select POOL, ALLOC_BYTES, USED_BYTES, POPULATE_STATUS, CON_ID from v$inmemory_area;

POOL                       ALLOC_BYTES USED_BYTES POPULATE_STATUS                CON_ID
-------------------------- ----------- ---------- -------------------------- ----------
1MB POOL                     166723584    2097152 DONE                                3
64KB POOL                     33554432     131072 DONE                                3
```

列式存储的内存结构 ---- IMCU测试案例

查询in memory area的中存储的列信息

```
select * from v$im_col_cu;
```

Script Output x | Query Result x

SQL | All Rows Fetched: 2 in 0.016 seconds

	OBJD	TSN	HEAD_PIECE_ADDRESS	COLUM...	LENGTH	ROW_LEN	TRANSFORMED_LEN	COMPRESSION_NUMBER	DICTIONARY_ENTR...	MINIMUM_VALUE	MAXIMUM_VALUE	CON_ID
1	91823	3	0000000062F...	1	909899	0	909883	4352	90945	C103	C30A1318	3
2	91823	3	0000000062F...	2	719886	0	1639554	4354	53589	2F31303...	79436243...	3

列式存储的内存结构 ---- IMCU测试案例

查看in memory area中的段信息

```sql
SELECT
  SEGMENT_NAME ,
  BYTES_NOT_POPULATED,
  POPULATE_STATUS,
  BYTES ORIG_SIZE,
  INMEMORY_SIZE ,
  BYTES/INMEMORY_SIZE COMP_RATIO
FROM
  V$IM_SEGMENTS;
```

Script Output × | Query Result ×

SQL | All Rows Fetched: 1 in 0.003 seconds

	SEGMENT_NAME	BYTES_NOT_POPULATED	POPULATE_STATUS	ORIG_SIZE	INMEMORY_SIZE	COMP_RATIO
1	T4	0	COMPLETED	5242880	2228224	2.35294117

列式存储的内存结构 ---- IMCU测试案例

查询表的in memory属性

```sql
SELECT TABLE_NAME, INMEMORY, INMEMORY_PRIORITY, INMEMORY_DISTRIBUTE, INMEMORY_COMPRESSION
FROM USER_TABLES;
```

Script Output | Query Result

SQL | All Rows Fetched: 5 in 0.114 seconds

	TABLE_NAME	INMEMORY	INMEMORY_PRIORITY	INMEMORY_DISTRIBUTE	INMEMORY_COMPRESSION
1	T2	ENABLED	NONE	AUTO	FOR QUERY LOW
2	T3	ENABLED	NONE	AUTO	FOR QUERY LOW
3	T4	ENABLED	LOW	AUTO	FOR QUERY LOW
4	T1	ENABLED	NONE	AUTO	FOR QUERY LOW
5	TT	DISABLED	(null)	(null)	(null)

3

PART ONE

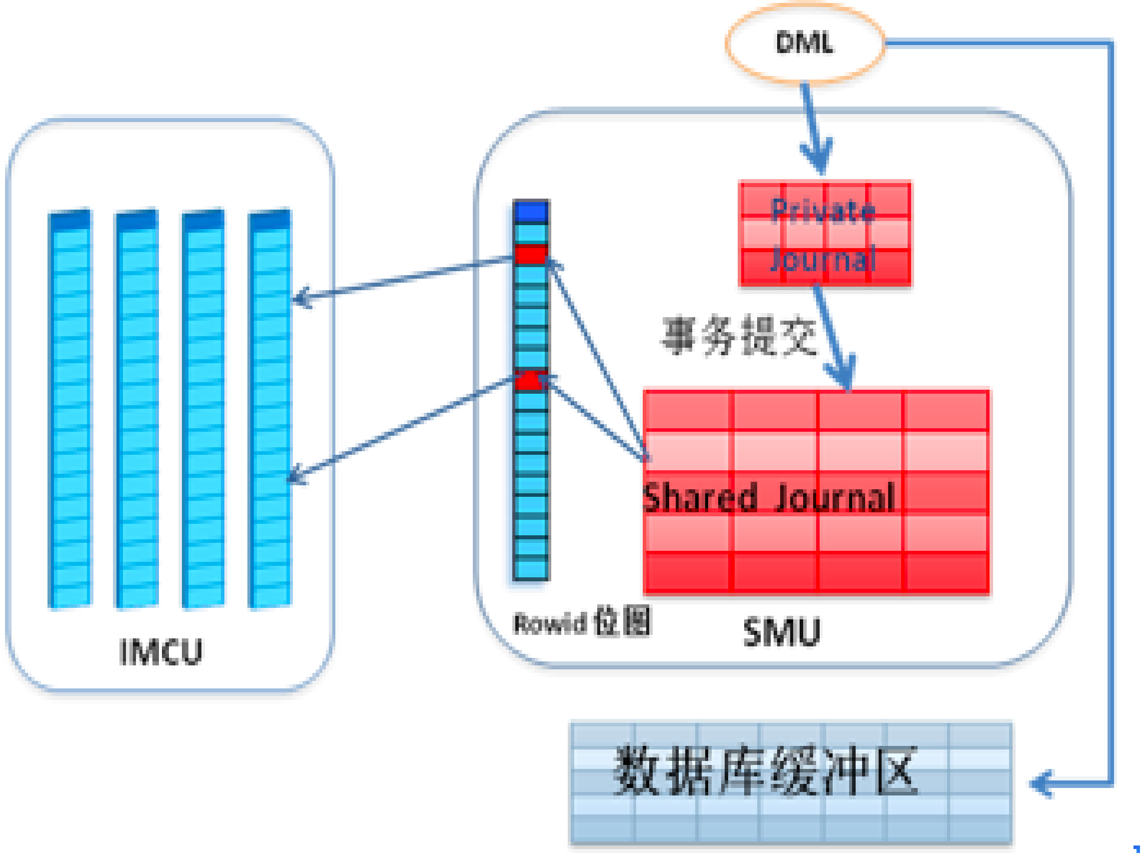
数据一致性的保证

IN Memory的数据一致性 ---- Transaction Journal

- 组件Oracle 通过额外维护一份事务日志(Transaction Journal) 的方式来确保数据的一致性

- Transaction Journal分为private journal和shared journal两种类型

- Transaction table当中对应的记录标识成为shared journal时，则该记录就会被标记为‘stale’



IN Memory的数据一致性对性能的影响

- 随着IMCU中stale数据和 journal table中的过多，将会非常影响in memory组件的查询性能

- INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT：定义了IMCU中‘stale’数据所占的最大比例

- Oracle新增 IMCO (In-Memory Coordinator) 后台进程会来对拥有过多stale条目的IMCU数据块进行检查。IMCO默认2分钟被唤醒一次，该进程会调用W00工作进程进行数据重载

- 连续的数据块和MEMORYCOMPRESS FOR DML这些轻量级的压缩方式会降低重载带来的性能影响

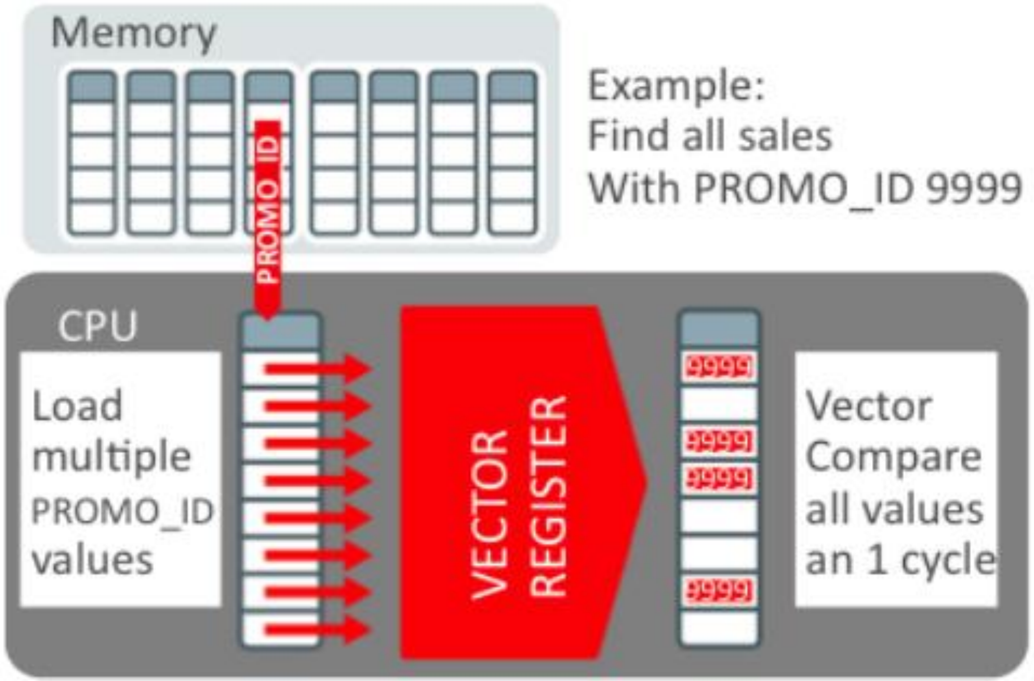
4

PART FOUR

IN Memory的数据访问

IN Memory的访问 ---- SIMD

Oracle 12c采用了SIMD技术（Single Instruction processing Multiple Data values）使CPU能在一个指令当中访问多个数据



IN Memory的访问 ---- 单表访问案例

```
SQL> SET autotrace on
SQL> select count(*) from t1;

  COUNT(*)
----------
     90945


Execution Plan
----------------------------------------------------------
Plan hash value: 3724264953

----------------------------------------------------------------------------
| Id  | Operation                    | Name | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |     1 |     7   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE              |      |     1 |            |          |
|   2 |   TABLE ACCESS INMEMORY FULL| T1   | 99629 |     7   (0)| 00:00:01 |
----------------------------------------------------------------------------

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)


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

IN Memory的访问 ---- Bloom Filter

Bloom Filter是由一个很长的二进制向量和一系列随机映射函数组成，布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率（假正例False positives，即Bloom Filter报告某一元素存在于某集合中，但是实际上该元素并不在集合中）和删除困难，但是没有识别错误的情形（即假反例False negatives，如果某个元素确实没有在该集合中，那么Bloom Filter 是不会报告该元素存在于集合中的，所以不会漏报）。

IN Memory的访问 ---- Bloom Filter案例

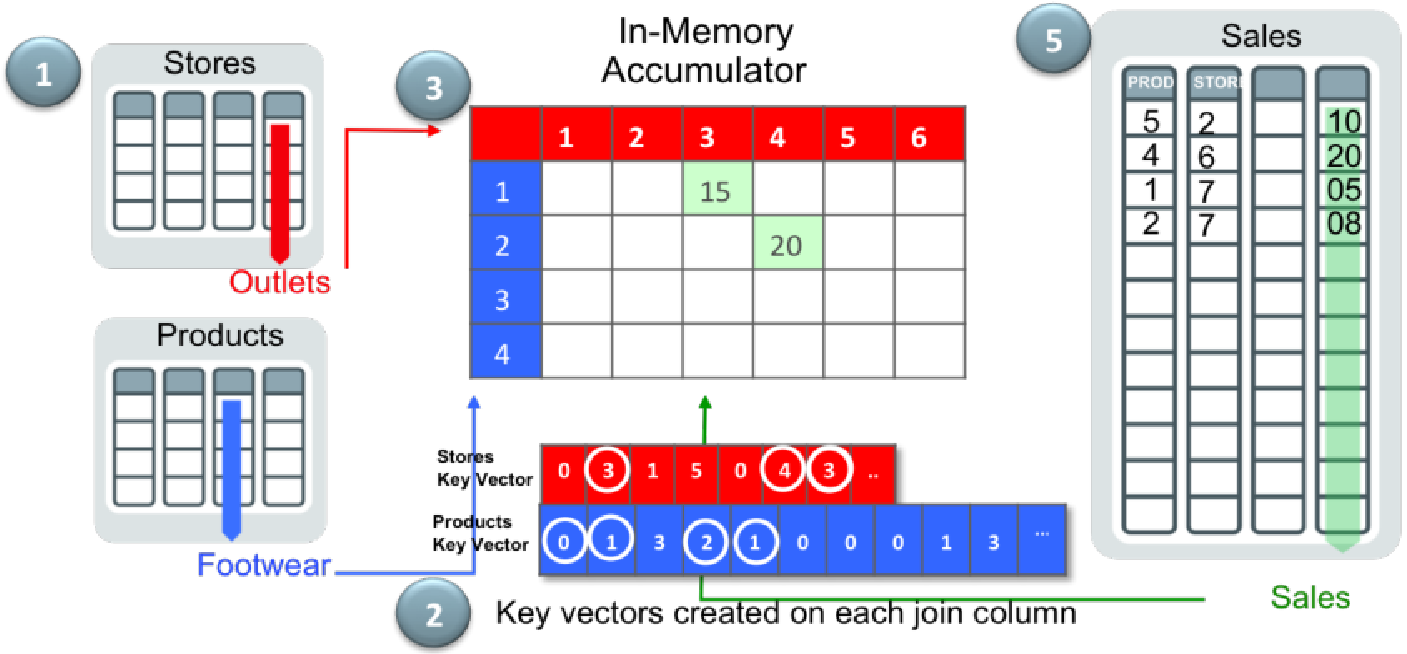
```
SQL> set autotrace on
SQL> ALTER table sh.sales inmemory;

Table altered.

SQL> ALTER table sh.times inmemory;

Table altered.

SQL> SELECT SUM(QUANTITY_SOLD * AMOUNT_SOLD) revenue
FROM sales s,
times d
WHERE s.time_id = d.time_id
AND s.channel_id BETWEEN 2 AND 3
AND d.time_id < to_date('20100601', 'yyyymmdd');  2    3    4    5    6

   REVENUE
----------
84221602.9
```

IN Memory的访问 ---- Bloom Filter案例

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1673358556

---------------------------------------------------------------------------------------------------------
 Id  | Operation                      | Name      | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
---------------------------------------------------------------------------------------------------------
   0 | SELECT STATEMENT               |           |     1 |    27 |   435   (3)| 00:00:01 |       |       |
   1 |  SORT AGGREGATE                |           |     1 |    27 |            |          |       |       |
*  2 |   HASH JOIN                    |           |  360K|  9517K|   435   (3)| 00:00:01 |       |       |
   3 |    JOIN FILTER CREATE          | :BF0001   |  360K|  9517K|   435   (3)| 00:00:01 |       |       |
   4 |     PART JOIN FILTER CREATE    | :BF0000   |  360K|  9517K|   435   (3)| 00:00:01 |       |       |
*  5 |      INDEX RANGE SCAN          | TIMES_PK  |  1826 | 14608 |     0   (0)| 00:00:01 |       |       |
   6 |    JOIN FILTER USE             | :BF0001   |  360K|  6697K|   434   (3)| 00:00:01 |       |       |
   7 |     PARTITION RANGE JOIN-FILTER|           |  360K|  6697K|   434   (3)| 00:00:01 |:BF0000|:BF0000|
*  8 |      TABLE ACCESS INMEMORY FULL| SALES     |  360K|  6697K|   434   (3)| 00:00:01 |:BF0000|:BF0000|
---------------------------------------------------------------------------------------------------------

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

  2 - access("S"."TIME_ID"="D"."TIME_ID")
  5 - access("D"."TIME_ID"<TO_DATE(' 2010-06-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
  8 - inmemory("S"."CHANNEL_ID"<=3 AND "S"."CHANNEL_ID">=2 AND "S"."TIME_ID"<TO_DATE(' 2010-06-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND SYS_OP_BLOOM_FILTER(:BF0001,"S"."TIME_ID"))
      filter("S"."CHANNEL_ID"<=3 AND "S"."CHANNEL_ID">=2 AND "S"."TIME_ID"<TO_DATE(' 2010-06-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND SYS_OP_BLOOM_FILTER(:BF0001,"S"."TIME_ID"))

ote
----
  - this is an adaptive plan


tatistics
----------------------------------------------------------
        285  recursive calls
          0  db block gets
       2133  consistent gets
          0  physical reads
          0  redo size
        545  bytes sent via SQL*Net to client
        551  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
         19  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

IN Memory的访问 ---- 多表连接

针对包含聚合和分组操作的多表链接，例如数据仓库中典型的星型查询，oracle提出了新的向量分组(Vector Group By)特性来提高select语句的性能。



IN Memory的访问 ---- 多表连接

阶段1

- 扫描会找到查询中数据量较小的如维度表（Dimension Table），扫描结果保存在一个新的数据结构Key Vector中，Key Vector和Bloom filter类似但不会返回 false positive
- 根据Key Vector同时也会在PGA中额外创建称为In-Memory Accumulator的多维数组生成若干临时表，这些临时表只包含select list中的字段

IN Memory的访问 ---- 多表连接

阶段2

- 扫描key vector和事实表满足条件的数据并添加到In-Memory Accumulator结构中
- 最终，In-Memory Accumulator结构join回临时表获得select list列

IN Memory的访问 ---- 多表连接

在整个过程中向量分组的构建和向量于事实表的比较都是在内存中完成的，而且也会使用SIMD，虽然可以极大的提升这种查询的性能，要求运行这种查询的进程拥有足够的PGA空间。

IN Memory的访问 ---- 多表连接案例

```
SQL>
SELECT /*+ VECTOR_TRANSFORM */  SUM(QUANTITY_SOLD * AMOUNT_SOLD) revenue
 -- ,p.prod_id, d.time_id
FROM sales s,
times d,
products p
WHERE s.time_id = d.time_id
AND s.prod_id = p.prod_id
AND s.channel_id BETWEEN 2 AND 3
AND d.time_id < to_date('20100601', 'yyyymmdd')
AND p.prod_name like '%P%'
group by p.prod_id, d.time_id;
SQL>   2    3    4    5    6    7    8    9   10   11
10467 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 1763863020

-----------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                              | Name                       | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |                            |  5165 |  342K|   131  (16)| 00:00:01 |       |       |
|   1 |  TEMP TABLE TRANSFORMATION             |                            |       |      |            |          |       |       |
|   2 |   LOAD AS SELECT                       | SYS_TEMP_0FD9D6679_18C10B  |       |      |            |          |       |       |
                                                                              PHASE ONE
|   3 |    VECTOR GROUP BY                     |                            |  1826 | 14608 |     4  (25)| 00:00:01 |       |       |
|   4 |     KEY VECTOR CREATE BUFFERED         | :KV0000                    |       |      |            |          |       |       |
|*  5 |      INDEX FAST FULL SCAN              | TIMES_PK                   |  1826 | 14608 |     3   (0)| 00:00:01 |       |       |
|   6 |   LOAD AS SELECT                       | SYS_TEMP_0FD9D667A_18C10B  |       |      |            |          |       |       |
|   7 |    VECTOR GROUP BY                     |                            |     4 |   120 |     2  (50)| 00:00:01 |       |       |
|   8 |     KEY VECTOR CREATE BUFFERED         | :KV0001                    |       |      |            |          |       |       |
|*  9 |      TABLE ACCESS INMEMORY FULL        | PRODUCTS                   |     4 |   120 |     1   (0)| 00:00:01 |       |       |
|  10 |   HASH GROUP BY                        |                            |  5165 |  342K|   126  (15)| 00:00:01 |       |       |
|* 11 |    HASH JOIN                           |                            |  5165 |  342K|   125  (14)| 00:00:01 |       |       |
|  12 |     TABLE ACCESS FULL                  | SYS_TEMP_0FD9D6679_18C10B  |  1826 | 14608 |     2   (0)| 00:00:01 |       |       |
|* 13 |     HASH JOIN                          |                            |  5165 |  302K|   123  (14)| 00:00:01 |       |       |
|  14 |      TABLE ACCESS FULL                 | SYS_TEMP_0FD9D667A_18C10B  |     4 |    84 |     2   (0)| 00:00:01 |       |       |
|  15 |      VIEW                              | VW_VT_0737CF93             |  5165 |  196K|   121  (15)| 00:00:01 |       |       |
|  16 |       VECTOR GROUP BY                  |                            |  5165 |  116K|   121  (15)| 00:00:01 |       |       |
|  17 |        HASH GROUP BY                   |                            |  5165 |  116K|   121  (15)| 00:00:01 |       |       |
|  18 |         KEY VECTOR USE                 | :KV0000                    |       |      |            |          |       |       |
                                                                                PHASE TWO
|  19 |          KEY VECTOR USE                | :KV0001                    |       |      |            |          |       |       |
|  20 |           PARTITION RANGE ALL          |                            |  806K |   17M|   120  (14)| 00:00:01 |     1 |    28 |
|* 21 |            TABLE ACCESS INMEMORY FULL  | SALES                      |  806K |   17M|   120  (14)| 00:00:01 |     1 |    28 |
-----------------------------------------------------------------------------------------------------------------------------------
```

IN Memory的访问 ---- 性能提升

column store对以下情况有显著性能提升

- Large scans that apply "=", "<", ">" and "IN" filters.
- Queries that return a small number of columns from a table with a large number of columns.
- Queries that join small tables to large tables.
- Queries that aggregate data.

在以下情况 IM column store对性能没有明显帮助

- Queries with complex predicates.
- Queries that return a large number of columns.
- Queries that return large numbers of rows.
- Queries with multiple large table joins.

Thank you！