Oracle 12c In-Memory Option

应用解析

DTCC

2015中国数据库技术大会

DATABASE TECHNOLOGY CONFERENCE CHINA 2015

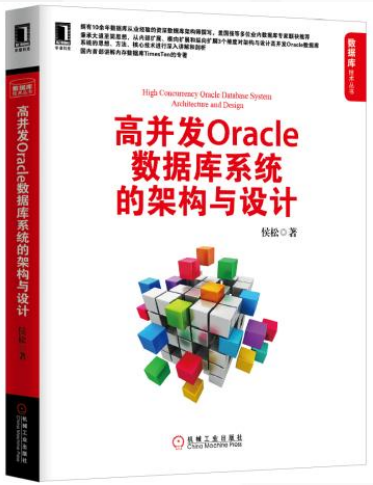
大数据技术探索和价值发现

IT168.com ChinaUnix ITPUB

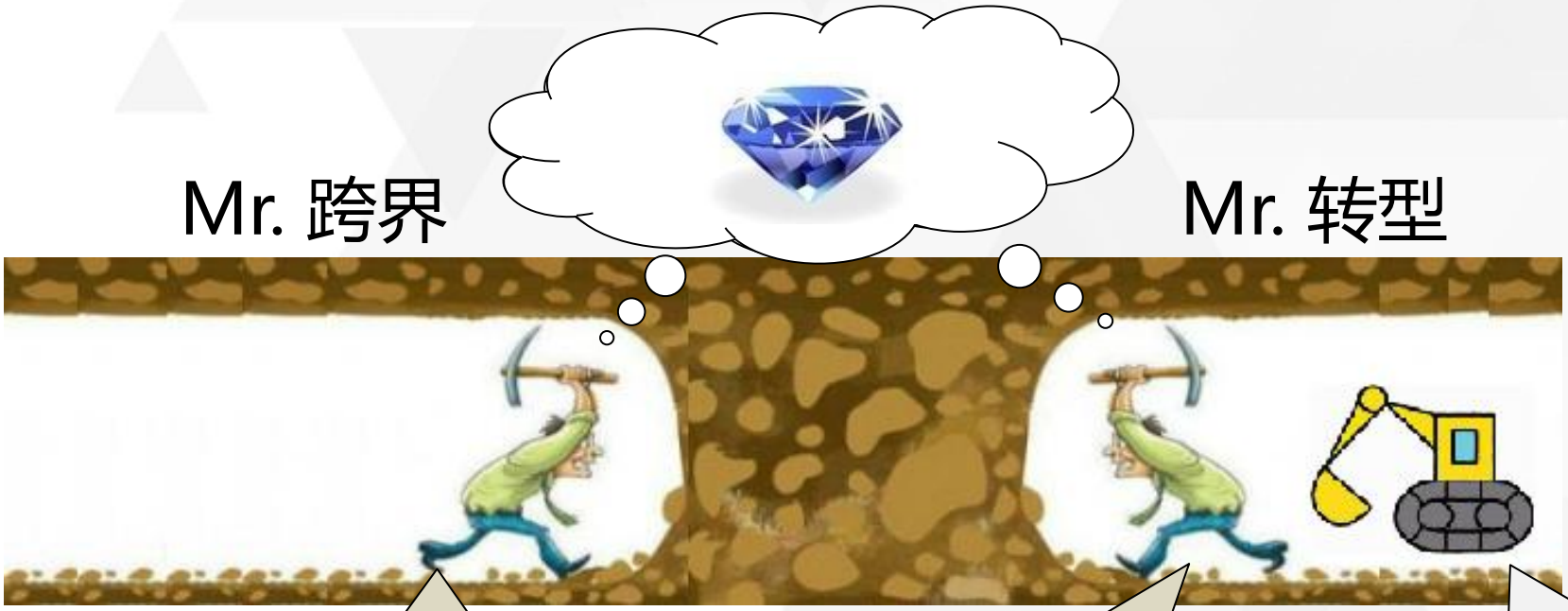
关于我

侯松 A

- 新浪微博：**@麻袋爸爸**
- 个人网站： http://www.housong.net
- 数据库架构师，ACOUG用户组核心成员，9i OCP，《高并发Oracle数据库系统的架构与设计》作者。
- 拥有10余年数据库开发、管理和运维经验。擅长项目管理及金融行业数据库应用架构设计。现任职于平安科技。
- 乐于分享与公益，曾担当甲骨文全球大会特别大使，WDP计划大学校园行志愿者。



闲话故事



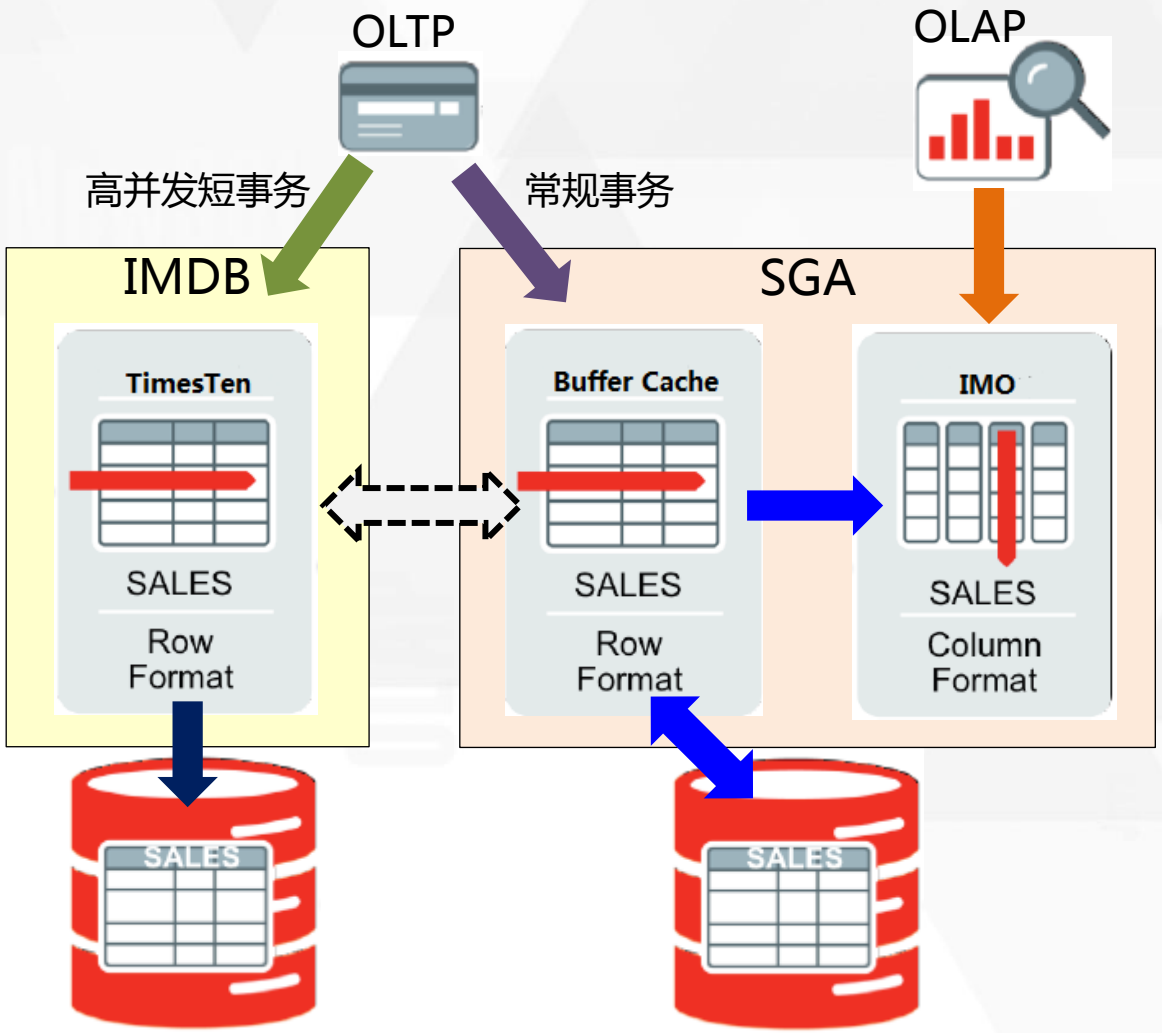
闲话故事


Google
MySQL
Baidu 百度
PostgreSQL
Tencent 腾讯
IBM
ORACLE
EMC
TREMD
Cassandra
HBASE
mongoDB
CouchDB
relax
riak
redis
amazon
Alibaba.com
阿里巴巴

目录

概述：IMDB

- TimesTen可独立运行或者与Oracle数据库配合使用。当与Oracle数据库配合使用时，将Oracle库中行式存储的表加载到TimesTen中，提供给前端应用访问。适合OLTP系统。
- IMO是在Oracle数据库的SGA区中开辟一块列式缓存，将Oracle中的行式数据转换成列式存储格式，然后提供给前端应用用来做查询分析。适合OLAP系统。



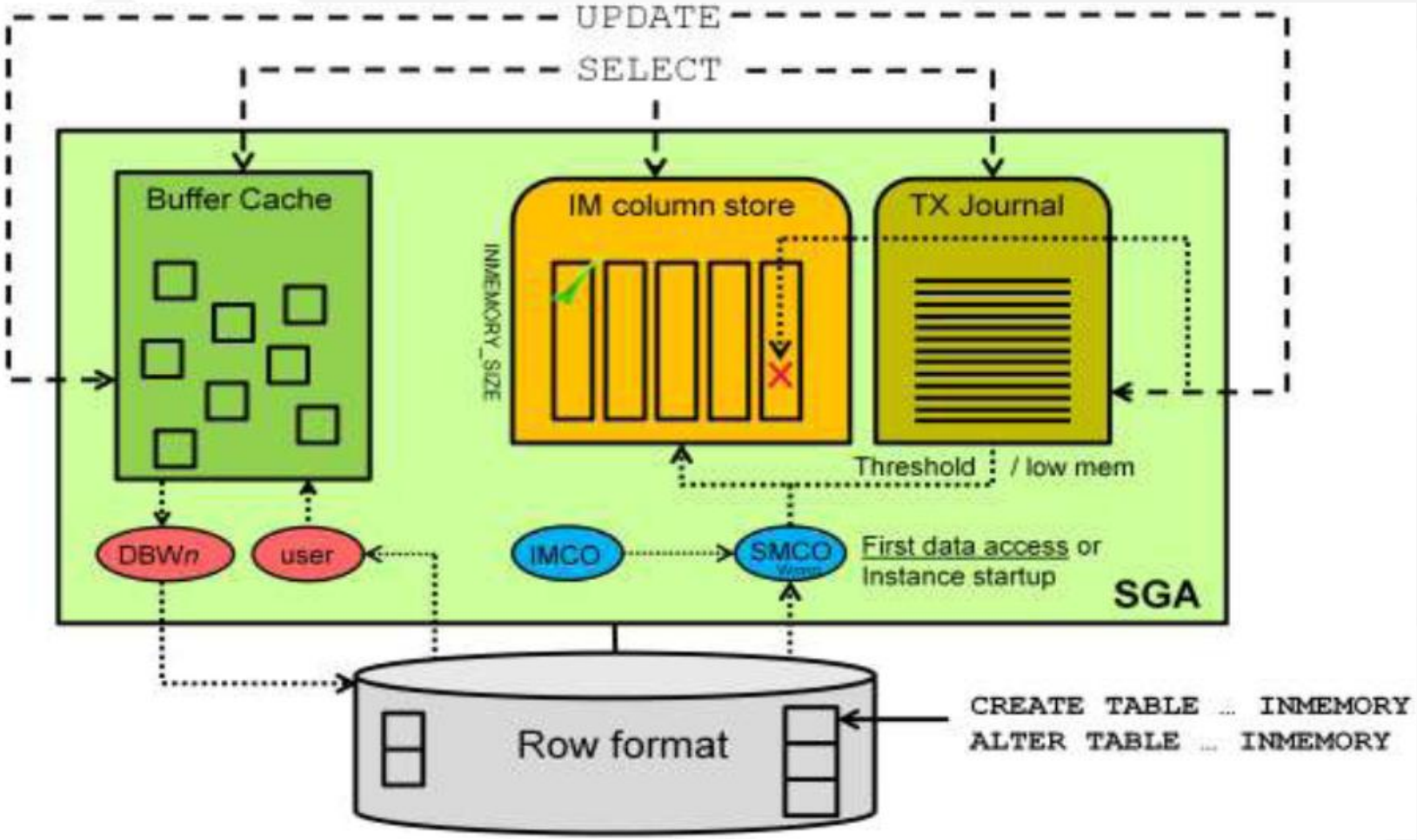
概述：In-Memory Option

IMO列式存储原理

- 依赖于行式存储的透明化的独立列式存储方式，可以与行式存储一起使用。
- 在存储上，IMO不需要额外的磁盘存储空间。
- 后台自动维护保证实时DML操作的数据读一致性。
- IMO特性无需业务应用变更，也无需数据库架构变更（安全、备份、容灾、RAC等）。

概述：IM内存结构

双重数据格式共同作用



概述：IMO标志特性

IMO的标志性特性：

查询优化

- IMO列式存储索引优化查询（In-Memory Storage Index）
- IMO优化的联立与聚合查询（Bloom Filter）
- SIMD矢量处理

列式存储

- 多级别二进制列式压缩（支持自选列列式存储）
- 行式存储与列式存储的透明化组合读写，并实时自动化后台同步
- IMO基于RAC架构的分布式应用

目录

- IMDB概述
- **查询优化**
- 列式存储
- 并发处理

神话：40万倍性能提升？

SQL语句：

```
SQL> select min(test_id), max(test_id),
  2         min(org_id), max(org_id)
  3    from alex.alex_test;
```

全表扫描执行计划：

```
-------------------------------------------------------------------------------------
| Id  | Operation          | Name                          | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                               |     1 |    16 |  1655K  (1)|
|   1 |  SORT AGGREGATE    |                               |     1 |    16 |            |
|   2 |   TABLE ACCESS FULL| ALEX_TEST                     |   206M|  3152M|  1655K  (1)|
-------------------------------------------------------------------------------------
```

IM扫描执行计划：

```
-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                          | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                               |     1 |    16 | 63273   (4)|
|   1 |  SORT AGGREGATE             |                               |     1 |    16 |            |
|   2 |   TABLE ACCESS INMEMORY FULL| ALEX_TEST                     |   206M|  3152M| 63273   (4)|
-------------------------------------------------------------------------------------------
```

神话：40万倍性能提升？

SQL语句：

```
SQL> select min(test_id), max(test_id),
  2         min(org_id), max(org_id)
  3    from alex.alex_test;
```

SQL语句统计信息：

	COST	CPU_TIME	ELAPSED_TIME	DISK_READS	BUFFER_GETS
行式存储	1655009	68678559	68868544	0	6102958
列式存储	63273	104984	104292	0	15
提升倍数	26	654	660	0	406864

会话级的统计信息：

	行式存储	列式存储
session logical reads	6102958	6103121
session logical reads – IM	0	6103106
IM scan CUs columns accessed	0	746
IM scan rows	0	206609165
IM scan rows valid	0	206609165

- ✓ 40万倍性能提升不成立
- ✓ 列式存储的IM逻辑读独立统计
- ✓ 列式存储的逻辑读包括IMCU的扫描和IM行扫描
- ✓ 性能提升倍数用执行时间来进行比较更为合理

神话：40万倍性能提升？

SQL语句：

```
SQL> select min(test_id), max(test_id),
  2         min(org_id), max(org_id)
  3    from alex.alex_test;
```

	Elapsed Time(s)	Cpu Time(s)	IO Waits(s)	Fetch Calls	Buffer Gets	Read Reqs	Read Bytes
FULL SCAN	166	140	27	1	6M	47759	47G
BUFFER CACHE	69	68	0.19	1	6M	0	0
DIRECT PATH	87	61	26	1	6M	47751	47G
IN-MEM SCAN	0.13	0.13	0	1	15	5	**90112**

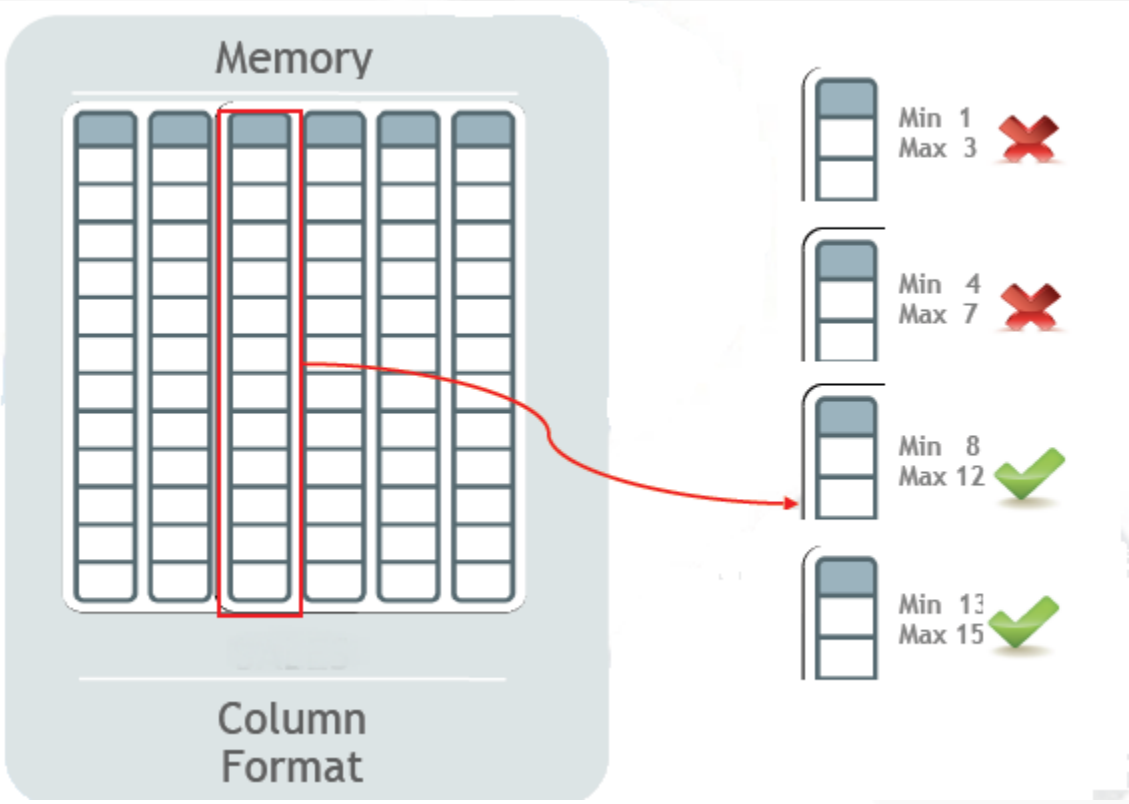
最大可达**1277倍**的性能提升

最大可达**1077倍**的性能提升

揭秘：列式存储索引

IMO列式存储索引：

- 列数据分割存储在多个IMCU内。
- 列式索引存储了每个IMCU的MIN/MAX值，优化MIN/MAX查询。
- 列式索引的存在，就避免了传统索引的创建，可以不必创建任何传统索引，在谓词过滤查询过程中，同样获得较大优势，也充分避免DML操作的索引维护成本。



揭秘：列式存储索引

In-Memory Storage Index：

- 查询V$IM_COL_CU视图，可以查看到IMCU的具体状态。
- 可以看到在IMCU的数据字典里记录了其最小值、最大值、长度条目数等信息。

```
SQL> select objd, tsn, length, dictionary_entries entries,
  2    minimum_value minval, maximum_value maxval
  3    from v$im_col_cu;
```

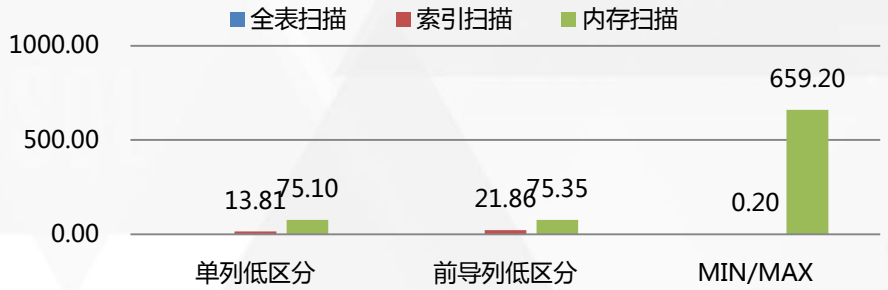
```
OBJD TSN   LENGTH ENTRIES MINVAL          MAXVAL
----- --- -------- ------- -------------- ---------------
92937   6  6648708  533688 C40C365A21     C50A082A2F0A
92937   6   744758     951 31             373030303137
.....
92937   6      984       3 3031           3032
92937   6  2303894  157580 786E0C0E021019 78950A0F0A3601
```

- IMCU会记录这些内容，是因为Oracle在数据加载时会自动创建各个列的列式索引，并会自动维护，这也是前言实例中为什么列式存储性能会提升600多倍的关键所在。
- 列式索引的存在，就避免了传统索引的创建，可以不必创建任何传统索引，在谓词过滤查询过程中，同样获得较大优势，也充分避免DML操作的索引维护成本。

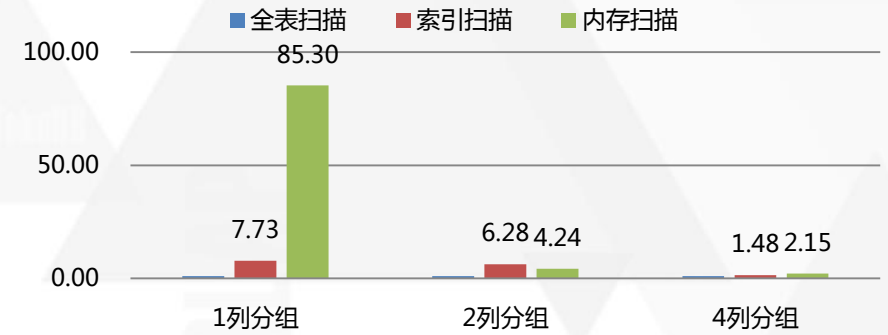
场景测试

IMO索引查询优化案例：

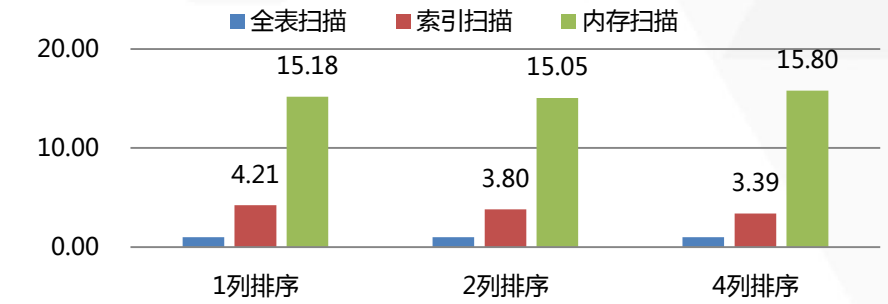
扫描	性能提升倍数（cache）		
	全表扫描	索引扫描	内存扫描
单列低区分	1.00	13.81	75.10
前导列低区分	1.00	21.86	75.35
MIN/MAX	1.00	0.20	659.20



分组	性能提升倍数（cache）		
	全表扫描	索引扫描	内存扫描
1列分组	1.00	7.73	85.30
2列分组	1.00	6.28	4.24
4列分组	1.00	1.48	2.15



排序	性能提升倍数（cache）		
	全表扫描	索引扫描	内存扫描
1列排序	1.00	4.21	15.18
2列排序	1.00	3.80	15.05
4列排序	1.00	3.39	15.80



IMO优势与劣势

优势

1. 查询操作扫描大量的行，并使用操作符进行数据过滤，操作符诸如：=、<、>、IN等；
2. 查询操作选择少量的列，而查询对象表或物化视图中包含了大量的列，比如：某个表包含100个列，而仅查询其中5个列；
3. 查询操作进行小表与大表的联立（Join）；
4. 查询要求进行数据聚合操作。

劣势

1. 进行复杂谓词的查询；
2. 查询操作选择大量的数据列；
3. 查询返回大量的数据行；
4. 查询要求进行大表复杂的联立（Join）。

IMO布隆过滤器

IMO的布隆过滤器：

- 在IMO列式存储中，当发生表和表的联立时，引进了布隆过滤器（Bloom Filters）。
- 当两个表发生联立的时候，特别是一个小表对一个大表发生哈希联立（Hash Join）的时候，Bloom Filters 的优势就非常明显了。
- 如下执行计划所示，其中DATE_DIM为维度表（小表），LINEORDER为事实表（大表）。

```
-----------------------------------------------------
| Id  | Operation                    | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |              |
|   1 |  SORT AGGREGATE              |              |
|*  2 |   HASH JOIN                  |              |
|   3 |    JOIN FILTER CREATE        | :BF0000      |
|*  4 |     TABLE ACCESS INMEMORY FULL| DATE_DIM    |
|   5 |    JOIN FILTER USE           | :BF0000      |
|*  6 |     TABLE ACCESS INMEMORY FULL| LINEORDER   |
-----------------------------------------------------
```

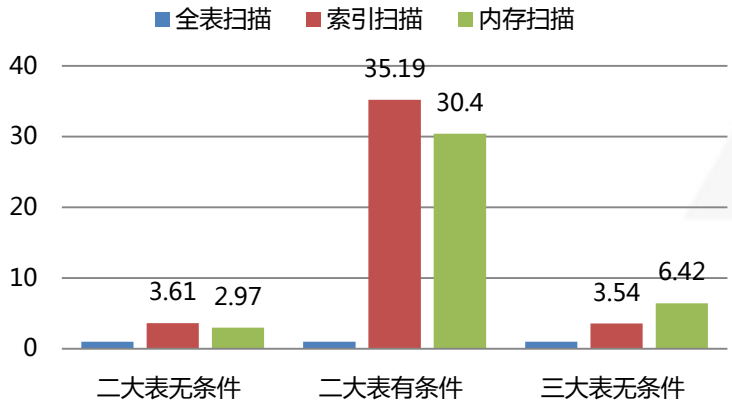
DATE_DIM表进行IM内存的全表扫描，谓词过滤后，Oracle为其生成一个名为“:BF0000”矢量或者Bloom Filters，以哈希表的形式保存在PGA中。

“:BF0000”即作为LINEORDER表进行IM内存扫描的一个组成部分，非常有效地提升了表哈希联立的性能。

IMO布隆过滤器

IMO联立查询优化案例：

联立	性能提升倍数（cache）		
	全表扫描	索引扫描	内存扫描
二大表无条件	1.00	3.61	2.97
二大表有条件	1.00	35.19	30.40
三大表无条件	1.00	3.54	6.42



--二大表有条件：

```
SQL> select count(x.test_id)
  2    from alex.alex_test         t,
  3         alex.alex_sales        x
  4   where t.test_id = x.test_id
  5   and x.test_id like '11%';
```

二大表有条件索引扫描：

```
-----------------------------------------------------------------
| Id  | Operation             | Name                          |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT      |                               |
|   1 |  SORT AGGREGATE       |                               |
|   2 |   NESTED LOOPS        |                               |
|*  3 |    INDEX FAST FULL SCAN| ALEX_SALES                   |
|*  4 |    INDEX UNIQUE SCAN  | PK_ALEX_TEST                  |
-----------------------------------------------------------------
```

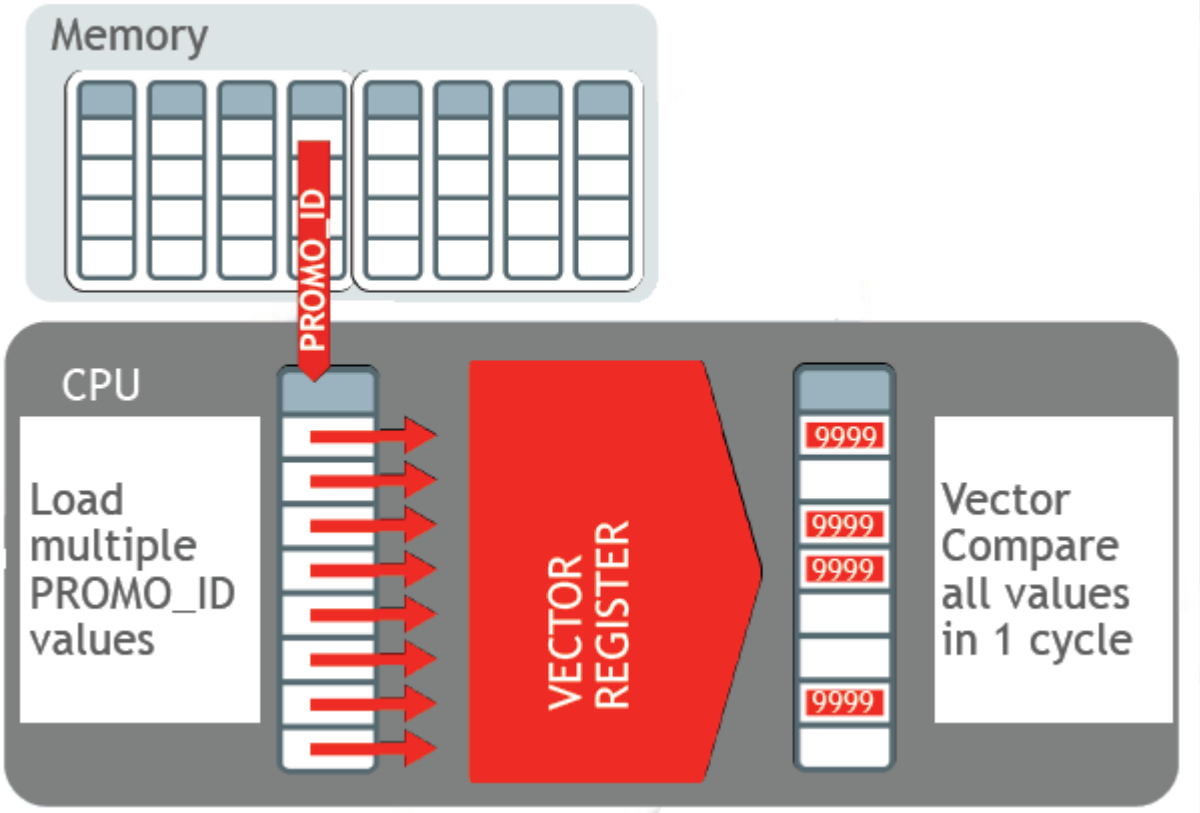
二大表有条件内存扫描：

```
-----------------------------------------------------------------------
| Id  | Operation                     | Name                          |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                               |
|   1 |  SORT AGGREGATE               |                               |
|*  2 |   HASH JOIN                   |                               |
|   3 |    JOIN FILTER CREATE         | :BF0000                       |
|*  4 |     TABLE ACCESS INMEMORY FULL| ALEX_SALES                    |
|   5 |    JOIN FILTER USE            | :BF0000                       |
|*  6 |     TABLE ACCESS INMEMORY FULL| ALEX_TEST                     |
-----------------------------------------------------------------------
```

SIMD矢量处理

SIMD处理方式：

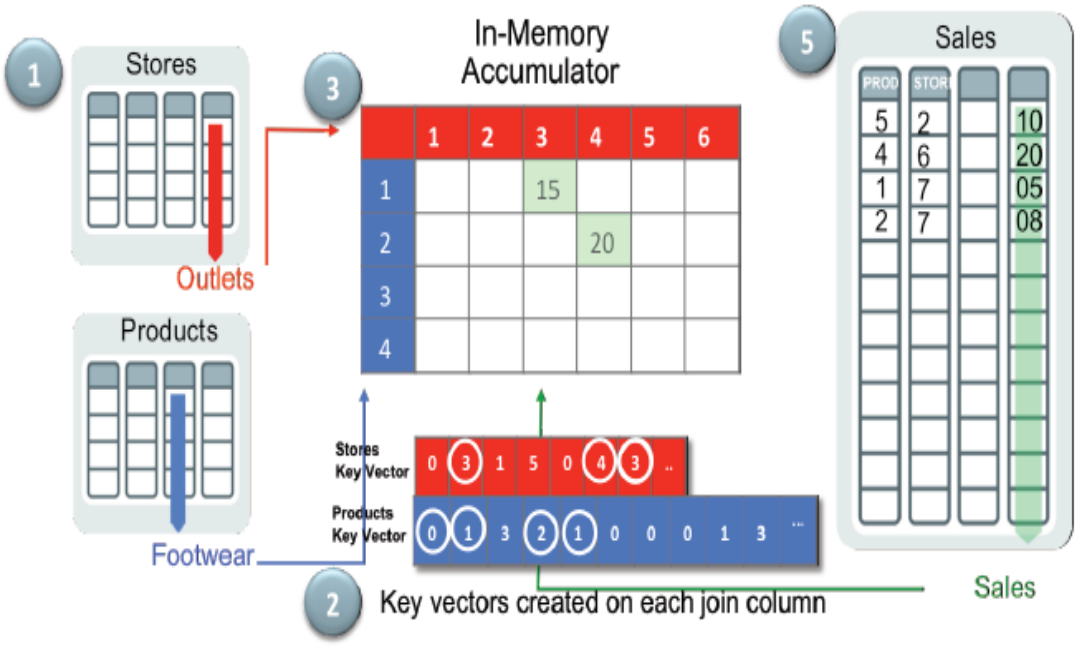
- 单指令多数据值的处理（Single Instruction processing Multiple Data values）
- 在一个单一CPU指令周期内进行一组列值的评估处理。
- 可以预期加快处理速度到每秒数十亿行。



SIMD矢量处理

SIMD矢量处理原理：

- 维度表（小表）与事实表（大表）的聚合计算在数据仓库的应用中是最为常见的，IMO基于SIMD矢量处理特点对列式存储的聚合计算也做了相对程度的优化。
- 如下是一个三个表的IM聚合过程，Stores、Products为维度表，Sales为事实表。
- 事实表的大小至少为维度表的10倍，随着维度越多聚合越复杂，该方法的效率将越高。



阶段一：准备

1. IM内存扫描各个维度表。
2. 两个表分布创建一个新的数据结构“Key Vector。
3. 根据各维度的Key Vector，在PGA中生成一个新的二维矢量数据的数据结构（In-Memory Accumulator）。
4. 根据选择的维度创建临时表

阶段二：计算

5. 应用各维度的Key Vector，IM扫描表Sales。
6. 与临时表进行Hash Join，输出最终结果。

SIMD矢量处理

IM

聚

3表星

3表雪

5表雪

40.00

30.00

20.00

10.00

0.00

--3表星

SQL> SEI

```
-----------------------------------------------------------------------------
| Id  | Operation                          | Name                         |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                              |
|   1 |  TEMP TABLE TRANSFORMATION         |                              |
|   2 |   LOAD AS SELECT                   | SYS_TEMP_0FD9D66E7_9322468   |
|   3 |    VECTOR GROUP BY                 |                              |
|   4 |     KEY VECTOR CREATE BUFFERED     | :KV0000                      |
|   5 |      TABLE ACCESS INMEMORY FULL    | ALEX_ORG_ID                  |
|   6 |   LOAD AS SELECT                   | SYS_TEMP_0FD9D66E8_9322468   |
|   7 |    VECTOR GROUP BY                 |                              |
|   8 |     KEY VECTOR CREATE BUFFERED     | :KV0001                      |
|   9 |      TABLE ACCESS INMEMORY FULL    | ALEX_CREATED_BY              |
|  10 |   HASH GROUP BY                    |                              |
|* 11 |    HASH JOIN                       |                              |
|* 12 |     TABLE ACCESS FULL              | SYS_TEMP_0FD9D66E7_9322468   |
|* 13 |     HASH JOIN                      |                              |
|  14 |      TABLE ACCESS FULL             | SYS_TEMP_0FD9D66E8_9322468   |
|  15 |      VIEW                          | VW_VT_1B35BA0F               |
|  16 |       VECTOR GROUP BY              |                              |
|  17 |        HASH GROUP BY               |                              |
|  18 |         KEY VECTOR USE             | :KV0001                      |
|  19 |          KEY VECTOR USE            | :KV0000                      |
|* 20 |           TABLE ACCESS INMEMORY FULL| ALEX_TASK                   |
-----------------------------------------------------------------------------
```

第一阶段

第二阶段

第一阶段

第二阶段

SIMD矢量处理

三表星型聚合查询优化案例：

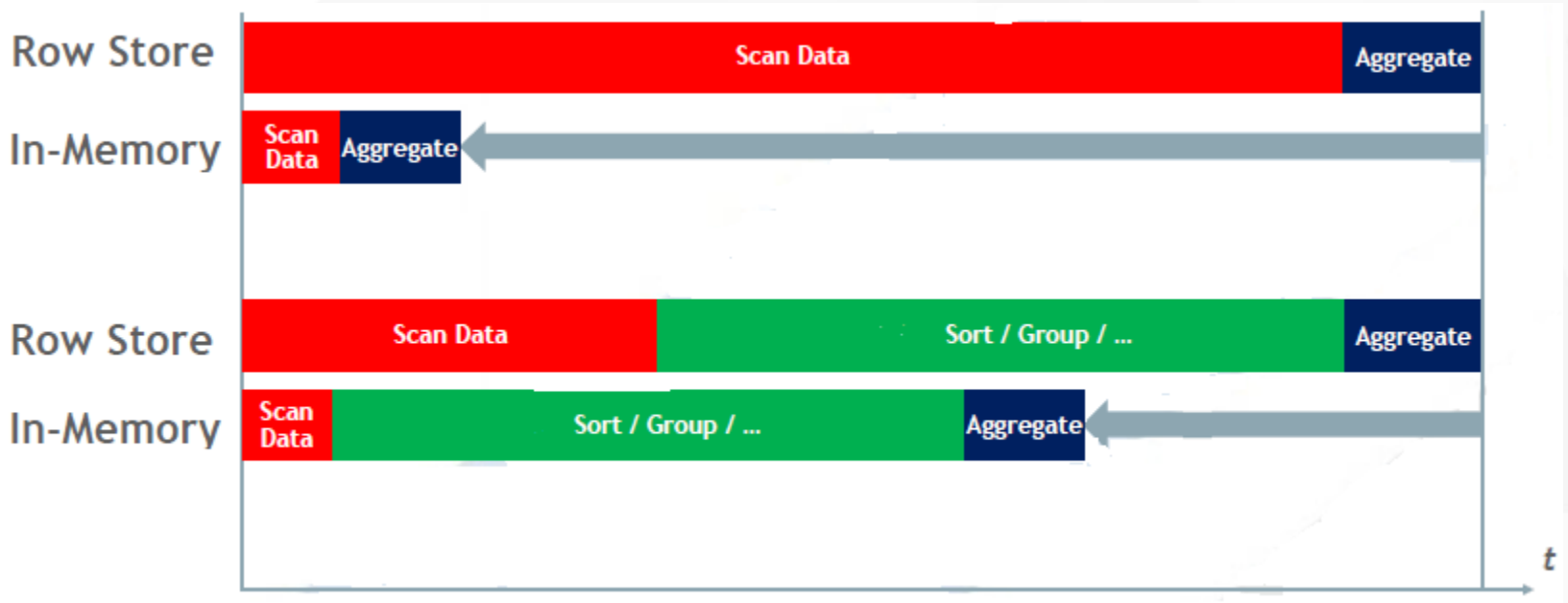
	Elapsed Time(s)	Cpu Time(s)	IO Waits(s)	Fetch Calls	Buffer Gets	Read Reqs	Read Bytes
FULL SCAN (cache)	69	69	0.13	1	6M	0	0
STAR TRANS (cache)	4.91	4.80	0.11	1	524k	2125	65M
IN-MEM SCAN (cache)	2.27	2.25	0.00	1	55	2	16384

IN-MEM SCAN执行计划分析：

Id	Operation	Name	Rows (Estim)	Cost	Time Active(s)	Start Active	Execs	Rows (Actual)	Read Reqs	Read Bytes	Write Reqs	Write Bytes	Mem (Max)	Activity (%)	Activity Detail (# samples)
0	SELECT STATEMENT				1	+2	1	13							
1	TEMP TABLE TRANSFORMATION				1	+2	1	13							
2	LOAD AS SELECT				1	+0	1	2			1	8192			
3	VECTOR GROUP BY		11	101	1	+0	1	11					7168		
4	KEY VECTOR CREATE BUFFERED	:KV0000			1	+0	1	560					28672		
5	TABLE ACCESS INMEMORY FULL	ALEX_ORG_ID	560	1	1	+0	1	560							
6	LOAD AS SELECT				1	+0	1	2			1	8192			
7	VECTOR GROUP BY		11	102	1	+0	1	11					9216		
8	KEY VECTOR CREATE BUFFERED	:KV0001			1	+0	1	1224					65536		
9	TABLE ACCESS INMEMORY FULL	ALEX_CREATED_BY	1224	1	1	+0	1	1224							
10	HASH GROUP BY		13	67055	1	+2	1	13					1M		
11	HASH JOIN		13	67054	1	+2	1	13					831K		
12	HASH JOIN		12	67052	1	+2	1	13					1M		
13	TABLE ACCESS FULL	SYS_TEMP_0FD9D66AE_96B9C90	10	2	1	+2	1	11	1	8192					
14	VIEW	VW_VT_1B35BA0F	13	67050	1	+2	1	13							
15	VECTOR GROUP BY		13	67050	1	+2	1	13					13312		
16	HASH GROUP BY		13	67050	1	+2	1	0							
17	KEY VECTOR USE	:KV0000			1	+2	1	572K							
18	KEY VECTOR USE	:KV0001			1	+2	1	572K							
19	TABLE ACCESS INMEMORY FULL	ALEX_TASK	534K	64544	2	+2	1	572K						100.00	in memory (2)
20	TABLE ACCESS FULL	SYS_TEMP_0FD9D66AF_96B9C90	11	2	1	+2	1	11	1	8192					

查询优化小结

- IMO特性能极大程度地提升查询性能，特别是数据扫描（Data Scanning）方面的。
- Joins和Vector-By聚合也可以预期非常高的性能优势。
- 然而，没有一项技术可以成为魔术，Sorting、常规聚合等仍然会消耗较长时间。



目录

- IMDB概述
- 查询优化
- **列式存储**
- 并发处理

压缩级别

IMO列式压缩的级别

压缩级别	说明
NO MEMCOMPRESS	数据无压缩（但是做了列式存储后，较行式存储仍有一定压缩量。）
MEMCOMPRESS FOR DML	最小级别的压缩，有利于DML操作的性能优化。
MEMCOMPRESS FOR QUERY LOW	最优化查询性能（缺省值）
MEMCOMPRESS FOR QUERY HIGH	查询性能优化，同时获得更大空间压缩比。
MEMCOMPRESS FOR CAPACITY LOW	获得更大的空间压缩量。
MEMCOMPRESS FOR CAPACITY HIGH	最优化空间节省，活动最大的压缩比。

- ✓ IMO进行列式压缩后，大致可以获得2~20倍的空间节省，不仅取决于压缩级别选项，同样依赖于数据本身的特点。
- ✓ MEMCOMPRESS FOR CAPACITY HIGH虽然压缩比最大，但在查询性能上不会有太大的劣势。

压缩级别

IMO列式压缩的级别

优先级别	说明
CRITICAL	数据库打开的时候，IM对象就自动加载。
HIGH	CRITICAL级别对象加载完成后，如果空间足够，自动加载该级别对象。
MEDIUM	CRITICAL、HIGH级别对象加载完成后，如果空间足够，自动加载该级别对象。
LOW	CRITICAL、HIGH、MEDIUM级别对象加载完成后，如果空间足够，自动加载该级别对象。
NONE	该级别为缺省值，当IM对象被第一次扫描时，如果空间足够，触发该对象的加载。

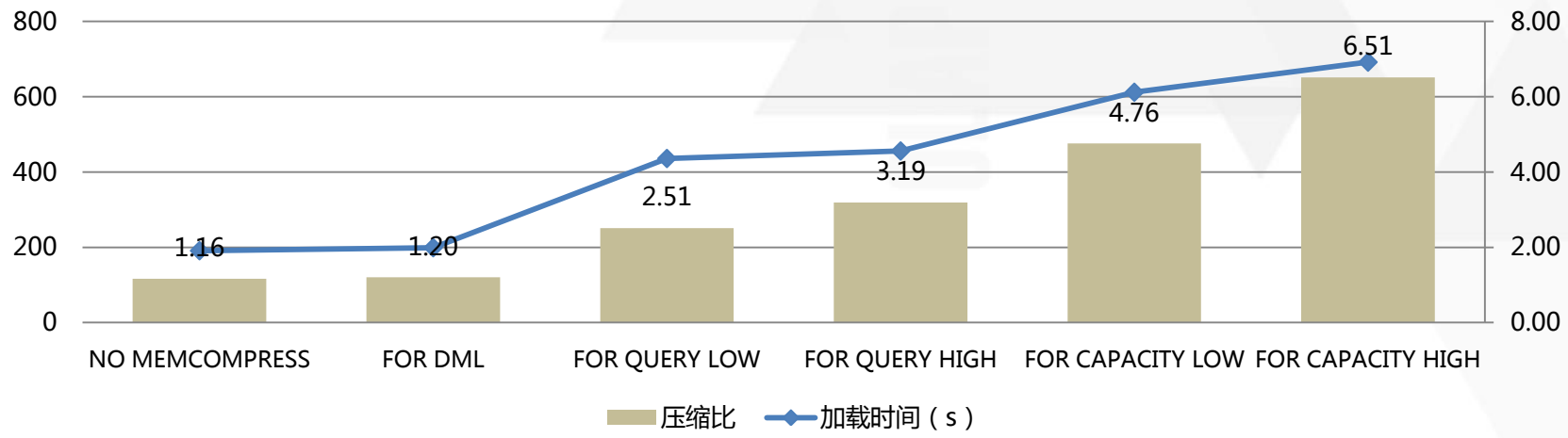
- ✓ 非必要情况，尽可能选择默认的NONE级别，提升数据库启动过程的效率。
- ✓ 对于必要加载对象，可以选择非繁忙时段，手工触发加载，因为加载过程会有较高的CPU开销。

压缩对比测试

列式存储加载压缩测试：

- 以测试大表ALEX_TEST为例，各级别压缩比为1.16~6.51，最大未及10倍压缩。
- 从加载速度上来看，开启10个加载进程，加载速度与压缩比呈正比例上升趋势。

Wnnn数	表名	行数	表大小（GB）	缓存大小（GB）	缓存类型	加载时间（s）	压缩比
10	ALEX_TEST	206,609,165	46.61	40.09	NO MEMCOMPRESS	191	1.16
10	ALEX_TEST	206,609,165	46.61	38.79	FOR DML	199	1.20
10	ALEX_TEST	206,609,165	46.61	18.59	FOR QUERY LOW	436	2.51
10	ALEX_TEST	206,609,165	46.61	14.6	FOR QUERY HIGH	456	3.19
10	ALEX_TEST	206,609,165	46.61	9.8	FOR CAPACITY LOW	612	4.76
10	ALEX_TEST	206,609,165	46.61	7.16	FOR CAPACITY HIGH	692	6.51



数据加载

列式存储加载压缩测试：

```
top - 14:33:44 up 23:39,  6 users,  load average: 10.96, 9.02, 7.87
Tasks: 1812 total,  14 running, 1797 sleeping,   0 stopped,   1 zombie
Cpu(s): 44.3%us,  5.1%sy,  0.0%ni, 47.6%id,  2.5%wa,  0.1%hi,  0.5%si,  0.0%st
Mem:  264266376k total, 262009960k used,  2256416k free,   274016k buffers
Swap: 18907128k total, 17438964k used,  1468164k free, 199443768k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7785      12c   25   0  168g 7.6g 7.4g R 99.0  3.0  31:23.43 ora_w000_test12
 9442      12c   25   0  169g  19g  19g R 99.0  7.7  33:22.83 ora_w002_test12
 9522      12c   25   0  168g  16g  16g R 99.0  6.5  21:27.28 ora_w003_test12
 9914      12c   25   0  168g  18g  18g R 99.0  7.2  30:06.77 ora_w005_test12
 9947      12c   25   0  169g  18g  18g R 99.0  7.3  29:13.75 ora_w009_test12
 9527      12c   25   0  168g  18g  18g R 98.6  7.3  30:03.75 ora_w004_test12
 9932      12c   25   0  168g  18g  18g R 98.6  7.4  31:08.01 ora_w008_test12
 9969      12c   25   0  169g  17g  17g R 98.6  7.1  27:09.91 ora_w00a_test12
 7795      12c   25   0  168g 7.4g 7.4g R 98.3  3.0  31:10.15 ora_w001_test12
 9929      12c   25   0  169g  18g  18g R 97.4  7.5  30:22.73 ora_w007_test12
```

通过top命令监视，可见在Wnnn进程进行IM数据加载的过程中，是非常消耗CPU资源的，10个Wnnn进程几乎都独占了一个逻辑CPU。进一步证明了数据加载过程不可以在实例启动时进行。

数据加载

列式存储压缩比估算方法：

```
DECLARE
  l_blkcnt_cmp       PLS_INTEGER;
  l_blkcnt_uncmp     PLS_INTEGER;
  l_row_cmp          PLS_INTEGER;
  l_row_uncmp        PLS_INTEGER;
  l_cmp_ratio        PLS_INTEGER;
  l_comptype_str     VARCHAR2(100);
  comp_ratio_allrows NUMBER := -1;
BEGIN
  dbms_compression.get_compression_ratio(
                   -- Input parameters
                   scratchtbsname => 'ALEXTBS',
                   ownname        => 'ALEX',
                   objname        => 'ALEX_TEST',

dbms_compression.

dbms_compression.c
  dbms_output.Put_line('The IM compression ratio is ' || l_cmp_ratio);
END;
```

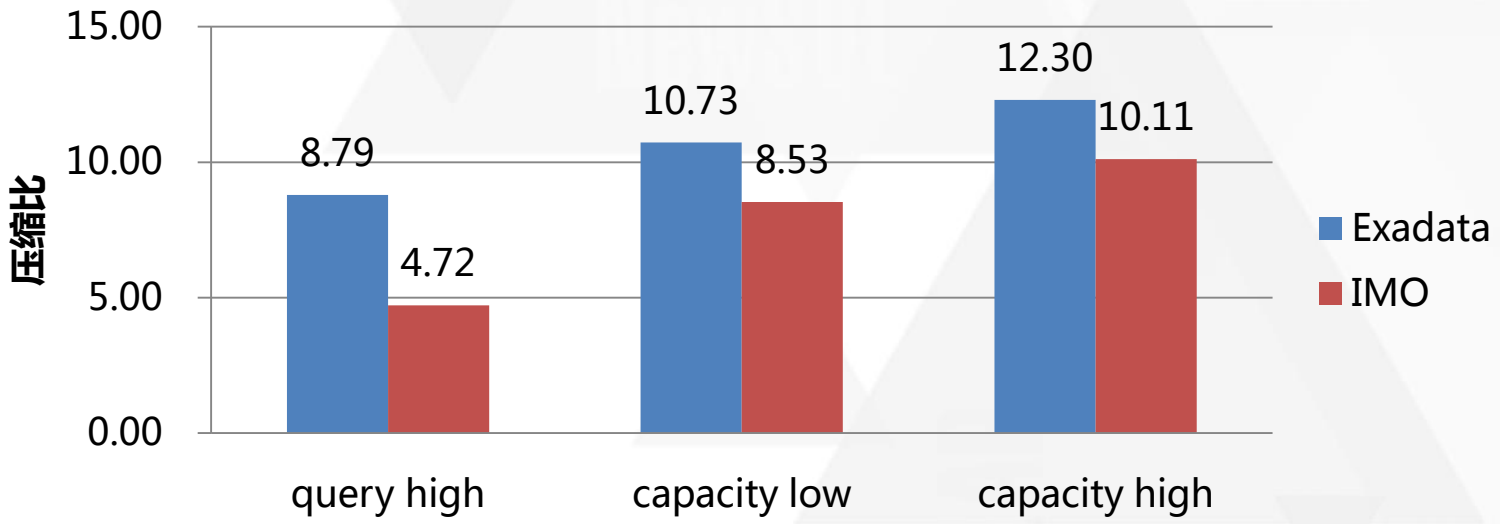
```
SQL> select segment_name,
  2         round(bytes / 1024 / 1024 / 1024, 2) seg_size,
  3         round(inmemory_size / 1024 / 1024 / 1024, 2) inmem_size,
  4         round(bytes_not_populated / 1024 / 1024 / 1024, 2) progress_size,
  5         inmemory_compression inmemory_comp,
  6         populate_status
  7    from v$im_segments;

SEGMENT_NAME  SEG_SIZE INMEM_SIZE PROGRESS_SIZE INMEMORY_COMP  POPULATE_STATUS
------------- -------- ---------- ------------- -------------- ---------------
CMP3$92937       46.87       14.8         16.23 FOR QUERY LOW  STARTED
```

- 此方法实际上就是通过进行列式缓存来进行估算，成本开销过大。
- 尽量不要使用。

压缩对比测试

IMO与同类DB压缩对比：

- 下例为IMO与Exadata上，财务库用户帐户信息表（压缩前约35GB大小）的对比测试结果。



- 下例为IMO与HANA上，财务库总账平衡表的对比测试结果。

表名	表大小（GB）	缓存大小（GB）	缓存类型	压缩比
GL_BALANCES	251.37	93.41	FOR QUERY HIGH	2.69
		67.78	FOR CAPACITY LOW	3.71
		52.37	SAP HANA	**4.80**
		51.53	FOR CAPACITY HIGH	4.88

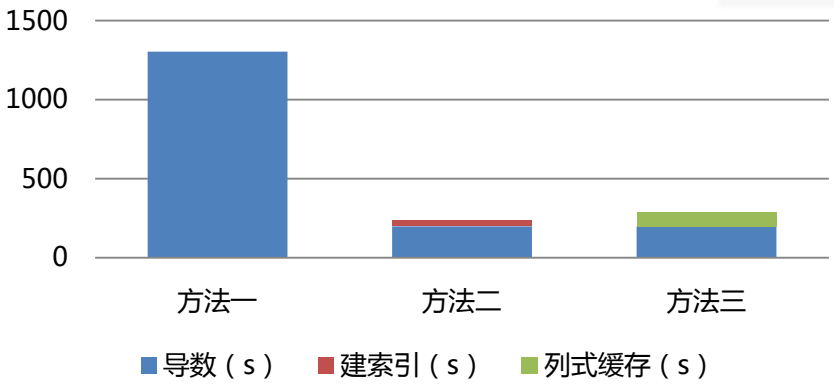
批量导数

事实表（Fact Table）批量导数：

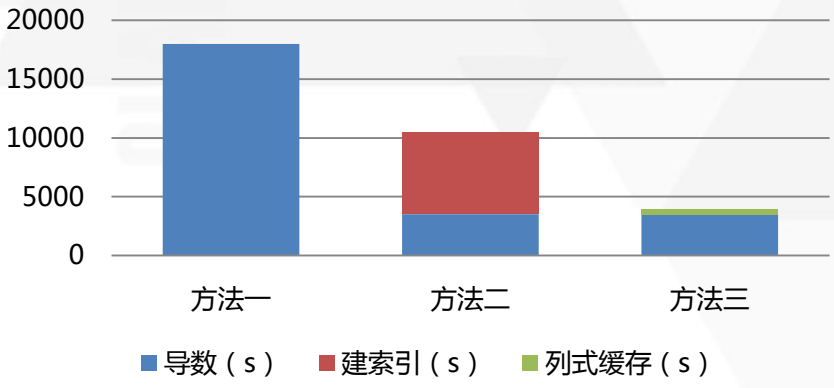
- 对于OLAP的应用来说，常见事实表的批量导数操作，极少出现短小事务行为。
- 通常情况，事实表上有较多数量的索引，批量导数时，会先删除索引，在完成导数后再重建。
- 列式存储事实表后，可以不建任何传统索引，同样可以获得高效查询能力，且避免了导数带来的索引维护成本，随着表的大小越大，其优势越明显。

表名	操作方法	大小（GB）	索引数	导数（s）	建索引（s）	列式缓存（s）	总时间（s）
TASK_TEST	方法一：建立索引，导数	2.25	5	1305.01	0	0	1305.01
	方法二：导数，重建索引			198.53	35.48	0	234.01
	方法三：导数，列式缓存			198.53	0	90	288.53
TASK_TEST	方法一：建立索引，导数	46.61	5	18000	0	0	18000
	方法二：导数，重建索引			3512.57	6949.2	0	10461.77
	方法三：导数，列式缓存			3512.57	0	436	3948.57

2G表批量导数



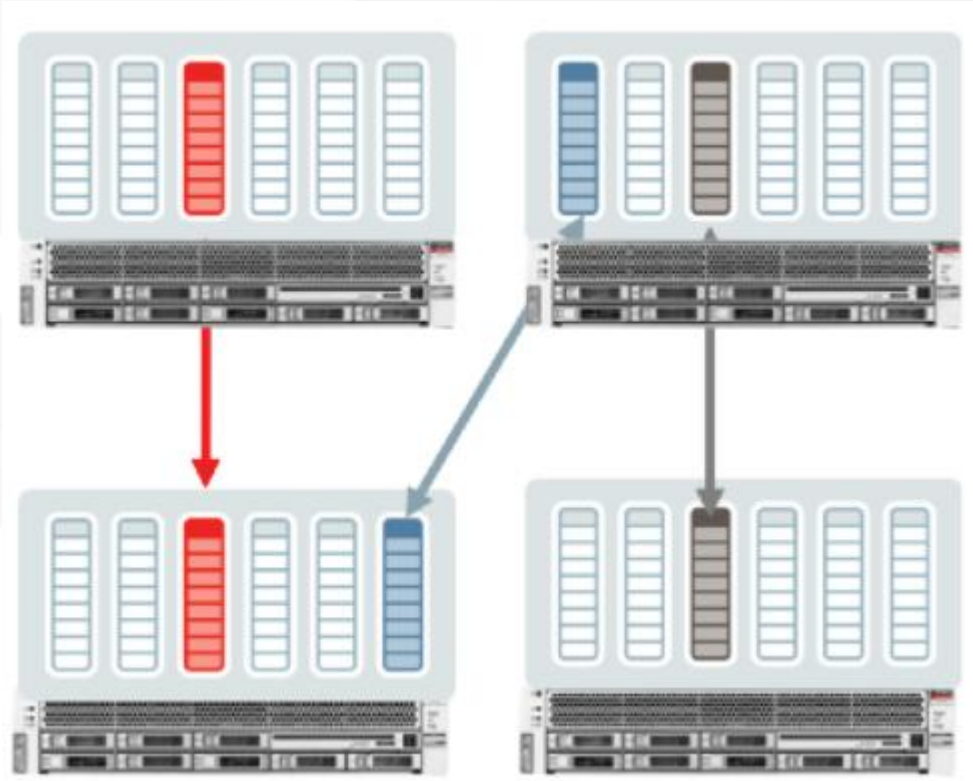
40G表批量导数



RAC应用

IMO基于RAC架构的分布式应用

- 大对象（分区表）IMCUs分布式存储：
  - AUTO（默认）
  - BY ROWID RANGE
  - BY {SUB}PARTITION
- 容错
  - DISTRIBUTE子句控制IMCU在RAC节点间的副本冗余方式
  - DISTRIBUTE ALL表示每个IMCU在所有RAC节点存储副本



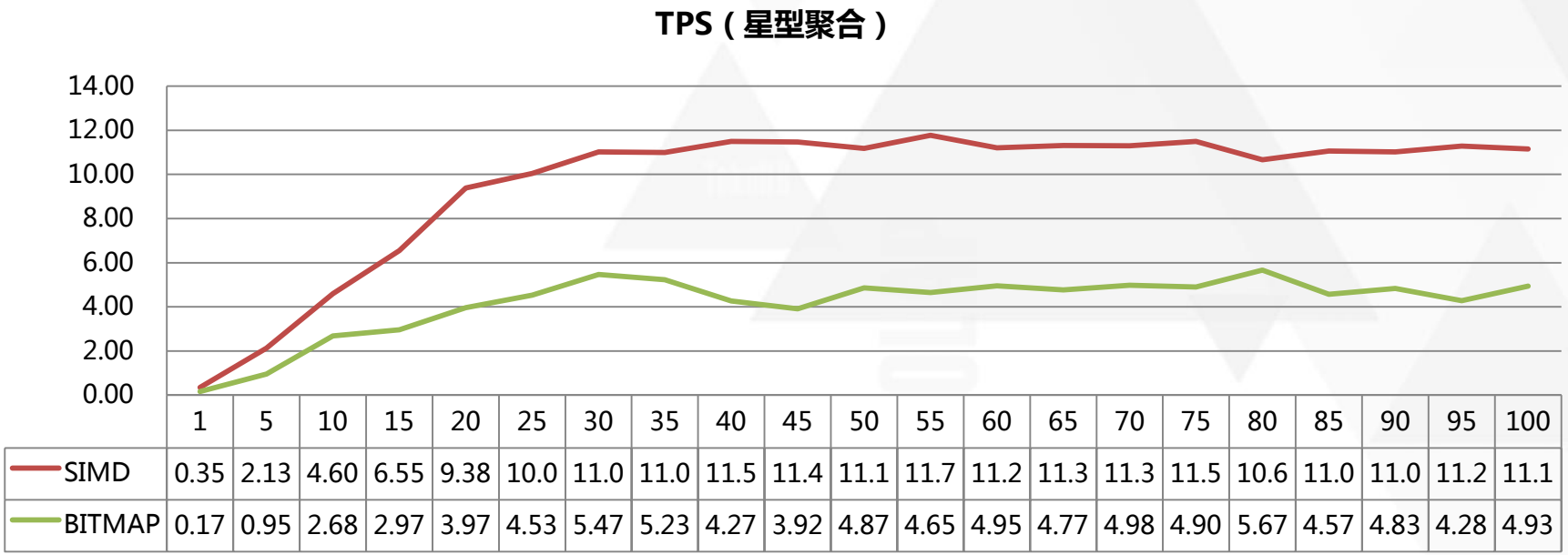
目录

- IMDB概述
- 查询优化
- 列式存储
- **并发处理**

并发星型聚合

星型聚合并发场景测试：

- 前面提到在星型聚合的场景测试中，IMO的SIMD查询方式较传统Bitmap星型转换查询方式的性能提升3~4倍；
- 当面对高并发查询的时候，SIMD的性能优势并没有减弱；
- 然而，并发压力较大的时候，Bitmap星型转换会需要使用较多PGA空间。



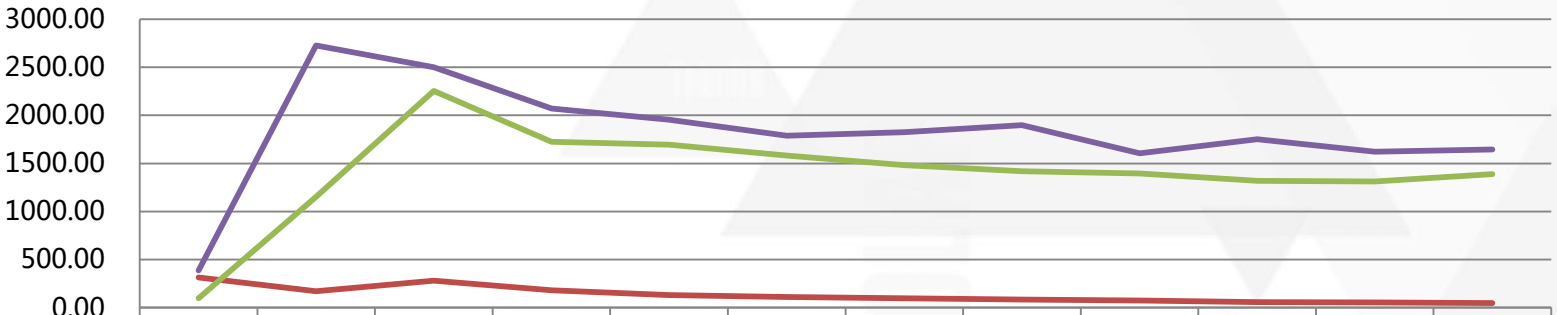
	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
SIMD	0.35	2.13	4.60	6.55	9.38	10.0	11.0	11.0	11.5	11.4	11.1	11.7	11.2	11.3	11.3	11.5	10.6	11.0	11.0	11.2	11.1
BITMAP	0.17	0.95	2.68	2.97	3.97	4.53	5.47	5.23	4.27	3.92	4.87	4.65	4.95	4.77	4.98	4.90	5.67	4.57	4.83	4.28	4.93

并发短事务处理

短事务并发场景测试：

- OLTP应用的短事务处理，概念中IMO为是列式存储，将会较大的性能劣势，然而事实并非如此。
- In-Mem较之No-Index仅有10%~20%的性能下降，列式存储并未出现预期的较大性能损失。
- 而Bitmap中，因为DML需要进行位图索引的维护，出现大量的enq: TX - row lock contention等待时间，整体性能下降明显。
- 之所以，IMO应用并没有像索引应用一样，造成短事务处理的劣势，是因为IMO的维护是异步完成的，后台进程自动维护保证DML操作的数据读一致性。

TPS（短事务）



	1	5	10	20	30	40	50	60	70	80	90	100
Bitmap	313.47	171.37	280.92	180.87	132.93	110.10	99.02	86.12	75.58	58.40	55.45	48.45
In-Mem	96.80	1151.98	2252.52	1724.50	1694.57	1582.62	1482.73	1419.53	1394.55	1317.48	1312.52	1388.22
No-Index	386.10	2726.63	2499.65	2072.22	1955.83	1787.60	1826.05	1899.27	1603.57	1751.45	1623.23	1645.05

并发短事务处理

Top 10 Foreground Events by Total Wait Time

Event	Waits	Total Wait Time (sec)	Wait Avg(ms)	% DB time	Wait Class
buffer busy waits	3,801,916	40.4K	10.62	66.1	Concurrency
enq: TX - row lock contention	1,929,423	16.3K	8.47	26.8	Application
DB CPU		4672.7		7.7	
latch: cache buffers chains	1,900,989	972.6	0.51	1.6	Concurrency
log file switch (checkpoint incomplete)	63	55.8	885.19	.1	Configuration

Bitmap，Top10等待事件：
出现大量的enq: TX - row lock contention等待事件，
整体性能下降明显。

Top 10 Foreground Events by Total Wait Time

Event	Waits	Total Wait Time (sec)	Wait Avg(ms)	% DB time	Wait Class
buffer busy waits	89,034	1871.5	21.02	66.2	Concurrency
IM buffer busy	174,513	496.3	2.84	17.5	Concurrency
DB CPU		380.4		13.4	
latch: cache buffers chains	105,736	106.4	1.01	3.8	Concurrency
db file sequential read	1,763	6.5	3.70	.2	User I/O

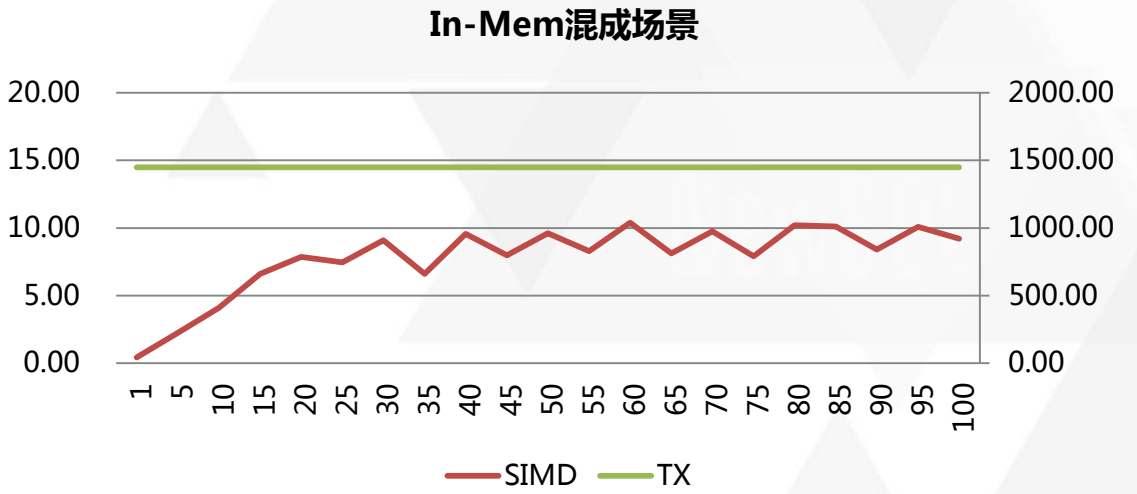
In-Mem，Top10等待事件：
出现预期内的buffer busy waits之外，还出现了IM buffer busy的等待，此为Oracle维护IMCU journal来保证数据一致读的行为，也是性能下降的原因。

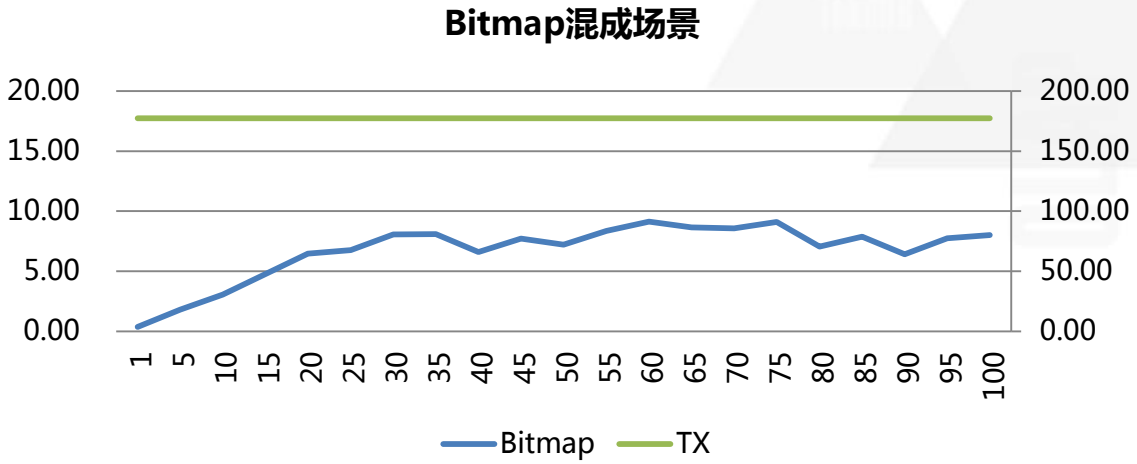
Top 10 Foreground Events by Total Wait Time

Event	Waits	Total Wait Time (sec)	Wait Avg(ms)	% DB time	Wait Class
buffer busy waits	113,255	1984	17.52	86.9	Concurrency
DB CPU		242.4		10.6	
latch: cache buffers chains	137,140	79.6	0.58	3.5	Concurrency
buffer deadlock	35,088	2.6	0.07	.1	Other
library cache: mutex X	2,645	1.8	0.67	.1	Concurrency
log file sync	566	1.4	2.46	.1	Commit
log file switch completion	22	.8	35.76	.0	Configuration
db file sequential read	289	.5	1.82	.0	User I/O
enq: SQ - contention	440	.5	1.04	.0	Configuration
cursor: pin S	245	.3	1.26	.0	Concurrency

No-Index，Top10等待事件：
无特别值得关注点。

混合并发场景





混合并发场景测试：

- 将前面两个并发测试场景（星型聚合查询、短事务处理）混合起来测试，模拟MIS库的操作行为。
- 其中，短事务处理持续为并发度为100，而星型聚合查询为并发度递增，如图横坐标所示。
- Bitmap的测试结果显示，短事务处理和星型聚合查询TPS并无较大损失。
- In-Mem的测试结果显示，短事务处理和星型聚合查询TPS也并无较大损失，同时保持了较Bitmap的优势。
- 可见，IMO对于MIS库的混合场景应用是有较大益处的。


ChinaUnix
THANKS