
ORACLE®

#odddtky

Oracle DBA & Developer Days 2014

ハンズオン：
Oracle Database In-Memory
リアルタイム分析を可能にする
Oracle Database In-Memoryの速さを実感!

日本オラクル株式会社

データベース事業統括 - 製品戦略統括本部
データベースエンジニアリング本部

for your **Skill**

使える実践的なノウハウがここにある

Oracle
DBA & Developer Days
2014

- 以下の事項は、弊社の一般的な製品の方向性に関する概要を説明するものです。また、情報提供を唯一の目的とするものであり、いかなる契約にも組み込むことはできません。以下の事項は、マテリアルやコード、機能を提供することをコミットメント（確約）するものではないため、購買決定を行う際の判断材料になさらないで下さい。オラクル製品に関して記載されている機能の開発、リリースおよび時期については、弊社の裁量により決定されます。

アジェンダ

- 概要
- はじめる前に
- Lesson 1 設定
- Lesson 2 インメモリ表スキャン
- Lesson 3 インメモリ結合と集計演算

ハンズオン実習概要

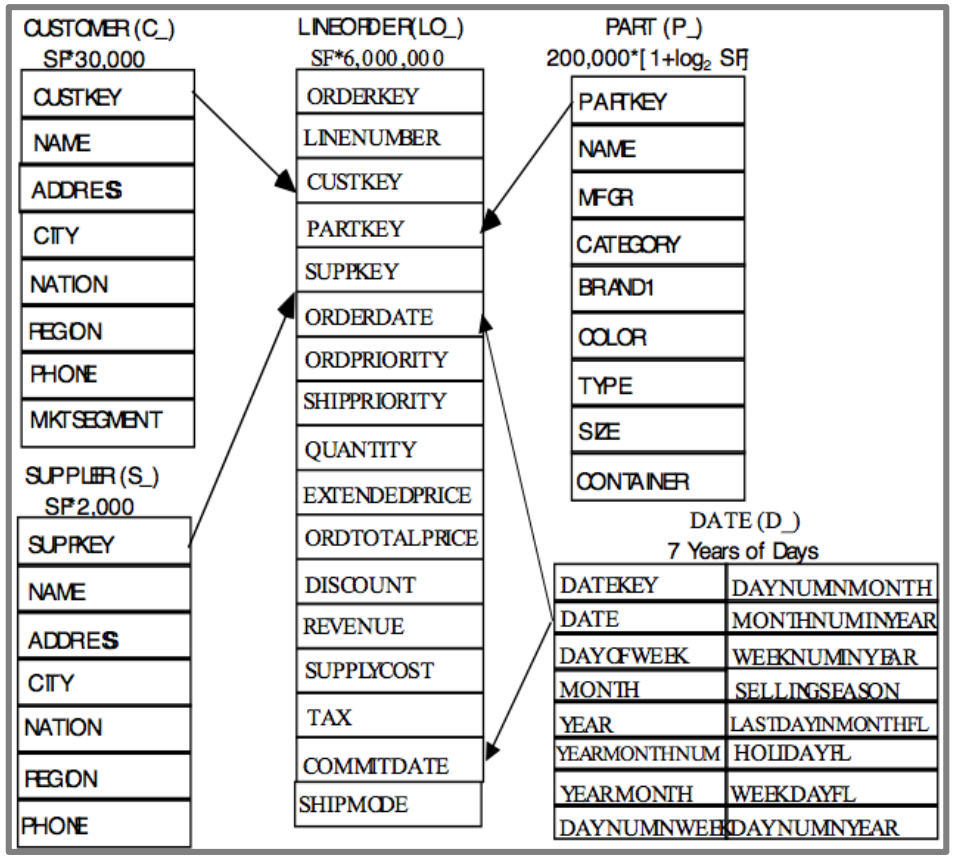
インメモリ・ハンズオン実習

概要

- ゴール
  - 新しいインメモリ・カラム・ストアが
    簡単な設定で、大きな効果を得られることを実習します

- 実習内容
  - ✓3つのレッスンで構成
  - ✓インメモリ・カラム・ストアの設定から
    バッファ・キャッシュとのパフォーマンスの比較まで実施
  - ✓SQL*Plusを使用

インメモリ ハンズオン実習
スキーマの概要



- 本実習では、TPC-Hベンチマークをベースにしたスター・スキーマを使用
  - 「LINEITEM」と「ORDERS」表を1つに結合
- 8GBのスケール・ファクタで構築
- ユーザ：ssb、パスワード：ssb

ハンズオン実習をはじめる前に

実習環境セットアップ

Teratermで実習環境に接続

- ✓ ホスト名： 192.168.56.101
- ✓ ユーザー：oracle、パスワード:oracle

ハンズオン環境に移動

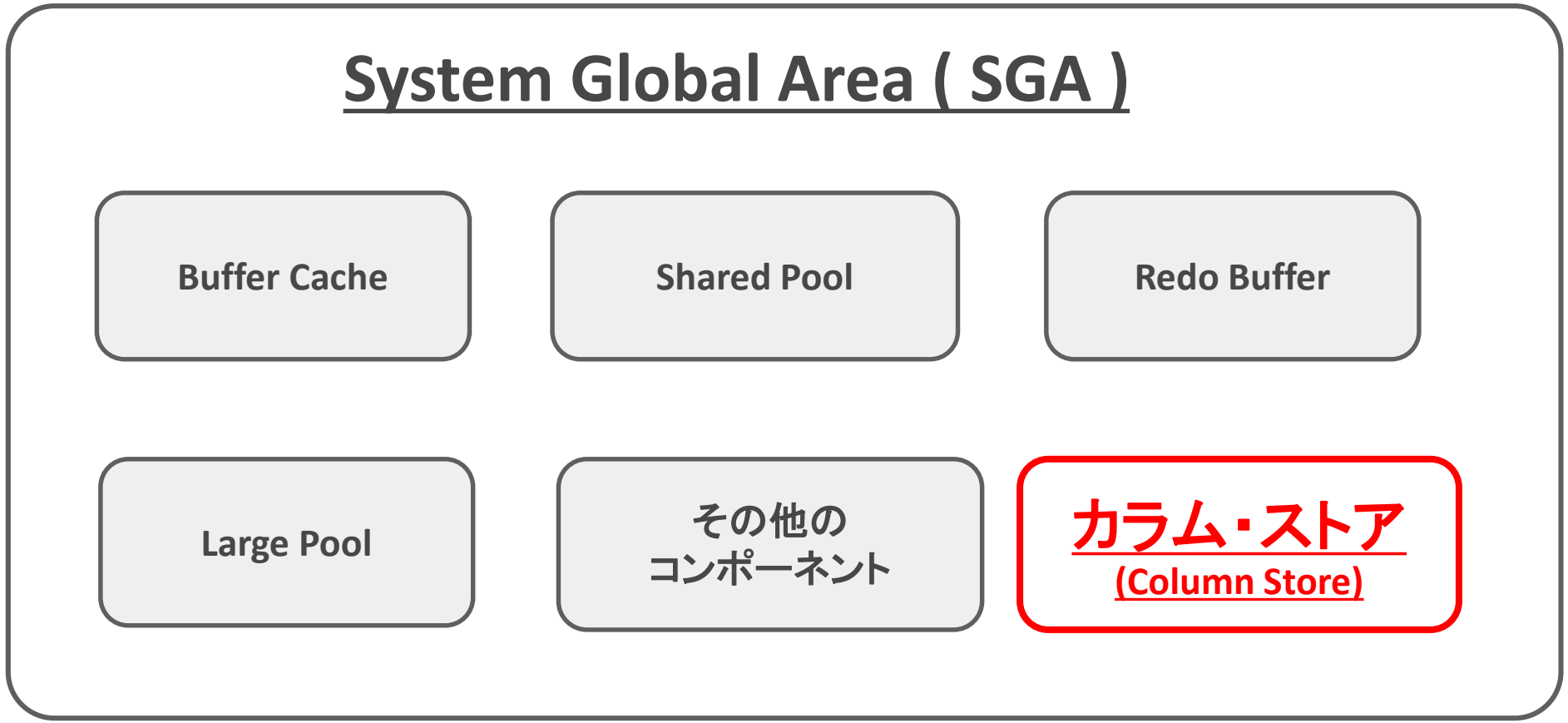
- ✓ "~/InMemory_HandsOn" に移動
- ✓ 最初の演習のスクリプトがある、"lesson1"ディレクトリに移動

```
$ cd ~/InMemory_HandsOn/
$ ls
lesson1/  lesson2/  lesson3/
$ cd lesson1
$ ls
step1.sql   step2.sql   step3.sql   step4.sql   step5.sql   step6.sql   step7.sql   step8.sql   step9.sql
```

Lesson 1
インメモリ カラム・ストアの設定

インメモリ・カラム・ストアの設定

Oracleデータベースのメモリ構造



インメモリ・カラム・ストアの設定
実習環境のメモリー構成

実習環境のメモリー構成

- ✓10GBメモリ搭載
- ✓INMEMORY_SIZE=3.5 G
- ✓MEMORY_MAX_TARGET=8.5 G
- ✓PGA_AGGREGATE_TARGET=1.4 G

＊INMEMORY_SIZEも設定済みです

演習

Lesson1: Step1

- まずはIn-Memory Column Storeに割当てられているメモリー・サイズを確認しましょう。In-Memory Column Store の作成は INMEMORY_SIZE 初期化パラメータを設定するのみで、非常に 簡単に行えることを覚えておいてください。
- SQL*Plusでデータベースに接続します（”sys/Welcome1 as sysdba”で接続する必要があります）。

```
$ sqlplus sys/Welcome1 as sysdba

SQL*Plus: Release 12.1.0.2.0 Production on 木 8月 28 13:58:21 2014

Copyright (c) 1982, 2014, Oracle.  All rights reserved.

Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing options
に接続されました。
SQL>
```

演習

Lesson1: Step1

- In-Memory Column Storeに割当てられているメモリー・サイズの確認には、以下のコマンドを実行します。
  show parameter inmemory
  または、step1.sqlスクリプトを実行します。

```
SQL> @step1.sql
SQL> show parameter inmemory
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
inmemory_clause_default              string
inmemory_force                       string      DEFAULT
inmemory_max_populate_servers        integer     1
inmemory_query                       string      ENABLE
inmemory_size                        big integer 3500M
inmemory_trickle_repopulate_servers_percent integer     1
optimizer_inmemory_aware             boolean     TRUE
```

演習

Lesson1: Step1

- 現在**3.5GB**がIn-Memory Column Storeに割当てられていることが 確認できます。また、他にもインメモリ関連の初期化パラメータ（inmemory_clause_default、inmemory_force、inmemory_max_populate_servers、inmemory_query、inmemory_trickle_repopulate_servers_percent、optimizer_inmemory_aware）が確認できます。
- **inmemory_max_populate_servers**は、ポピュレート用のワーカー・プロセスの最大数を指定します。デフォルトではCPU_COUNT x 0.5の値に設定されます。
- **inmemory_trickle_repopulate_servers_percent**は、トリクル・リポピュレーションを行うワーカー・プロセスの割合(%)を指定します。この値はinmemory_max_populate_serversで指定したプロセス数に対する割合になります。
- **optimizer_inmemory_aware**は、オプティマイザはインメモリ・カラム・ストアを考慮するかどうかを指定します。そのため、コスト計算や実行計画について影響があります。このパラメータにFALSEを指定した場合、例えば、インメモリ・カラム・ストアにあるデータについて考慮せずにSQLを実行します。

演習

Lesson1: Step2

- それでは、In-Memory Column Storeがどのようなものなのか、実際に確認してみましょう。
- 2つの新しいV$ビュー（V$IM_SEGMENTSとV$IM_USER_SEGMENTS）が追加されました。これらのビューでは、In-Memory Column Storeにどのオブジェクトがロードされているかを確認できます。どのような情報（列）がビューに含まれるか、以下のコマンドで確認します。

```
desc   v$im_segments
```

または、step2.sqlのSQLスクリプトを実行します。

演習

Lesson1: Step2

```
SQL> @step2.sql
接続されました。
SQL> -- This command displays the column in the view V$IM_SEGMENTS
SQL> desc V$IM_SEGMENTS
 名前                                      NULL?    型
 ----------------------------------------- -------- ----------------------
 OWNER                                              VARCHAR2(128)
 SEGMENT_NAME                                       VARCHAR2(128)
 PARTITION_NAME                                     VARCHAR2(128)
 SEGMENT_TYPE                                       VARCHAR2(18)
 TABLESPACE_NAME                                    VARCHAR2(30)
 INMEMORY_SIZE                                      NUMBER
 BYTES                                              NUMBER
 BYTES_NOT_POPULATED                                NUMBER
 POPULATE_STATUS                                    VARCHAR2(9)
 INMEMORY_PRIORITY                                  VARCHAR2(8)
 INMEMORY_DISTRIBUTE                                VARCHAR2(15)
 INMEMORY_DUPLICATE                                 VARCHAR2(13)
 INMEMORY_COMPRESSION                               VARCHAR2(17)
 CON_ID                                             NUMBER
```

- 列名からも分かるように、このビューはIn-Memory Column Storeにどのオブジェクトがあるかを表示します。

演習

Lesson1: Step3

- どのオブジェクトがIn-Memory Column Storeにあるかを確認するには、以下のようにV$IM_SEGMENTS ビューを検索します。
  SELECT v.owner, v.segment_name name, v.populate_status status
  FROM v$im_segments;
  または、SQLスクリプトstep3.sqlを実行します。

```
SQL> @step3
接続されました。
SQL>
SQL> -- This query displays what objects are in the In-Memory Column Store
SQL>
SQL> Select v.owner, v.segment_name name, v.populate_status status From v$im_segments v;

レコードが選択されませんでした。
```

- 結果から、現在はIn-Memory Column Storeにまだオブジェクトがロードされていないことが分かります。

インメモリ・カラム・ストアの設定

3つのステップで設定／構成

1. 初期化パラメータINMEMORY_SIZEの設定

   INMEMORY_SIZE = <インメモリ・カラム・ストアのサイズ>

2. 表にインメモリ（INMEMORY）を指定

   ALTER TABLE <表名> INMEMORY

3. データのポピュレート（メモリーへのロード）

   表にアクセス or **データベース起動時に自動ロード**
   （設定により指定）

バックグラウンド・プロセスが非同期でデータをメモリーにポピュレート

演習

Lesson1: Step4

- In-Memory Column Storeにオブジェクトをロードするには、インメモリ属性をオブジェクトに設定する必要があります。5つの新しい列（INMEMORY、INMEMORY_PRIORITY、INMEMORY_DISTRIBUTE、INMEMORY_COMPRESSION、INMEMORY_DUPLICATE）が*_TABLES（DBA_TABLES、USER_TABLES）ビューに追加され、表の現在のインメモリ属性を表します。（これらの属性の意味は、後述します）
- SQL*Plusでssbユーザーで接続し、以下のようにして現在の属性を確認します。

```
connect ssb/ssb
SELECT table_name, inmemory, inmemory_priority, inmemory_compression,
inmemory_distribute, inmemory_duplicate
FROM   user_tables;
```

- または、SQLスクリプトstep4.sqlを実行します。

演習

Lesson1: Step4

```
SQL> @step4
接続されました。
SQL>
SQL> -- This query allows you to review the current attributes of the tables in SSB scheam
SQL>
SQL> Select table_name, inmemory,
  2       inmemory_priority, inmemory_compression,
  3       inmemory_distribute, inmemory_duplicate
  4  From   user_tables;
TABLE_NAME     INMEMORY   INMEMORY_PRIORITY    INMEMORY_COMPRESSION  INMEMORY_DISTRIBUTE  INMEMORY_DUPLICATE
---------------        ----------------    ----------------------------   ------------------------------------   -------------------------------   -------------------------------
LINEORDER          DISABLED
PART               DISABLED
CUSTOMER           DISABLED
SUPPLIER           DISABLED
DATE_DIM           DISABLED
```

- それぞれの表のINMEMORY属性は、DISABLED に設定されています。これは、In-Memory Column Store について設定がされていないことを意味します。

演習

Lesson1: Step5

- 以下のALTER TABLE文をそれぞれの表に実行することにより、この属性を変更できます。
  ALTER TABLE lineorder INMEMORY;
  または、SQLスクリプト step5.sqlを実行します。

```
SQL> @step5
接続されました。
SQL> -- The following commands enables the inmemory attributes for the tables in the SSB
SQL> Alter table LINEORDER inmemory;
表が変更されました。
SQL> Alter table PART inmemory;
表が変更されました。
SQL> Alter table CUSTOMER inmemory;
表が変更されました。
SQL> Alter table SUPPLIER inmemory;
表が変更されました。
SQL> Alter table DATE_DIM inmemory;
表が変更されました。
```

演習

Lesson1: Step5

- それぞれの表にインメモリ属性を設定することにより、これらの表がIn-Memory Column Storeにロードするべき表であることを指定できます。
- オブジェクトがロードされるタイミングは、**INMEMORY_PRIORITY**属性により設定されます。デフォルトでは、この属性には”NONE”が設定されていて、Oracleが自動的にIn-Memory Column Storeにロードするタイミングを決定することを意味します。これは”On Demand”と呼ばれ、通常は最初に表にアクセスしたタイミングでロードされます。
- また他の方法として、優先度（Priority Level）の設定が可能です。優先度に対応したキューを使用して、In-Memory Column Storeへデータをロードします。キューの優先度はCRITICAL から LOWまであります。これは“At Startup”とも呼ばれ、データベース起動時、もしくは、表定義の変更後に、指定された優先度でIn-Memory Column Storeにデータがロードされます。

演習

Lesson1: Step6

- それでは、以下のSQL文を実行して、ssbスキーマの表の現在の属性を確認してみましょう。

```
connect ssb/ssb
SELECT table_name, inmemory, inmemory_priority, inmemory_compression,
inmemory_distribute, inmemory_duplicate
FROM   user_tables;
```

または、SQLスクリプト step6.sqlを実行します。

演習

Lesson1: Step6

```
SQL> @step6
接続されました。
SQL> -- This query allows you to review the current attributes of the tables in SSB schema
SQL>
SQL> Select table_name, inmemory,
  2       inmemory_priority, inmemory_compression,
  3       inmemory_distribute, inmemory_duplicate
  4  From   user_tables;
TABLE_NAME    INMEMORY   INMEMORY_PRIORITY    INMEMORY_COMPRESSION  INMEMORY_DISTRIBUTE  INMEMORY_DUPLICATE
------------------- ---------------  ------------------------------  ---------------------------------------  ---------------------------------  ------------------------------
LINEORDER       ENABLED                    NONE              FOR QUERY LOW                 AUTO             NO DUPLICATE
PART            ENABLED                    NONE              FOR QUERY LOW                 AUTO             NO DUPLICATE
CUSTOMER        ENABLED                    NONE              FOR QUERY LOW                 AUTO             NO DUPLICATE
SUPPLIER        ENABLED                    NONE              FOR QUERY LOW                 AUTO             NO DUPLICATE
DATE_DIM        ENABLED                    NONE              FOR QUERY LOW                 AUTO             NO DUPLICATE
```

- それぞれの表のインメモリ属性が現在設定されていますが、これらの表はまだIn-Memory Column Storeに格納されていません。重要な点に、「デフォルトでは、In-Memory Column Storeへのロードは、最初にオブジェクトにアクセスされた時に行われる」ということがあります。

演習

Lesson1: Step7

- それでは、5つの表に対して簡単なクエリーを実行し、In-Memory Column Storeにデータをロードしましょう。SQL*Plusから、以下のSELECT count(*)文をそれぞれの表に対して実行してください。

```
connect ssb/ssb
SELECT /*+ full(d)  NO_PARALLEL (d )*/ Count(*)   FROM   date_dim d;
SELECT /*+ full(s)  NO_PARALLEL (s )*/ Count(*)   FROM   supplier s;
SELECT /*+ full(p)  NO_PARALLEL (p )*/ Count(*)   FROM   part p;
SELECT /*+ full(c)  NO_PARALLEL (c )*/ Count(*)   FROM   customer c;
SELECT /*+ full(lo)  NO_PARALLEL (lo )*/ Count(*) FROM   lineorder lo;
```

または、SQLスクリプトstep7.sqlを実行します。

Lesson1: Step7

演習

```
SQL> @step7
SQL> -- The following commands populate the In-Memory Column Store with the tables from SSB schema
SQL> select /*+ full(DATE_DIM) NO_PARALLEL(DATE_DIM) */ count(*) from DATE_DIM;
 COUNT(*)
----------
     2556
SQL> select /*+ full(SUPPLIER) NO_PARALLEL(SUPPLIER) */ count(*) from SUPPLIER;
 COUNT(*)
----------
    16000
SQL> select /*+ full(CUSTOMER) NO_PARALLEL(CUSTOMER) */ count(*) from CUSTOMER;
 COUNT(*)
----------
   240000
SQL> select /*+ full(PART) NO_PARALLEL(PART) */ count(*) from PART;
 COUNT(*)
----------
   800000
SQL> select /*+ full(LINEORDER) NO_PARALLEL(LINEORDER) */ count(*) from LINEORDER;
 COUNT(*)
----------
 47989007
```

Lesson1: Step7
ポピュレート負荷状況の確認例

演習

```
last pid: 16002;  load avg:  13.4,  5.05,  1.93;  up 1+16:13:47
16:46:21
251 processes: 234 sleeping, 17 on cpu
CPU states: 78.6% idle, 19.9% user,  1.5% kernel,  0.0% iowait,  0.0% swap
Kernel: 8297 ctxsw, 7604 trap, 6083 intr, 3746 syscall, 6269 flt
Memory: 128G phys mem, 15G free mem, 4096M total swap, 4096M free swap

   PID USERNAME NLWP PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
 15975 oracle      7  10    0  101G   79G cpu/44   1:41  1.58% ora_w00d_orcl
 15973 oracle      7  10    0  101G   81G cpu/47   1:43  1.58% ora_w004_orcl
 15979 oracle      7  10    0  101G   79G cpu/2    1:33  1.57% ora_w00b_orcl
 15951 oracle      7  10    0  101G   79G cpu/6    2:09  1.57% ora_w009_orcl
 15967 oracle      7  10    0  101G   79G cpu/52   1:53  1.56% ora_w000_orcl
 15965 oracle      7  10    0  101G   79G cpu/26   1:54  1.56% ora_w003_orcl
 15959 oracle      7  10    0  101G   79G cpu/60   2:03  1.56% ora_w002_orcl
 15977 oracle      7  10    0  101G   81G cpu/30   1:36  1.56% ora_w008_orcl
 15963 oracle      7  10    0  101G   79G cpu/21   1:56  1.56% ora_w001_orcl
 15971 oracle      7  10    0  101G   79G cpu/15   1:45  1.56% ora_w006_orcl
 15319 oracle      7  10    0  101G   77G cpu/38   2:10  1.56% ora_w00c_orcl
 15985 oracle      7  10    0  100G   79G cpu/51   1:31  1.19% ora_w007_orcl
 ....
```

1. Teraterm のウィンドウをもう一つ開き接続
2. ‘top’ コマンドを実行
3. ora_w{NNN}_orcl バックグラウンドプロセスを確認
4. データのロードが完了したことは、ora_w**プロセスがtopの画面から見えなくなったことで分かります。
5. または、V$IM_SEGMENTS ビューからロードの進捗を確認することもできます。

演習

Lesson1: Step8

- V$IM_SEGMENTSビューにより、どのオブジェクトがIn-Memory Column Storeにロードされているか確認できることを覚えておいて下さい。SQL*Plusから以下のSQL文を実行します。

```
SELECT v.owner, v.segment_name name,
          v.populate_status status, v.bytes_not_populated
FROM   v$im_segments v;
```

または、SQLスクリプトstep8.sqlを実行します。

演習

Lesson1: Step8

```
SQL> @step8
接続されました。
SQL> -- Query the view v$IM_SEGMENTS to shows what objects are in the column store and how much of the objects
SQL> -- were populated. When the BYTES_NOT_POPULATED is 0, it indicates the entire table was populated.
SQL> SELECT v.owner, v.segment_name name, v.populate_status status, v.bytes_not_populated FROM v$im_segments v;
OWNER                NAME            STATUS  BYTES_NOT_POPULATED
-------------- ----------------- ----------- -------------------------------
SSB                        PART    COMPLETED                             0
SSB                    CUSTOMER    COMPLETED                             0
SSB                   LINEORDER      STARTED                      12485434
SSB                    DATE_DIM    COMPLETED                             0
SSB                    SUPPLIER    COMPLETED                             0
```

- BYTES_NOT_POPULATED列に”0”が表示されている場合は、その表の全てのデータがIn-Memory Column Storeにロードされています。
- しかし、圧縮についてははどうでしょうか？　In-Memory Column Storeに格納されたオブジェクトのサイズは、実際はどのくらいでしょうか？

演習

Lesson1: Step9

- in-Memory Column Storeでの、実際のオブジェクトのサイズは、V$IM_SEGMENTSビューで確認できます。SQL*Plus から以下のSQL文を実行します。

```
SELECT v.owner, v.segment_name,
          v.bytes                orig_size,
          v.inmemory_size        in_mem_size,
          ROUND(v.bytes / v.inmemory_size,2) comp_ratio
FROM   v$im_segments v
ORDER BY 4;
```

または、SQLスクリプトstep9.sqlを実行します。

演習

Lesson1: Step9

```
SQL> @step9
接続されました。
SQL> -- This query compares the actual size of the tables on disk to the size in the
SQL> -- In-Memory Column Store, inorder to calculate the compression ratio
SQL> Select v.segment_name                name,
  2        v.bytes                    orig_size,
  3        v.inmemory_size            in_mem_size ,
  4        ROUND(v.bytes/v.inmemory_size, 2)  comp_ratio
  5  From   v$im_segments v
  6  Order by 4;

NAME                      ORIG_SIZE   IN_MEM_SIZE   COMP_RATIO
------------------------- ----------- ------------- -----------
DATE_DIM                     196608       1179648         .17
SUPPLIER                    2097152       2228224         .94
CUSTOMER                   20971520      17956864        1.17
LINEORDER                2954690560    2499411968        1.18
PART                       58720256      18022400        3.26
```

***NOTE:** このクエリは、ディスクで圧縮されているかどうかを考慮されていません。しかし、このハンズオンでは基本圧縮を使用してデータが圧縮されているため、圧縮率が低く表示されています。例えばLINEORDER表のインメモリ・カラム・ストアでの圧縮率は、実際は2.4倍以上になります。

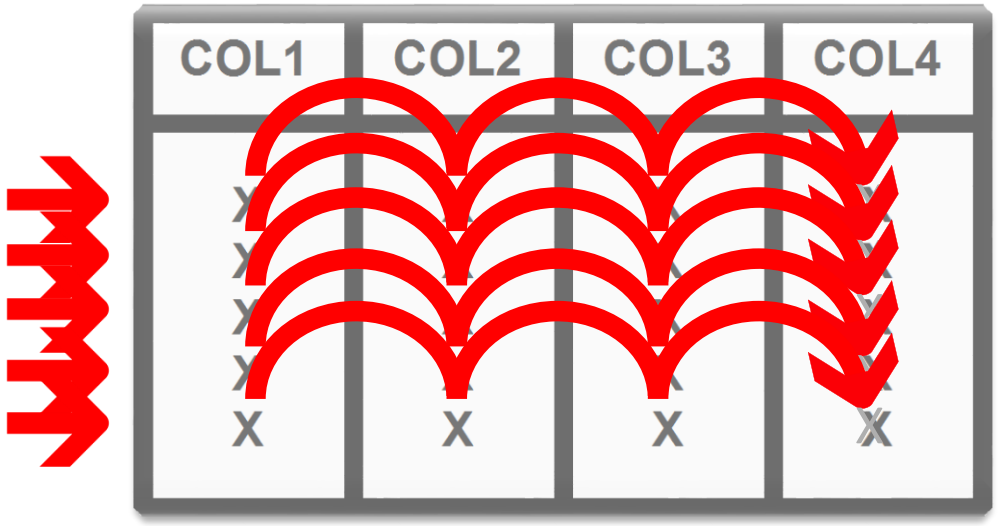
Lesson 1 まとめ

- ✓カラムストアのサイズは「INMEMORY_SIZE」パラメータで制御
  - 利用中のシステムのカラムストアのサイズを確認するには？
- ✓カラムストアに表を追加するには「INMEMORY」属性を設定
  - alter table <表名> inmemory;
- ✓「V$IM_SEGMENTS」を使ってカラムストアの状態を監視
- ✓「V$IM_SEGMENTS」でMEMCOMPRESSの圧縮比を表示
  - 一番大きい圧縮比率は何の表になっていましたか？

Lesson 2

インメモリ表スキャン

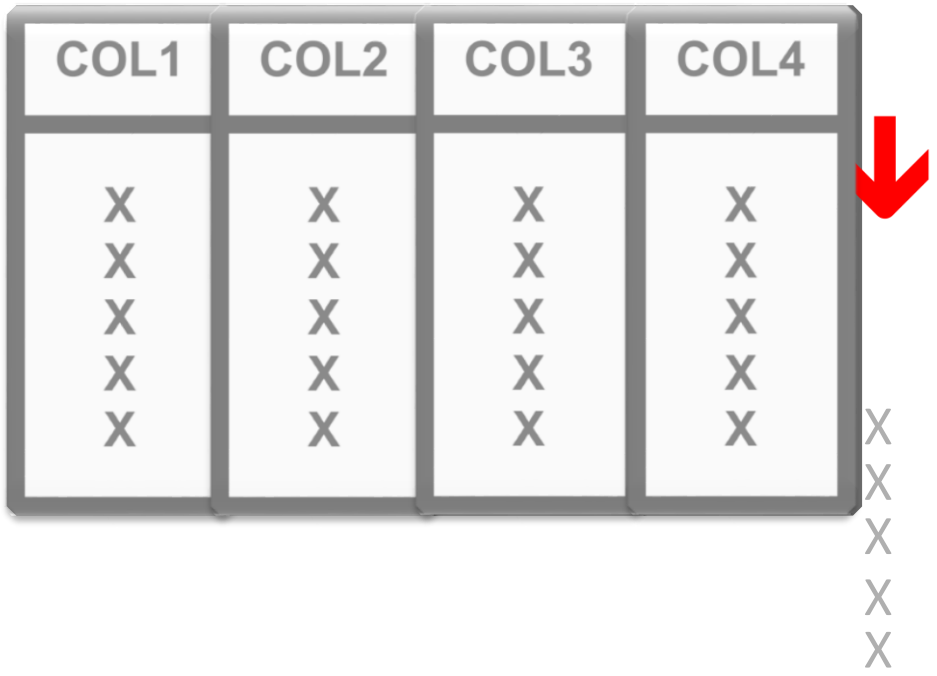
行（ロー）ベースのデータ・アクセス



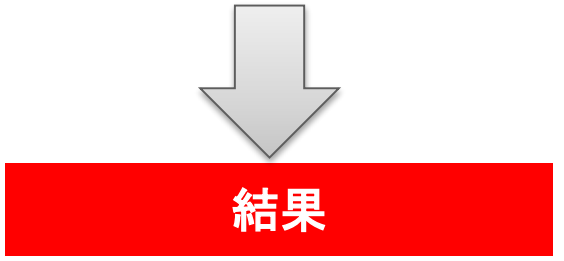
SELECT **COL4**
FROM MYTABLE

結果

カラム・ベースのデータアクセスによる高速化

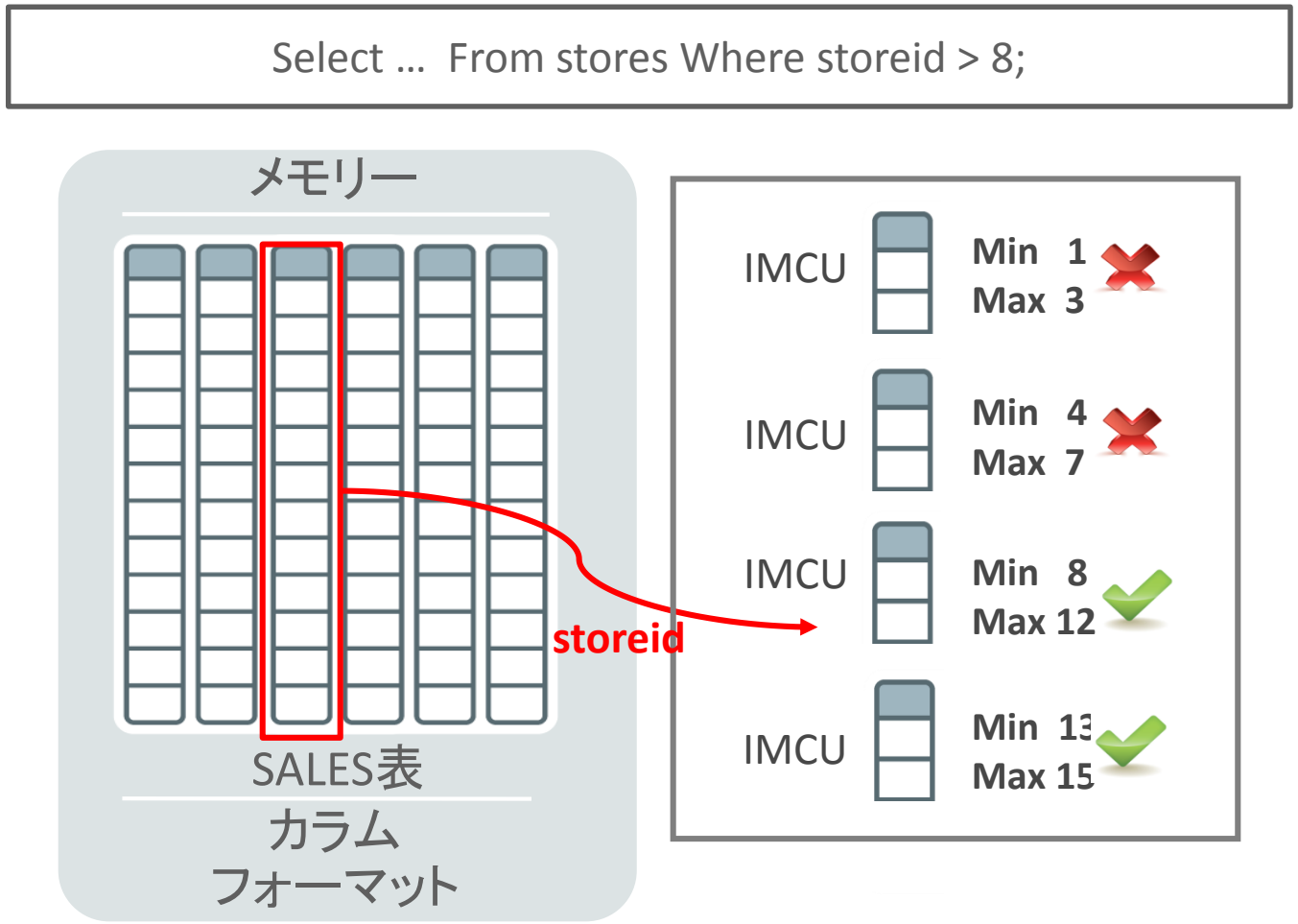


SELECT **COL4**
FROM MYTABLE



インメモリ・ストレージ索引による高速化

- 各カラムは複数のカラム・ユニット(IMCU)で構成される
- 各IMCUで最小値/最大値を自動的に記録
- WHERE句の条件に合致する領域だけを読み込み
- すべての検索でパーティション・プルーニングと同様のパフォーマンスを提供



*1: IMCU - In-Memory Compression Unit

演習

Lesson2: Step1

- In-Memory Column Storeにデータが全てロードされたので、次にそれを使うことで得られるメリットを確認しましょう。このLessonでは、大きいファクト表のLINEORDERに対して、一連のクエリーを実行します。Buffer CacheとColumn Storeの両方に対して実行し、インメモリによるパフォーマンス以上のパフォーマンスが、Column Storeにより得られることを確認します。
- パフォーマンスの比較を行うため、SQL*Plusのセッションから次のコマンドでtimerを設定します。
  set timing ON
- In-Memory Column Storeに対してクエリーを行うため、以下のSQL文を実行します。

```
SELECT /*+ no_parallel */ Max(lo_ordtotalprice) most_expensive_order
FROM   lineorder;
```

演習

Lesson2: Step1

- 同じクエリーをBuffer Cacheに対しても行うため、まずはIn-Memory Column Storeを無効にします。そのためには、INMEMORY_QUERY パラメータにdisableを設定します。
  INMEMORY_QUERY パラメータは、クエリ実行時に、In-Memory Column Store を利用するかどうかを制御するパラメータです。以下のSQL文を実行して、Buffer Cacheに対してクエリーを実行します。(このハンズオンではセッション・パラメータを使用していますが、代わりに”/*+ NO_INMEMORY */ヒントを使用することも可能です)

```
ALTER SESSION set inmemory_query = disable;
SELECT /*+ no_parallel BUFFER CACHE */
       Max(lo_ordtotalprice) most_expensive_order
FROM   lineorder;
```

  または、SQLスクリプトstep1.sqlを実行します。

演習

Lesson2: Step1

```
SQL> @step1
接続されました。
SQL> -- In-Memory Column Store query
SQL> Select /*+ no_parallel */ max(lo_ordtotalprice) most_expensive_order From LINEORDER;
MOST_EXPENSIVE_ORDER
--------------------
           56963813
経過: 00:00:00.01
SQL> -- Buffer Cache query with the column store disables via INMEMORY_QUERY parameter
SQL> alter session set inmemory_query = disable;
セッションが変更されました。
経過: 00:00:00.00
SQL> Select /*+ no_parallel BUFFER CACHE */ max(lo_ordtotalprice) most_expensive_order From LINEORDER;
MOST_EXPENSIVE_ORDER
--------------------
           56963813
経過: 00:00:05.17
SQL>
SQL> alter session set inmemory_query = enable;
セッションが変更されました。
```

演習

Lesson2: Step1

- クエリー実行後に、以下のSQL文でIn-Memory Column Storeを再度有効にするのを忘れないで下さい。

```
ALTER SESSION set inmemory_query = enable;
```

- クエリーが非常に早いことが確認できたと思います。これは完全にメモリーへのクエリーのためです。しかし、In-Memory Column Storeへのクエリーの方が、Buffer Cacheへのクエリーと比べて、非常に早いことが確認できたと思います。これはなぜでしょうか？

  In-Memory Column Storeの場合は一つの列（lo_ordtotalprice）のみを検索します。それに対してRow Store（Buffer Cache）の場合はlo_ordtotalprice列のデータが見つかるまで全ての列を検索します。

演習

Lesson2: Step2

- クエリーがIn-Memory Column Storeの表を使用しているかどうか、どうやって確認できるでしょうか？
- 通常、SQL文がどの様に実行されているかは実行プランで確認します。Step1で実行したSQLの、2つの実行プランを確認するため、SQLスクリプトstep2.sqlを実行します。

```
SQL> @step2
接続されました。
SQL> -- In-Memory Column Store query
SQL> Select /*+ no_parallel */ max(lo_ordtotalprice) most_expensive_order From LINEORDER;

MOST_EXPENSIVE_ORDER
--------------------
          56963813

経過: 00:00:00.01
SQL>
```

演習

Lesson2: Step2

```
SQL> select * from table(dbms_xplan.display_cursor());
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------

SQL_ID  26640ayms972j, child number 0
-------------------------------------
Select /*+ no_parallel */ max(lo_ordtotalprice) most_expensive_order From LINEORDER

Plan hash value: 22267213921
-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |           |       |       |  4359 (100)|          |
|   1 |  SORT AGGREGATE             |           |     1 |     6 |            |          |
|   2 |   TABLE ACCESS INMEMORY FULL| LINEORDER |    47M|   274M|  4359   (12)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Note
-----
   - Degree of Parallelism is 1 because of hint
19行が選択されました。
経過: 00:00:00.02
```

演習

Lesson2: Step2

```
SQL> -- Buffer Cache query with the column store disables via INMEMORY_QUERY parameter
SQL> alter session set inmemory_query = disable;
セッションが変更されました。
経過: 00:00:00.00
SQL> Select /*+ no_parallel BUFFER CACHE */ max(lo_ordtotalprice) most_expensive_order From LINEORDER;
MOST_EXPENSIVE_ORDER
--------------------
            56963813
経過: 00:00:05.19
SQL> select * from table(dbms_xplan.display_cursor());
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------

SQL_ID  7yptr51n9ym7w, child number 0
-------------------------------------
Select /*+ no_parallel BUFFER CACHE */ max(lo_ordtotalprice) most_expensive_order From LINEORDER
Plan hash value: 2267213921
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |       |       | 98174 (100)|          |
|   1 |  SORT AGGREGATE    |           |     1 |     6 |            |          |
|   2 |   TABLE ACCESS FULL| LINEORDER |    47M|   274M| 98174   (1)| 00:00:04 |
-------------------------------------------------------------------------------------------
```

演習

Lesson2: Step2

- 最初のクエリーでは、”IN MEMORY”を含む新しいキーワードが表示され、Buffer Cacheの場合には表示されないことが確認できます。これらのキーワードは、LINEORDER表がIN MEMORYに設定されて、このクエリーがColumn Storeを「使用可能」なことを意味します。「使用可能」というのは、どういう事でしょうか？
これはオブジェクトがIN MEMORYと設定されていても、In-Memory Column Storeを使用しない場合があるためです。ちょうどExadataの環境でのSTORAGEキーワードの使われ方と似ています。
- In-Memory Column Storeが使用されたことを確認するためには、セッション・レベルの統計を確認します。セッション・レベルの統計は、v$mystatおよびv$statnameビューを検索して確認できます。

演習

Lesson2: Step2

- In-Memory Column Store 関連の統計の名前は、ほぼ全て“IM”から始まります。また、以下のSQL文でほぼ全てのIn-Memory Column Store に対するクエリ実行関連の統計を確認できます。

```
column  display_name format a30

SELECT display_name
FROM   v$statname
WHERE  display_name LIKE 'IM scan%';
```

または、SQLスクリプト step2_1.sqlを実行します。

演習

Lesson2: Step2

```
SQL> @step2_1
接続されました。
SQL>
SQL> -- Dispaly the list of IM session statstics
SQL>
SQL> SELECT display_name
  2  FROM   v$statname
  3  WHERE  display_name LIKE 'IM scan%';

DISPLAY_NAME
----------------------------------------------------------------
IM scan CUs rollback
IM scan CUs no rollback
IM scan CUs undo records applied
IM scan CUs cleanout
IM scan CUs no cleanout
IM scan journal cleanout
IM scan journal no cleanout
IM scan journal
IM scan rows journal total
IM scan found invalid smu
IM scan CUs no memcompress
IM scan CUs memcompress for dml
IM scan CUs memcompress for query low
IM scan CUs memcompress for query high
IM scan CUs memcompress for capacity low
IM scan CUs memcompress for capacity high
IM scan segments disk
IM scan bytes in-memory
IM scan bytes uncompressed
IM scan CUs columns accessed
IM scan CUs columns decompressed
IM scan CUs columns theoretical max
```

演習

Lesson2: Step2

IM scan rows
IM scan rows valid
IM scan rows range excluded
IM scan rows discontinuous
IM scan rows excluded
IM scan rows optimized
IM scan rows projected
IM scan rows cache
IM scan blocks cache
IM scan fetches journal
IM scan rows journal
IM scan CUs split pieces
IM scan CUs predicates received
IM scan CUs predicates applied
IM scan CUs predicates optimized

IM scan CUs optimized read
IM scan CUs pruned
IM scan segments minmax eligible
IM scan CUs column not in memory
IM scan CUs invalid
IM scan invalid all blocks
IM scan CUs invalid or missing revert to on disk extent
IM scan CUs failed to reget pin
IM scan CUs invalid (all rows are invalid)

46行が選択されました。

経過: 00:00:00.01

演習

Lesson2: Step2

- 非常に多くの統計があることが確認できます。しかし、全ての統計を把握する必要はありません。確認が必要なのは、クエリーが本当にIn-Memory Column Storeに対して実行されたかどうかを表す以下の統計だけです。

  IM scan CUs columns accessed:クエリーにより、実際にアクセスされた “IMCU”内のColumn(列) の合計数

- 実行したクエリーはLINEORDER表のフル･スキャンを行うため、LINEORDER表の全てのIMCUにアクセスしています。また、アクセス列は1列（lo_ordtotalprice列）のみです。

- セッション統計の値を確認するには、以下のSQL文を実行します。

```
SELECT display_name, value
FROM   v$mystat m,       v$statname n
WHERE  m.statistic# = n.statistic# AND display_name ='IM scan CUs columns accessed';
```

この時点では、この統計値が、”0”であることが確認できると思います。

演習

Lesson2: Step2

- それでは、step 1のクエリーを以下のSQL文で再実行してみましょう。

```
SELECT /*+ no_parallel */ Max(lo_ordtotalprice) most_expensive_order
FROM   lineorder;
```

- 以下のSQL文を実行して、統計を再確認してみましょう。

```
SELECT display_name, value
FROM   v$mystat m,
       v$statname n
WHERE  m.statistic# = n.statistic#
AND    display_name ='IM scan CUs columns accessed';
```

- または、SQLスクリプトstep2_2.sqlを実行します。

演習

Lesson2: Step2

```
SQL> @step2_2
接続されました。
SQL> --  Check the current values for the IM session statstics
SQL> Select display_name, value
  2  From v$mystat m, v$statname n
  3  Where m.STATISTIC#=n.STATISTIC#
  4  And display_name ='IM scan CUs columns accessed';
DISPLAY_NAME                                                     VALUE
---------------------------------------------------------------- ----------
IM scan CUs columns accessed                                              0
経過: 00:00:00.00
SQL> -- In-Memory Column Store query
SQL> Select /*+ no_parallel */ max(lo_ordtotalprice)
most_expensive_order From LINEORDER;
MOST_EXPENSIVE_ORDER
--------------------
            56963813
経過: 00:00:00.01
```

```
SQL> --  Check the current values for the IM session
SQL> --  statstics  again
SQL>
SQL> Select display_name, value
  2  From v$mystat m, v$statname n
  3  Where m.STATISTIC#=n.STATISTIC#
  4  And display_name ='IM scan CUs columns accessed';

DISPLAY_NAME                                                     VALUE
---------------------------------------------------------------- ----------
IM scan CUs columns accessed                                             89

経過: 00:00:00.00
```

- セッション統計により、In-Memory Column Storeには約90のLINEORDER表のCUがあることが確認できます。

演習

Lesson2: Step2

- それでは、以下のSQL文を実行して、Buffer Cacheについても同じstepを実行してみましょう。

```
SELECT display_name, value
FROM   v$mystat m,       v$statname n
WHERE  m.statistic# = n.statistic# AND   display_name ='IM scan CUs columns accessed';

ALTER SESSION set inmemory_query = disable;
SELECT /*+ no_parallel BUFFER CACHE */
      Max(lo_ordtotalprice) most_expensive_order
FROM   lineorder;
ALTER SESSION set inmemory_query = enable;

SELECT display_name, value
FROM   v$mystat m,       v$statname n
WHERE  m.statistic# = n.statistic# AND   display_name ='IM scan CUs columns accessed';
```

- または、SQLスクリプトstep2_3.sqlを実行します。
- 今回は、IM統計が全く増えないことが確認できたと思います。これは、クエリーがIn-Memory Column Storeを使用せず、代わりにBuffer Cacheを使用したためです。

演習

Lesson2: Step3

- 通常、表の特定のデータを検索する場合、Full Table Scanは良い実行プランではありません。しかし、In-Memory Column Storeの表では、この概念は覆されます。このことを、実際にLINEORDER表の特定の注文（order）をorderkeyで検索して確認します。
- 以下のSQL文を実行して、In-Memory Column Storeに対してクエリーを実行します。

```
SELECT /*+ no_parallel */ lo_orderkey, lo_custkey, lo_revenue
FROM   lineorder WHERE  lo_orderkey = 5000000;
```

- 以下のSQL文を実行して、Buffer Cacheに対してクエリーを実行します。

```
ALTER SESSION set inmemory_query = disable;
SELECT /*+ no_parallel BUFFER CACHE */
      lo_orderkey, lo_custkey, lo_revenue FROM   lineorder
WHERE  lo_orderkey = 5000000;
ALTER SESSION set inmemory_query = enable;
```

- または、SQLスクリプトstep3.sqlを実行します。

演習

Lesson2: Step3

```
SQL> @step3
接続されました。
SQL>
SQL> -- In-Memory Column Store query
SQL>
SQL> Select /*+ no_parallel */ lo_orderkey, lo_custkey,
lo_revenue
  2  From      LINEORDER
  3  Where     lo_orderkey = 5000000;

LO_ORDERKEY  LO_CUSTKEY   LO_REVENUE
------------------- ------------------ ------------------
      5000000           38918         2456268

経過: 00:00:00.02
SQL>
```

```
SQL> -- Buffer Cache query with the column store disables via
INMEMORY_QUERY parameter
SQL> alter session set inmemory_query = disable;
セッションが変更されました。
経過: 00:00:00.00
SQL>
SQL> Select /*+ no_parallel BUFFER CACHE */
  2         lo_orderkey, lo_custkey, lo_revenue
  3  From      LINEORDER
  4  Where     lo_orderkey = 5000000;

LO_ORDERKEY  LO_CUSTKEY   LO_REVENUE
------------------- ------------------ ------------------
      5000000           38918         2456268
経過: 00:00:04.07
SQL> alter session set inmemory_query = enable;
セッションが変更されました。
```

演習

Lesson2: Step3

- In-Memory Column Storeにより、今までのBuffer Cacheに比較して大幅なパフォーマンスの向上が得られます。しかし、どのようにして、このような高いパフォーマンスを実現しているのでしょうか？
- In-Memory Column Storeが、それぞれの列のストレージ・インデックスにアクセスしていることを思い出して下さい。これにより、Min/Maxを使用したプルーニングが可能になります。
- WHERE句の条件は、それぞれの列のインメモリ・セグメントにあるMin/Maxの範囲と比較され、もし値がMin/Maxの範囲にない場合、そのセグメントは完全にスキップされます。

演習

Lesson2: Step3

- Min/Maxプルーニングが発生しているかどうかは、IMセッション統計により確認できます。以下のSQL文を実行して確認します。

```
SELECT display_name, value
FROM   v$mystat m,       v$statname n
WHERE  m.statistic# = n.statistic# AND    display_name IN ('IM scan CUs columns accessed',
                  'IM scan segments minmax eligible',                  'IM scan CUs pruned');
```

- 次に、検索を実行します。

```
SELECT /*+ no_parallel */ lo_orderkey, lo_custkey, lo_revenue
FROM   lineorder WHERE  lo_orderkey = 5000000;
```

- そして、セッション統計を再度確認します。

```
SELECT display_name, value
FROM   v$mystat m,       v$statname n
WHERE  m.statistic# = n.statistic# AND    display_name IN ('IM scan CUs columns accessed',
                  'IM scan segments minmax eligible',                  'IM scan CUs pruned');
```

- または、SQLスクリプトstep3_1.sqlを実行します。

演習

Lesson2: Step3

```
SQL> @step3_1
接続されました。
SQL> -- Check the In-Memory storge index statistics before the query
SQL> SELECT display_name, value
  2  FROM   v$mystat m,
  3         v$statname n
  4  WHERE  m.statistic# = n.statistic#
  5  AND    display_name in ('IM scan CUs columns accessed','IM scan segments minmax eligible','IM scan CUs pruned');

DISPLAY_NAME                                                     VALUE
---------------------------------------------------------------- ----------
IM scan CUs columns accessed                                              0
IM scan CUs pruned                                                        0
IM scan segments minmax eligible                                          0

経過: 00:00:00.01
SQL> --  Execute the In-Memory Column Store query
```

演習

Lesson2: Step3

```
SQL> Select     /*+ no_parallel */ lo_orderkey, lo_custkey, lo_revenue
  2  From       LINEORDER
  3  Where      lo_orderkey = 5000000;
LO_ORDERKEY  LO_CUSTKEY   LO_REVENUE
------------------- ------------------ ------------------
      5000000            38918         2456268
経過: 00:00:00.01
SQL> -- Check the In-Memory storge index statistics after the query
SQL> SELECT display_name, value FROM   v$mystat m, v$statname n WHERE  m.statistic# = n.statistic#
  2  AND    display_name in ('IM scan CUs columns accessed','IM scan segments minmax eligible','IM scan CUs pruned');

DISPLAY_NAME                                                     VALUE
---------------------------------------------------------------- ----------
IM scan CUs columns accessed                                         22
IM scan CUs pruned                                                   69
IM scan segments minmax eligible                                     89

経過: 00:00:00.00
```

演習

Lesson2: Step3

- この結果から、Min/Maxプルーニングがいかに効果的に動作するかを確認できたと思います。この統計により、多くのCUがMin/Maxの範囲比較の結果、検索対象外と判断され、プルーニングされている（IM scan CUs pruned）ことが分かります。これは、これらのCUのデータが読み込まれないことを意味します。ほんの少しのCUのみMin/Maxの範囲に含まれ、読み込まれています（IM scan CUs columns accessed）。
このケースでは、89個のCUの内、69個のCUがpruned（検索対象外）とされています。残りの20個(89-69)のCU内のlo_orderkey列が検索されます。“IM scan CUs columns accessed”は、CU内のアクセスされた列の数を示します。検索結果を表示するために、2つの列（lo_custkey列とlo_revenue列）にもアクセスするため、IM scan CUs columns accessed”の値は22となります(20 + 2)。
- この段階で、もしかすると単純な索引をlo_orderkey 列に作成した場合にも、In-Memory Column Storeと同様のパフォーマンスが得られるのでは？と考えるかもしれません。そして、それは正しいです。

ORACLE

演習

Lesson2: Step3

- LINEORDER表には、lo_orderkey列に対して不可視索引が既に作成されています。OPTIMIZER_USE_INVISIBLE_INDEXES パラメータを使用することで、In-Memory Column Storeと索引のパフォーマンスを比較できます。
- SQLスクリプトstep3_2.sql を実行して、索引とColumn Storeのパフォーマンスを比較してみましょう。

```
SQL> @step3_2
接続されました。
SQL> -- Execute the In-Memory query as baseline
SQL> Select /*+ no_parallel Without index */ lo_orderkey, lo_custkey, lo_revenue From  LINEORDER
  3  Where      lo_orderkey = 5000000;
LO_ORDERKEY  LO_CUSTKEY   LO_REVENUE
------------------- ------------------ ------------------
      5000000            38918         2456268
経過: 00:00:00.01
SQL>
```

演習

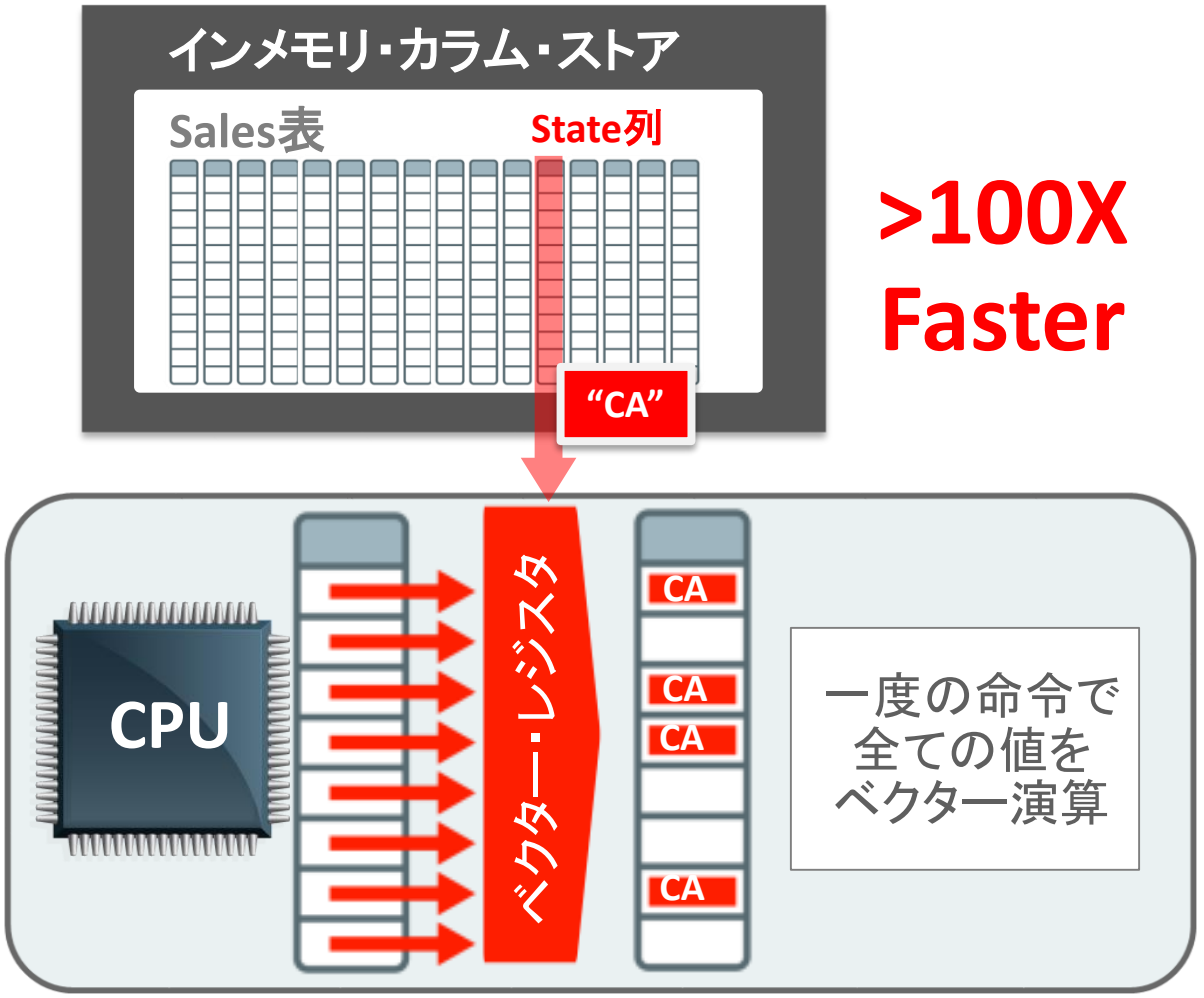
Lesson2: Step3

```
SQL> --  Enable the use of invisible indexes
SQL> alter session set optimizer_use_invisible_indexes=true;
セッションが変更されました。
経過: 00:00:00.01
SQL> -- Execute the query again include a new comment to ensure a hard parse
SQL> Select     /*+ no_parallel With index */ lo_orderkey, lo_custkey, lo_revenue
  2  From       LINEORDER
  3  Where      lo_orderkey = 5000000;

LO_ORDERKEY  LO_CUSTKEY   LO_REVENUE
------------------- ------------------ -------------------
      5000000            38918         2456268

経過: 00:00:00.01
SQL> -- Compare the Elaspe time of the query In-Memory and in the index access
SQL> alter session set optimizer_use_invisible_indexes=false;
セッションが変更されました。
```

SIMDプロセッサによる高速化

- WHERE句条件を圧縮されたデータ形式のまま評価
- 複数の入力を1回のCPU命令で処理可能

例: カリフォルニア州(CA)の全販売実績



>100X Faster

インメモリが利用されたかの判断
実行計画

```
-------------------------------------------------------------------------------------------
| Id  | Operation                          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |            |       |       | 33606 (100)|          |
|   1 |  SORT AGGREGATE                    |            |     1 |     6 |            |          |
|   2 |   TABLE ACCESS INMEMORY FULL| LINEORDER |   300M|  1716M| 33606   (30)| 00:00:02 |
-------------------------------------------------------------------------------------------
```

- インメモリ・カラム・ストアが"使用可能"な場合、実行計画に"TABLE ACCESS INMEMORY FULL"が表示される
- ただし、実際にインメモリ・カラム・ストアにアクセスしていない場合でも、"TABLE ACCESS INMEMORY FULL"は表示される

インメモリが利用されたかの判断

- セッション・レベルの統計（v$mystat）
- インメモリの利用を判断する最適な方法
- インメモリ効果の確認

```
DISPLAY_NAME
----------------------------------------------------------------
IM scan CUs undo records applied
IM scan CUs cleanout
IM scan journal
IM scan rows journal total
IM scan found invalid smu
IM scan CUs no memcompress
IM scan CUs memcompress for dml
IM scan CUs memcompress for query low
IM scan CUs memcompress for query high
IM scan CUs memcompress for capacity low
IM scan CUs memcompress for capacity high
IM scan segments disk
IM scan bytes in-memory
IM scan bytes uncompressed
IM scan CUs columns accessed
IM scan CUs columns decompressed
IM scan CUs columns theoretical max
IM scan rows
. . . .
```

セッション統計は実際にスキャンしたCU(IMCU)数を表示

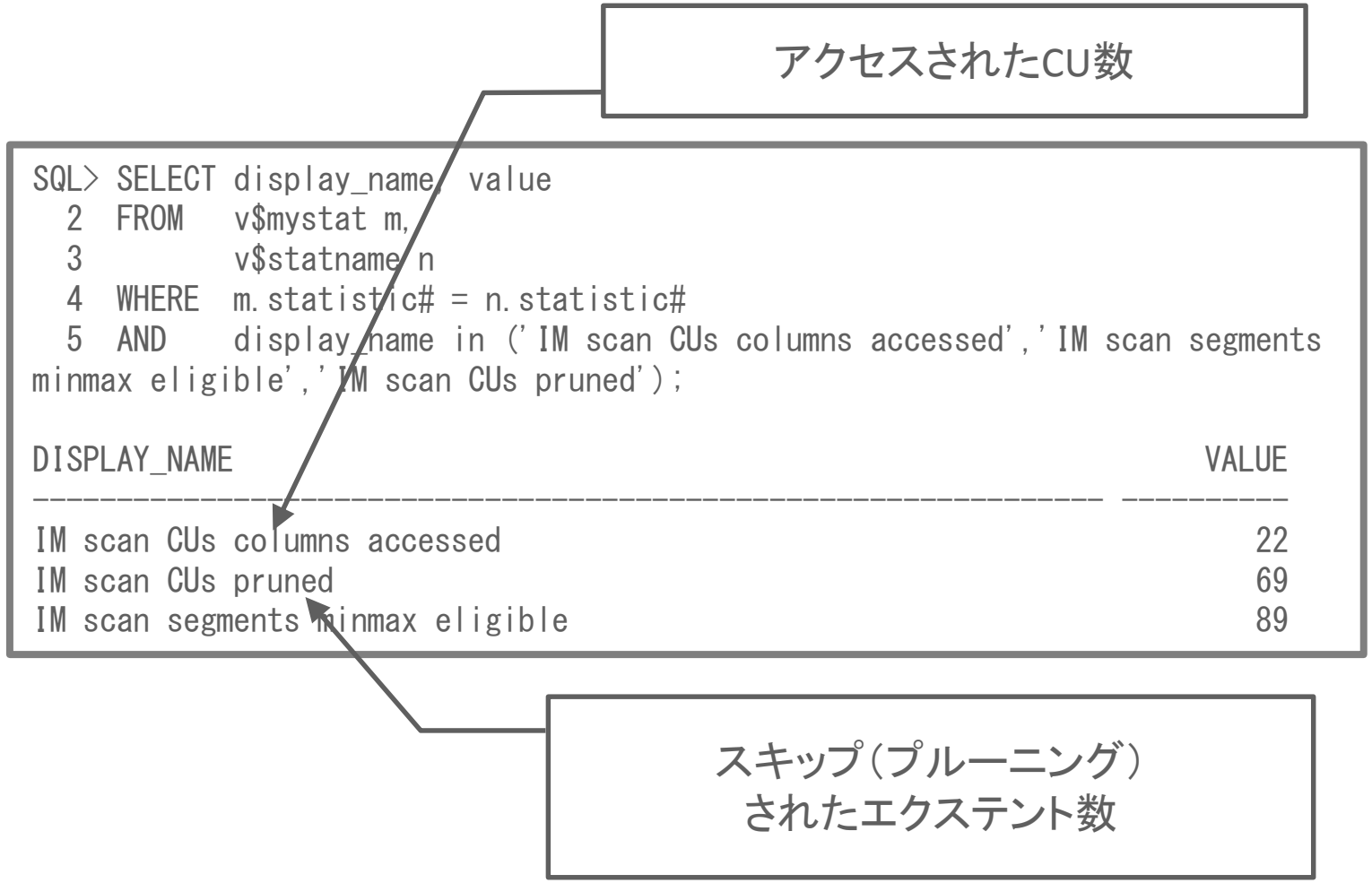
- v$mystatで、カラム・ストアをスキャンした処理量（CU x 列）をチェック
- 実際にスキャンしたCU内の列数：89

```
SQL> Select display_name, value
  2  From v$mystat m, v$statname n
  3  Where m.STATISTIC#=n.STATISTIC#
  4  And display_name ='IM scan CUs columns accessed';

DISPLAY_NAME                                                     VALUE
---------------------------------------------------------------- ----------
IM scan CUs columns accessed                                         89
```

アクセスしたカラムの
実際にスキャンされたCU内の列数

Min/Maxプルーニングを表示するセッション統計

- v$mystatでストレージ索引によりmin/maxプルーニングされたCU数を確認
- "CUs columns accessed"
  実際にアクセスしたCU内の列数
- "CUs pruned"
  プルーニング(スキップ)されたCU数



演習

Lesson2: Step4

- しかし、分析クエリーのWHERE句にたった一つの等価条件しか含まれないというのは、あまり多くありません。もし複数の列の条件があった場合、どうするでしょうか？
  今までは、複数列に対する索引を作成していたのではないでしょうか。ストレージ・インデックスはこれより良いパフォーマンスが得られるでしょうか？
- それでは、特定の注文（order）の項目（Line Items）を検索するようにクエリーを変更して、セッション統計を確認してみましょう。
- 以下のSQL文を実行して、In-Memory Column Storeに対するクエリーを実行します。

```
SELECT display_name, value
FROM   v$mystat m,       v$statname n
WHERE  m.statistic# = n.statistic#
AND    display_name IN ('IM scan CUs columns accessed',
                        'IM scan segments minmax eligible',
                        'IM scan CUs pruned');
```

演習

Lesson2: Step4

```
SELECT /*+ no_parallel */ lo_orderkey, lo_custkey, lo_revenue
FROM   lineorder WHERE  lo_custkey = 5641
AND    lo_shipmode = 'AIR'
AND    lo_orderpriority = '5-LOW';

SELECT display_name, value
FROM   v$mystat m,       v$statname n
WHERE  m.statistic# = n.statistic#
AND    display_name IN ('IM scan CUs columns accessed',
                        'IM scan segments minmax eligible',
                        'IM scan CUs pruned');
```

- または、SQLスクリプトstep4.sqlを実行します。

演習

Lesson2: Step4

- 経過時間（elapsed time）およびセッション統計から、インメモリ・ストレージ・インデックスが使用されていることが確認できます。実際、複数ストレージ・インデックスは同時に使用できます。これは、Oracleデータベースが複数のビットマップ索引を組み合わせて、同時に使用できるのと同様です。
- 残念ながら複数列に対する索引は作成されていないため、Buffer Cacheへの検索はFull Table Scanが実行されます。
  - これがどのような影響があるかを確認するには、SQLスクリプトstep4_1.sqlを実行します。

演習

Lesson2: Step5

- もしWHERE句の条件が等価条件で無い場合はどうでしょうか？それでもMin/Maxプルーニングの効果はあるでしょうか？
  これを確認するためにクエリーを変更し、“15%以上ディスカウントした注文において、どのくらいの収益があったのか？”という検索を実行してみましょう。
- In-Memory Column Storeについて検索するには、以下を実行します。

```
SELECT display_name, value FROM   v$mystat m,
      v$statname n WHERE  m.statistic# = n.statistic#
AND    display_name IN ('IM scan CUs columns accessed',
                        'IM scan segments minmax eligible',
                        'IM scan CUs pruned');

SELECT  /*+ no_parallel */ sum(lo_revenue)
FROM    LINEORDER
WHERE   lo_discount > 15;
```

演習

Lesson2: Step5

```
SELECT display_name, value FROM   v$mystat m, v$statname n
WHERE  m.statistic# = n.statistic# AND    display_name IN
('IM scan CUs columns accessed',     'IM scan segments minmax eligible', 'IM scan CUs pruned');
```

- 同じクエリーをBuffer Cacheに対して実行するため、inmemory_queryパラメータを設定してIn-Memory Column Storeを無効にします。
  以下のSQL文を実行し、Buffer Cacheについてクエリーを実行します。

```
ALTER SESSION set inmemory_query = disable;
SELECT /*+ no_parallel BUFFER CACHE */
      sum(lo_revenue) FROM    LINEORDER
WHERE   lo_discount > 15;
ALTER SESSION set inmemory_query = enable;
```

- または、SQLスクリプトstep5.sqlを実行します。

演習

Lesson2: Step5

- まだIn-Memory Column Storeの方がBuffer Cacheよりパフォーマンスが良いことが分かります。これは、等価条件でない場合でもIn-Memory Column Storeは一つの列（lo_quantity）のみスキャンすれば良いためです。また、インメモリ・ストレージ・インデックスも使用できるためです。
- 実際には15%以上ディスカウントをしている注文は無いため、インメモリ・ストレージ・インデックスによりLINEORDER表の全てのCUを対象外とします。つまり、関連する検索処理を全てスキップできるため、カラム・ストア検索の方が非常に高速となります。

演習

Lesson2: Step6

- それでは今度は、複数の列に対する条件とLINEORDER表へのサブ・クエリーを含んだ、複雑なWHERE句の条件を試してみましょう。今回のクエリーは、“トラックで発送した場合に、どの高額バルク注文が一番利益が少なかったか？”を検索します。
- 以下のSQL文を実行して、In-Memory Column Storeに対してクエリーを実行します。

```
SELECT display_name, value
FROM   v$mystat m,
       v$statname n
WHERE  m.statistic# = n.statistic#
AND    display_name IN ('IM scan CUs columns accessed',
                        'IM scan segments minmax eligible',
                        'IM scan CUs pruned');
```

演習

Lesson2: Step6

```
SELECT /*+ no_parallel */ lo_orderkey, lo_revenue
FROM   lineorder
WHERE  lo_revenue = (SELECT Min(lo_revenue)
                     FROM   lineorder
                     WHERE  lo_supplycost = (SELECT Max(lo_supplycost)
                                             FROM   lineorder
                                             WHERE  lo_quantity > 10)
                     AND lo_shipmode LIKE 'TRUCK%'
                     AND lo_discount BETWEEN 2 AND 5);

SELECT display_name, value FROM   v$mystat m,       v$statname n
WHERE  m.statistic# = n.statistic#
AND    display_name IN ('IM scan CUs columns accessed',
                        'IM scan segments minmax eligible',
                        'IM scan CUs pruned');
```

演習

Lesson2: Step6

- 以下のSQL文を実行して、同じクエリーをBuffer Cacheに対して実行します。

```
ALTER SESSION set inmemory_query = disable;
SELECT /*+ no_parallel BUFFER CACHE */
      lo_orderkey, lo_revenue FROM   lineorder
WHERE  lo_revenue = (SELECT Min(lo_revenue) FROM   lineorder
                            WHERE  lo_supplycost = (SELECT Max(lo_supplycost)
                                                    FROM   lineorder
                                                    WHERE  lo_quantity > 10)
                     AND lo_shipmode LIKE 'TRUCK%'  AND lo_discount BETWEEN 2 AND 5);
ALTER SESSION set inmemory_query = enable;
```

- または、SQLスクリプトstep6.sqlを実行します。

演習

Lesson2: Step6

```
SQL> @step6
接続されました。
SQL> -- Check session statistics before executing the query
[…省略…]
SQL> -- In-Memory Column Store query
SQL> Select /*+ no_parallel */ lo_orderkey, lo_revenue
  2  From   LINEORDER
  3  Where  lo_revenue = (Select min(lo_revenue)
[…省略…]
LO_ORDERKEY  LO_REVENUE
-------------------- ------------------
      30645955          814408
[…省略…]
経過: 00:00:07.00
```

```
SQL> -- Buffer Cache query with the column store disables via a hint
SQL> ALTER SESSION set inmemory_query = disable;
セッションが変更されました。
SQL> SELECT /*+ no_parallel BUFFER CACHE */ lo_orderkey,
lo_revenue
  2  From LINEORDER
  3  Where  lo_revenue = (Select min(lo_revenue)
  4                  From LINEORDER
[…省略…]
LO_ORDERKEY  LO_REVENUE
-------------------- ------------------
      30645955          814408
[…省略…]
経過: 00:00:23.90
```

- 複雑な条件で、かつインメモリ・ストレージ・インデックスが使用されない場合でも、In-Memory Column Storeへのクエリーは非常に速いことが確認できます。これにより、大量行のデータをスキャンする場合には、In-Memory Column Storeが有効なアプローチであることが分かります。

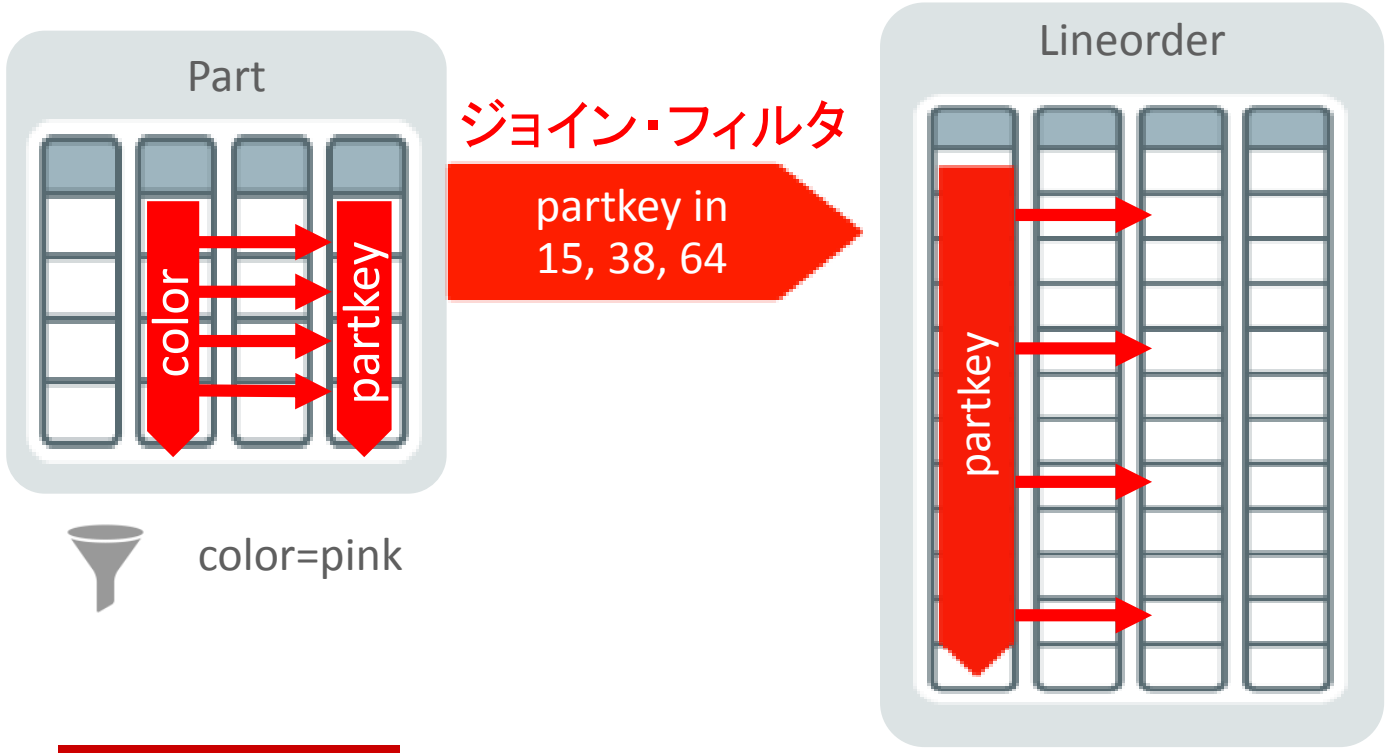
Lesson 2 まとめ

- ✓「INMEMORY_QUERY」を使って、カラムストア検索の利用を制御
  - alter session set INMEMORY_QUERY=disable;
- ✓単純な検索ではカラムストアはバッファ・キャッシュより10倍高速
  - v$mystatでカラムストア内の処理量を確認
- ✓単一行の検索ではカラムストアはバッファ・キャッシュより40倍高速
  - 索引とカラムストアでは同等の速度だが、索引はメンテナンスが必要
- ✓ストレージ索引のMIN/MAXプルーニングはスキャンされるデータ量を大幅に削減することが可能
  - v$mystat からどの程度プルーニングされたか確認可能

Lesson 3
インメモリ結合と集計

ベクター結合（vector join）

```
SELECT … FROM LINEORDER l, PART p
  WHERE  l.lo_partkey = p.p_partkey
    AND  p.p_color= ‘pink’;
```



p.p_color=‘pink'

内部的にLINEORDER表の
カラムスキャンに変換

l.lo_partkey in (15, 38, 64, ….)

ジョイン・フィルタを使ったプラン

ブルーム・フィルタを利用したジョイン・フィルタの作成

```
-----------------------------------------------------------
| 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  |
-----------------------------------------------------------
```

何故この機能が重要か？

結合処理をファクト表のカラム・スキャンに変換することで、処理が高速化される

演習

Lesson3: Step1

- ここまで、一つの表（LINEORDER）へのクエリーを見てきました。次に、結合やパラレル実行にまで範囲を広げてみましょう。このLessonでは、ファクト表（LINEORDER）とディメンション表への単一の結合から 、5つの表を結合するクエリーまで、一連のクエリーを実行します。このクエリーは、Buffer CacheとIn-Memory Column Storeの両方に対して実行し、In-Memory Column Storeが異なる手法でパフォーマンス向上を実現し、カラム・フォーマットのスキャン以上のパフォーマンス向上が得られることを確認します。
- まずは簡単な結合を行うクエリーから始めましょう。ファクト表（LINEORDER）とディメンション表（DATE_DIM）を結合します。これは“What if”クエリーです。特定の日付（Christmas eve 1996）に配送された商品について、特定の範囲のディスカウント率の結果得られた収益の合計を返します。
- 比較を行うため、SQL*PlusでTimerを有効にする必要があります。有効にするには、”set timing on”と入力します。

演習

Lesson3: Step1

- 以下のSQL文を実行して、In-Memory Column Storeに対してクエリーを実行します。

```
SELECT
/*+ no_parallel */
SUM(lo_extendedprice * lo_discount)  revenue
FROM   lineorder l,
          date_dim d
WHERE  l.lo_orderdate = d.d_datekey
AND      l.lo_discount  BETWEEN  2  AND  3
AND      l.lo_quantity < 24
AND      d.d_date='December 24, 1996';
```

演習

Lesson3: Step1

- 以下のSQL文を実行して、Buffer Cacheについてクエリーを実行します。

```
ALTER SESSION set inmemory_query = disable;

SELECT /*+ no_parallel BUFFER CACHE */
      SUM(lo_extendedprice * lo_discount) revenue
FROM   lineorder l, date_dim d
WHERE  l.lo_orderdate = d.d_datekey
AND    l.lo_discount BETWEEN 2 AND 3 AND    l.lo_quantity < 24
AND    d.d_date='December 24, 1996';

ALTER SESSION set inmemory_query = enable;
```

- または、SQLスクリプトstep1.sqlを実行します。

演習

Lesson3: Step1

- In-Memory Column Storeでの結合でも、問題無くBloom Filterも活用できます。Bloom Filterは、結合を変換して、ファクト表の検索（スキャン）処理の一部とする機能です。Hash結合のパフォーマンスを拡張するためにOracle Database 10gで導入された機能で、In-Memory Column Storeのためだけの機能ではありません。しかし、SIMDベクター処理により、非常に効果的にColumnarデータに適用できます。
- 2つの表をHash結合する場合、最初の表（通常は小さい表）がスキャンされ、”WHERE”句の条件を満たす行を使用してHash表が作成されます。Hash表が作成される時に、同時に結合で使用する列に対してBit Vector（Bloom Filter）の作成が行われます。Bit Vectorは、二番目の表のスキャンで追加条件として使用されます。
- Exadataでは、Bloom Filter（Bit Vector）は追加条件として、Storage Cellにオフロードされ、非常に効果的に使用されます。
- Step1のクエリーの全ての実行プランを表示するには、SQLスクリプトstep1_1.sqlを実行してください。

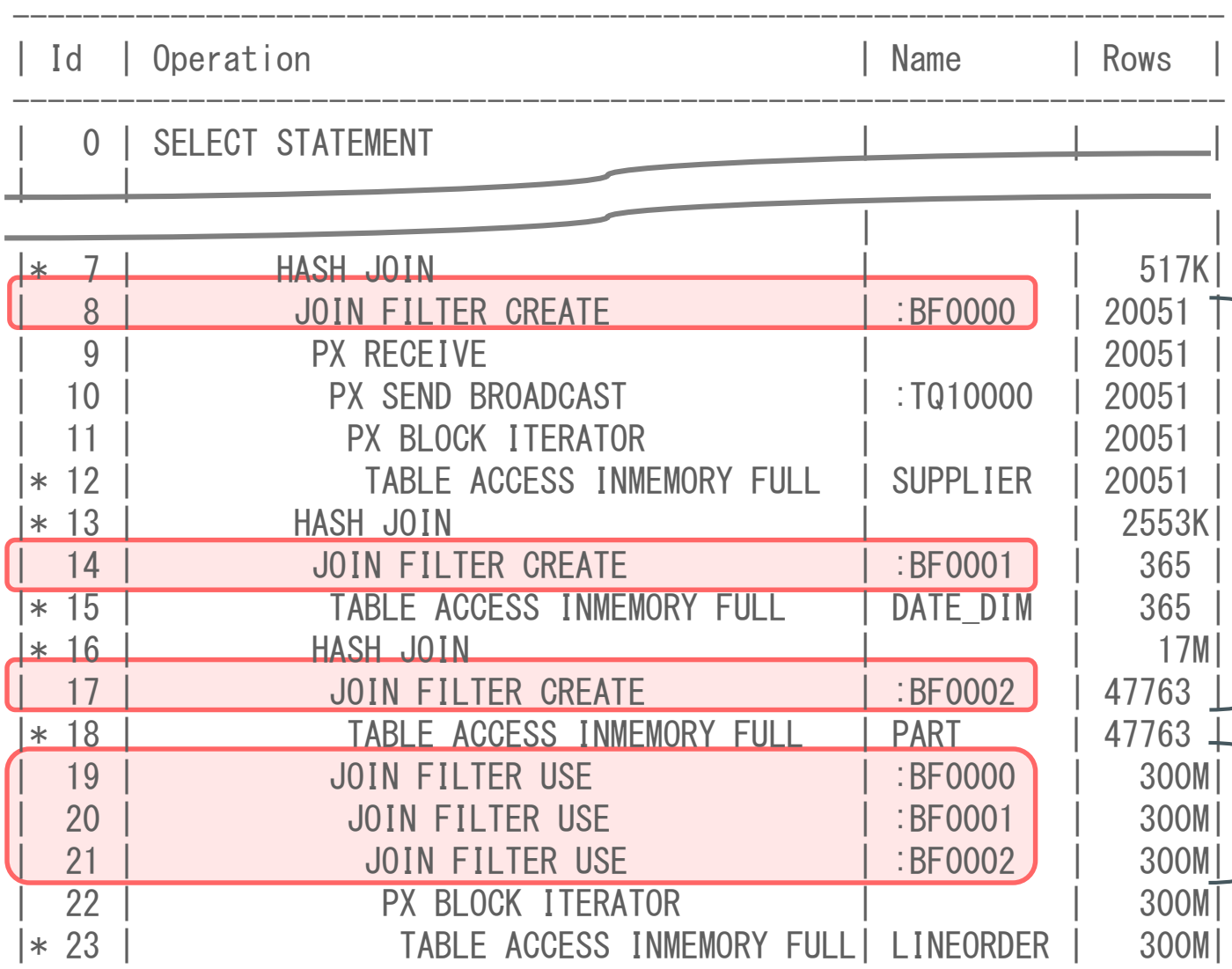
ORACLE

複数のジョイン・フィルタ(ブルーム・フィルタ)

- 最新のオプティマイザでは、ファクト表に対して複数のブルーム・フィルタを適用可能
- ファクト表のスキャンと同時に全ディメンション表を結合

```
SELET   d.d_year, c.c_nation,
        sum(lo_revenue - lo_supplycost) profit
 FROM   LINEORDER l, DATE_DIM d, PART p,
        SUPPLIER s, CUSTOMER C
WHERE   l.lo_orderdate = d.d_datekey
  AND   l.lo_partkey = p.p_partkey
  AND   l.lo_suppkey = s.s_suppkey
  AND   l.lo_custkey = c.c_custkey
  AND   (p.p_mfgr = 'MFGR#12' or p_mfgr='MFGR#2')
  AND   s.s_region = 'AMERICA'
  AND   c.c_region = 'AMERICA'
GROUP BY d.d_year, c.c_nation
ORDER BY d.d_year, c.c_nation;
```

複数のジョイン・フィルタ適用時の実行計画



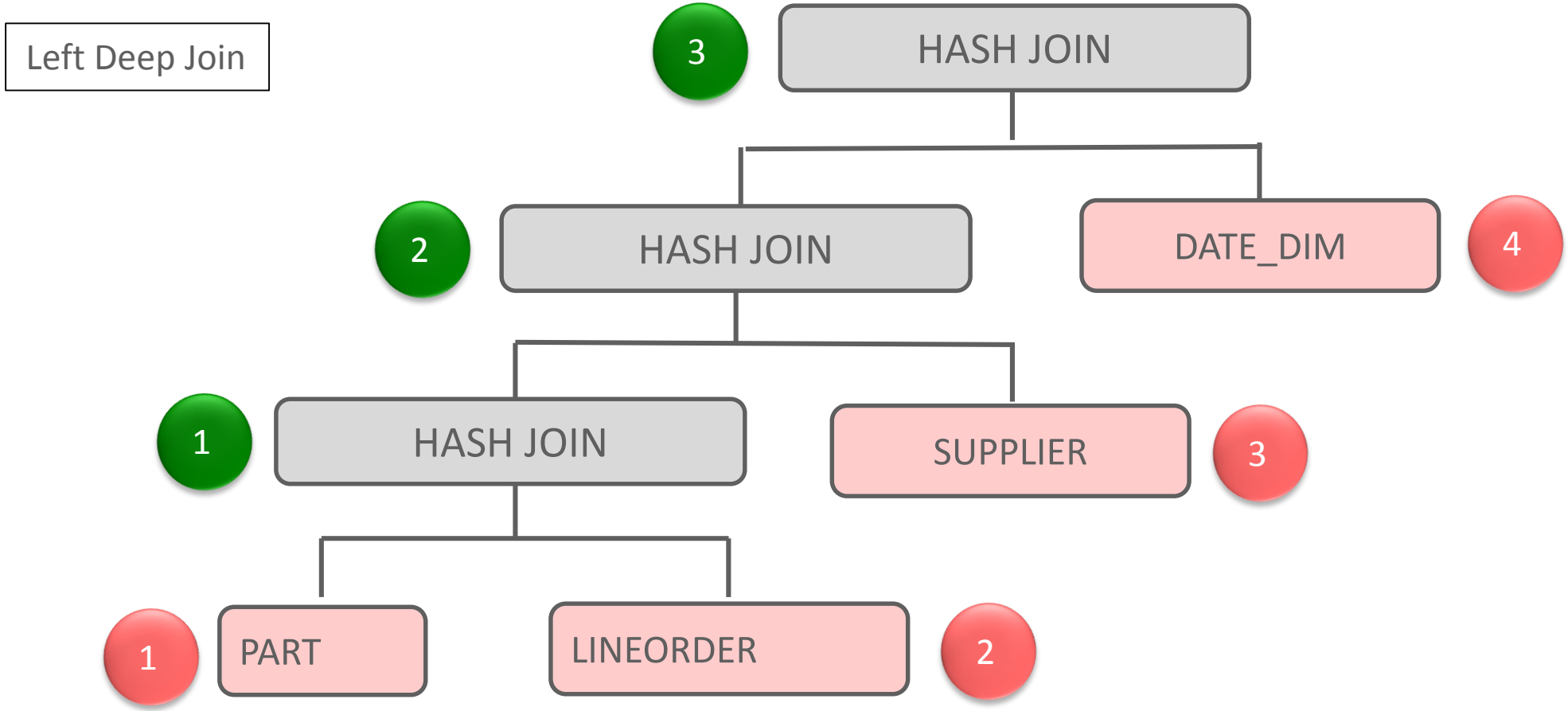
Id	Operation	Name	Rows
0	SELECT STATEMENT		
* 7	HASH JOIN		517K
8	JOIN FILTER CREATE	:BF0000	20051
9	PX RECEIVE		20051
10	PX SEND BROADCAST	:TQ10000	20051
11	PX BLOCK ITERATOR		20051
* 12	TABLE ACCESS INMEMORY FULL	SUPPLIER	20051
* 13	HASH JOIN		2553K
14	JOIN FILTER CREATE	:BF0001	365
* 15	TABLE ACCESS INMEMORY FULL	DATE_DIM	365
* 16	HASH JOIN		17M
17	JOIN FILTER CREATE	:BF0002	47763
* 18	TABLE ACCESS INMEMORY FULL	PART	47763
19	JOIN FILTER USE	:BF0000	300M
20	JOIN FILTER USE	:BF0001	300M
21	JOIN FILTER USE	:BF0002	300M
22	PX BLOCK ITERATOR		300M
* 23	TABLE ACCESS INMEMORY FULL	LINEORDER	300M

ジョイン・フィルタ作成

ジョイン・フィルタ利用

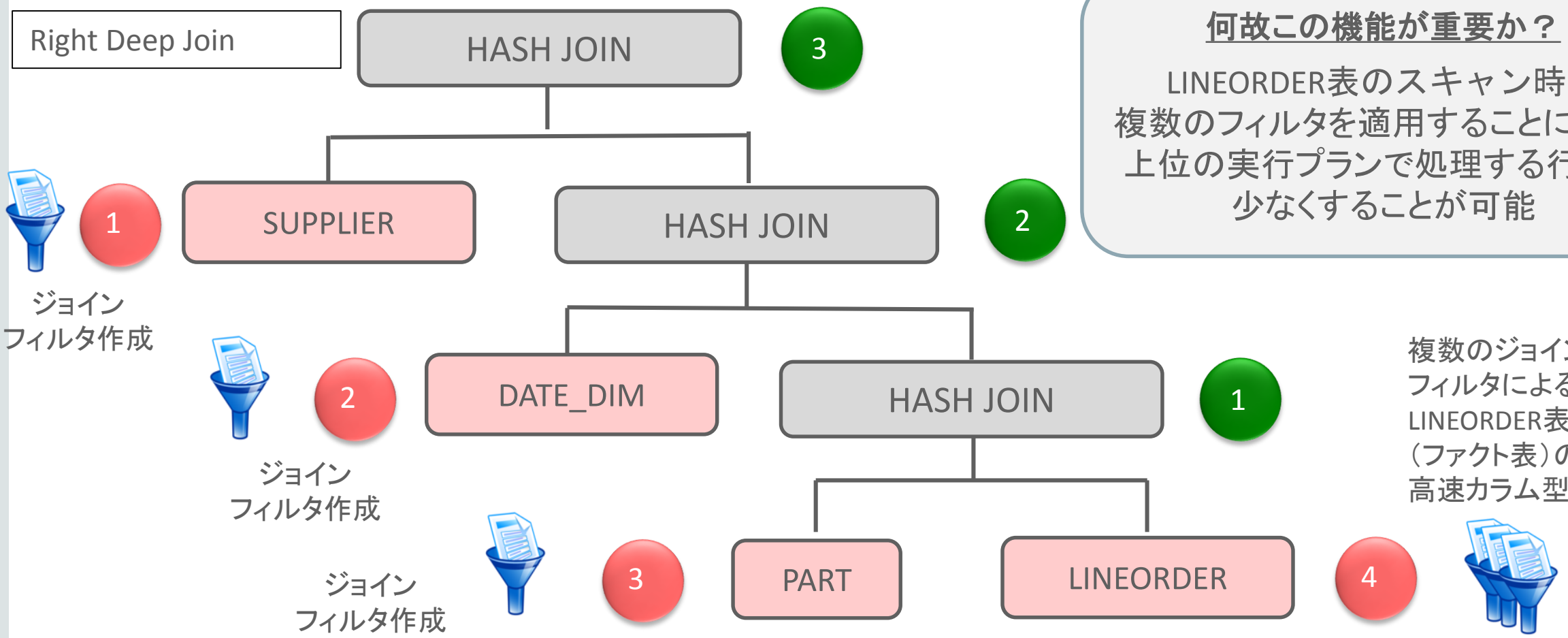
Swap Join Input Optimization

HASH JOINを順番に実施（今までの実行プラン）



Swap Join Input Optimization

複数のジョイン・フィルタを利用してファクト表の高速カラム検索



何故この機能が重要か？

LINEORDER表のスキャン時に
複数のフィルタを適用することにより、
上位の実行プランで処理する行数を
少なくすることが可能

演習

Lesson3: Step2

- それでは、2つの結合と集計をより多くのデータに対して行う、より複雑なクエリーを試してみましょう。このクエリーは、一年の間で一番忙しい月（12月）に、特定の商品により得られた収益を計算します。より多くのデータを扱うため、パラレル実行を使用して処理速度を上げ、かかる時間を短くします。
- 以下のSQL文を実行して、In-Memory Column Storeに対してクエリーを実行します。

```
SELECT /*+ parallel(2) */ p.p_name,  SUM(l.lo_revenue)
FROM   lineorder l,
       date_dim d,
       part p
WHERE  l.lo_orderdate = d.d_datekey
AND    l.lo_partkey = p.p_partkey AND    p.p_name = 'misty gainsboro'
AND    d.d_year = 1992                   AND    d.d_month = 'December'
GROUP  BY p.p_name;
```

演習

Lesson3: Step2

- Buffer Cacheに対してクエリーを実行するには、以下のSQL文を実行します。

```
ALTER SESSION set inmemory_query = disable;

SELECT /*+ parallel(2) BUFFER CACHE */
      p.p_name,  SUM(l.lo_revenue)
FROM   lineorder l, date_dim d,       part p
WHERE  l.lo_orderdate = d.d_datekey AND    l.lo_partkey = p.p_partkey
AND    p.p_name = 'misty gainsboro' AND    d.d_year = 1992
AND    d.d_month = 'December'
GROUP  BY p.p_name;

ALTER SESSION set inmemory_query = enable;
```

- または、SQLスクリプトstep2.sqlを実行します。

演習

Lesson3: Step2

```
SQL> @step2
SQL> connect ssb/ssb
接続されました。
[…省略…]
SQL> set timing on
SQL> -- In-Memory Column Store query
SQL> Select /*+ parallel(2) */ p.p_name, sum(l.lo_revenue)
  2  From       LINEORDER l, DATE_DIM d, PART p
  3  Where      l.lo_orderdate = d.d_datekey
  4  And        l.lo_partkey   = p.p_partkey
  5  And        p.p_name          = 'misty gainsboro'
  6  And        d.d_year = 1992
  7  And        d.d_month = 'December'
  8  Group by p.p_name;
P_NAME                    SUM(L.LO_REVENUE)
------------------------- ------------------------------
misty gainsboro                    254007042
経過: 00:00:00.45
```

```
SQL> -- Buffer Cache query with the column store disables
via the inmemory_query parameter
SQL> ALTER SESSION set inmemory_query = disable;
セッションが変更されました。
SQL> SELECT      /*+ parallel(2) BUFFER Cache */ p.p_name,
sum(l.lo_revenue)
  2  From       LINEORDER l, DATE_DIM d, PART p
  3  Where      l.lo_orderdate = d.d_datekey
  4  AND        l.lo_partkey   = p.p_partkey
  5  And        p.p_name          = 'misty gainsboro'
  6  And        d.d_year = 1992
  7  And        d.d_month = 'December'
  8  Group by p.p_name;

P_NAME                    SUM(L.LO_REVENUE)
------------------------- ------------------------------
misty gainsboro                    254007042
経過: 00:00:04.20
```

演習

Lesson3: Step2

- このSQL文の実行プランを確認すると（step2_2.sqlを実行）、Bloom Filterが作成されていることが確認できます。Bloom Filterはpartkey列、および、datekey列に対して作成されます。Oracleデータベースは、一度のクエリーで複数のBloom Filterを使用することも可能です。適切であれば1つの表のスキャンで複数のBloom Filterが使用されます。

演習

Lesson3: Step3

- Lesson2で確認した表スキャンで得られる利点は、結合でも関連があるのでしょうか？答えは、もちろん“YES”です。
- 結合のためのBloom Filterの作成は、インメモリ・ストレージ・インデックスおよびMin/Maxプルーニングのメリットを受けています。前述のクエリーの統計を確認することによって、どのようにメリットがあるのかを確認できます。
- 前述のクエリー実行時の統計を確認するには、SQLスクリプトstep3.sqlを実行します

```
SQL> @step3
[…省略…]
DISPLAY_NAME                                                     VALUE
---------------------------------------------------------------- ----------
IM scan CUs columns accessed                                        281
IM scan CUs pruned                                                    0
IM scan segments minmax eligible                                      6
```

- Oracle Databaseが、クエリー実行時に少ない数のCompression Unit（CU）しかアクセスしていないことが確認できます。lo_partkeyにBloom Filterを適用し、ストレージ・インデックスを使用することで、結合処理でスキャンするCUの数を減らすことができます。

演習

Lesson3: Step4

- 次に4つ目の表を使用しましょう。今回のクエリーでは、1997年の、特定の地域のサプライヤーについて、商品クラスでの利益の比較を行います。このクエリーは、より多くの行を返します。
- 以下のSQL文を実行して、In-Memory Column Storeに対してクエリーを実行します。

```
SELECT /*+ parallel(2) */ d.d_year, p.p_brand1,SUM(lo_revenue) rev
FROM   lineorder l,
       date_dim d,
       part p,
       supplier s
WHERE  l.lo_orderdate = d.d_datekey AND   l.lo_partkey = p.p_partkey
AND    l.lo_suppkey = s.s_suppkey   AND   p.p_category = 'MFGR#12'
AND    s.s_region   = 'AMERICA'     AND   d.d_year     = 1997
GROUP  BY   d.d_year,p.p_brand1;
```

演習

Lesson3: Step4

- 以下のSQL文を実行して、Buffer Cacheに対してクエリーを実行します。

```
ALTER SESSION set inmemory_query = disable;

SELECT /*+ parallel(2) BUFFER CACHE */
      d.d_year, p.p_brand1,SUM(lo_revenue) rev
FROM   lineorder l,   date_dim d,     part p,     supplier s
WHERE  l.lo_orderdate = d.d_datekey
AND    l.lo_partkey = p.p_partkey  AND    l.lo_suppkey = s.s_suppkey
AND    p.p_category = 'MFGR#12'  AND    s.s_region = 'AMERICA'
AND    d.d_year     = 1997  GROUP  BY d.d_year,p.p_brand1;

ALTER SESSION set inmemory_query = enable;
```

- または、SQLスクリプトstep4.sqlを実行します。

演習

Lesson3: Step4

- やはりIn-Memory Column Storeの方がBuffer Cacheよりもパフォーマンスが良いことが確認できます。しかし、より注目すべきなのは、このクエリーの実行プランです。SQLスクリプトstep4_1.sqlを実行して、実行プランを確認してみましょう。
- このケースでは、3つのJoin Filterが作成され、LINEORDER表のスキャンに使用されたことが確認できます。PART表との結合に1つ、DATE_DIM表との結合に1つ、SUPPLIER表との結合に1つ作成されています。LINEORDER表がSUPPLIER表の前にアクセスされるようになっている場合、どのようにしてOracle Databaseは2つのJoin Filterを適用するのでしょうか？
- オプティマイザは、“SWAP_JOIN_INPUTS”と呼ばれる手法を用いて、左側が深い木構造（Left Deep Tree）から右側が深い木構造（Right Deep Tree）へと変換します。In-Memory Column Storeで”SWAP_JOIN_INPUTS”が使用される利点は、ファクト表をスキャンする前に、複数のBloom Filterを作成できることです。これにより、結合を行う前、ファクト表のスキャン時に行を絞込んで削減できます。
- 複数のBloom Filterによってどのくらい効果があるのかを確認するため、SQLスクリプトstep4_2.sql を実行して、セッション統計を確認してみてください。

ベクター Group By – 集計演算の強化

- 新しいベクターGroup Byはインメモリ配列ベースの集計演算を非常に効率的に実行
- ファクト表のスキャンをしながら同時に集計演算した値をインメモリ配列へ累積演算

```
SELECT  d.d_year, c.c_nation,
        sum(lo_revenue - lo_supplycost) profit
  FROM  LINEORDER l, DATE_DIM d, PART p,
      SUPPLIER s, CUSTOMER C
 WHERE  l.lo_orderdate = d.d_datekey
   AND  l.lo_partkey   = p.p_partkey
   AND  l.lo_suppkey   = s.s_suppkey
   AND  l.lo_custkey   = c.c_custkey
   AND  s.s_region     = 'AMERICA’
   AND  c.c_region     = 'AMERICA’
 GROUP BY d.d_year, c.c_nation
 ORDER BY d.d_year, c.c_nation;
```

インメモリ集計: 詳細イメージ

アニメーション

例）OutletのFootwearの売上をブランド、地域ごとに集計する

ストア表（Stores）

ID	Name	SType	Region	…
1	ABC	Dept Store	APAC	…
2	XYZ	Outlet	NAS	…
3	CCC	Outlet	EMEA	…
…	…	…	…	…

商品表（Products）

ID	Name	Category	Brand	…
1	XS-1234	T-Shirt	PUMA	…
2	AJ-2322	Footwear	FILA	…
3	PW-698	Footwear	NIKE	…
…	…	…	…	…

OutletのFootwearの売上をブランド、地域ごとに集計

売上表（Sales）

ID	Ord Date	Prod_ID	Store_id	Sales
1	2012/7/2	2	5	10
2	2012/7/14	6	4	20
3	2012/9/25	7	1	8
4	2013/4/8	7	2	5
…	…	…	…	…

```
Select st.region, p.brand, sum( s.sales )
From stores st, products p, sales s
Where st.id = s.store_id
And p.id = s.prod_id
And st.Stype = 'Outlet'
And p.category = 'Footwear'
Group by st.region, p.brand
```

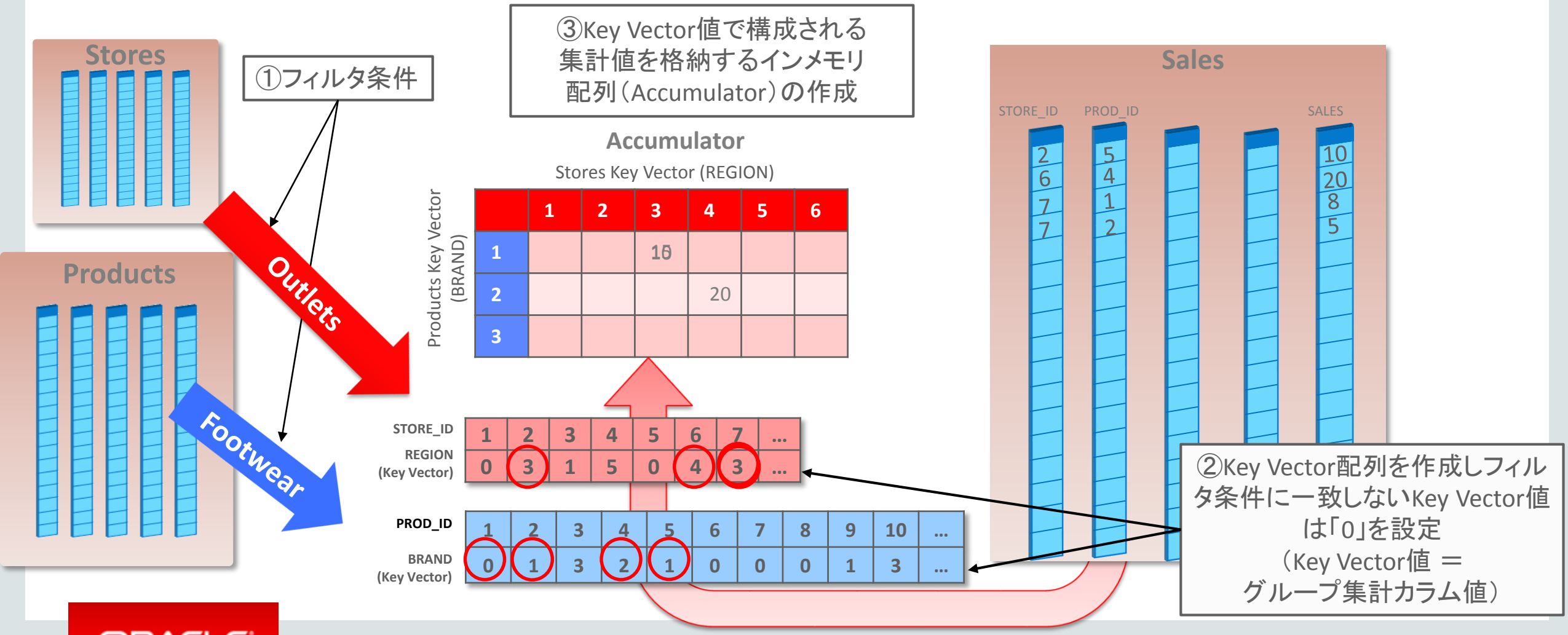
フィルタ条件

キーベクター

インメモリ集計: 詳細

アニメーション

革新的な技術: スタースキーマのジョインと集計処理にメモリ上の配列（インメモリ配列）を使う



Vector Group By – 演習で確認する点

SQL文

```
SELECT  d.d_year, c.c_nation,
        sum(lo_revenue - lo_supplycost) profit
  FROM  LINEORDER l, DATE_DIM d, PART p,    SUPPLIER s, CUSTOMER C
 WHERE  l.lo_orderdate = d.d_datekey
   AND  l.lo_partkey   = p.p_partkey
   AND  l.lo_suppkey   = s.s_suppkey
   AND  l.lo_custkey   = c.c_custkey
   AND  s.s_region     = 'AMERICA’
   AND  c.c_region     = 'AMERICA’
 GROUP BY d.d_year, c.c_nation
 ORDER BY d.d_year, c.c_nation;
```

演習

Lesson3: Step5

- ここまで、結合について確認しました。そして、In-Memory Column Storeが非常に効率的に結合を行うことが確認できました。それでは次に、OLAPでよく使用される” What If”クエリーに目を向けてみましょう。
- このケースで使用するクエリーでは、特定の地域とマニファクチャの１年ごとの収益を、全てのデータを使用して返します。

演習

Lesson3: Step5

- 以下のSQL文を実行して、In-Memory Column Storeに対してクエリーを実行します。

```
SELECT /*+  parallel(2) NO_VECTOR_TRANSFORM */
      d.d_year, c.c_nation, SUM(lo_revenue - lo_supplycost) profit
FROM   lineorder l,
           date_dim d,
           part p,
           supplier s,
           customer c
WHERE  l.lo_orderdate = d.d_datekey AND   l.lo_partkey = p.p_partkey
AND      l.lo_suppkey = s.s_suppkey    AND   l.lo_custkey = c.c_custkey
AND      s.s_region = 'AMERICA'          AND   c.c_region = 'AMERICA'
GROUP  BY d.d_year, c.c_nation
ORDER  BY d.d_year, c.c_nation;
```

演習

Lesson3: Step5

- 以下のSQL文を実行して、Buffer Cacheに対してクエリーを実行します。

```
ALTER SESSION set inmemory_query = disable;

SELECT /*+ parallel(2) NO_VECTOR_TRANSFORM BUFFER CACHE */
      d.d_year, c.c_nation, SUM(lo_revenue - lo_supplycost) profit
FROM   lineorder l,  date_dim d,  part p,   supplier s,   customer c
WHERE  l.lo_orderdate = d.d_datekey AND    l.lo_partkey = p.p_partkey
AND      l.lo_suppkey = s.s_suppkey    AND    l.lo_custkey = c.c_custkey
AND      s.s_region = 'AMERICA'          AND    c.c_region = 'AMERICA'
GROUP  BY d.d_year, c.c_nation          ORDER  BY d.d_year, c.c_nation;

ALTER SESSION set inmemory_query = enable;
```

- または、SQLスクリプトstep5.sqlを実行します。

演習

Lesson3: Step5

- クエリーはより複雑になり、もはやスキャンではありません。結合やソート、データの集計に、より多くの時間がかかります。実行プランを確認してみましょう。
- 実行プランを確認するには、SQLスクリプトstep5_2.sqlを実行します。
- 次にStep6に進み、スキャン後の操作のパフォーマンスを、他の実行プランにより、どのように向上させるのか確認しましょう。

演習

Lesson3: Step6

- 実行プランのスキャン（scan）の上部にある操作をスピード アップするため、Oracleは新たに” Vector Group By”というオプティマイザ変換を導入しました。
- この変換は、2つのフェーズで構成されている処理で、スター型変換と似ています。
- 最初のフェーズではディメンション表をスキャンします。その際、WHERE句の条件が使用されます。このスキャンで得られた行データを使用して、”Key Vector”と呼ばれる新しいデータ構造が作成されます。このKey VectorはBloom Filterに似ていて、結合条件をファクト表スキャン時の追加フィルタ条件として使用します。しかしそれだけではなく、”GROUP BY”や集計処理についても、ファクト表をスキャンした後から行うのではなく、スキャンと同時に行います。
- 2つめのフェーズでは、ファクト表のスキャンで得られた行を、ディメンション表のスキャン時に作成した一時表と結合します。この一時表には、ディメンション表の列（SELECTの対象の列）の情報が含まれています。この2つのフェーズの組合せにより、複雑な集計処理を含む、複数の表の結合処理の効率を大幅に向上します。

演習

Lesson3: Step6

- 両方のフェーズとも、クエリーの実行プランで確認できます。
- 新しい“Vector Group By” の動作を確認するため、この処理を有効化してクエリーを再実行します。Step5のクエリーを、Vector Group Byを使用して実行するため、SQLスクリプトstep6.sqlを実行します。

```
SQL> @step6
接続されました。
SQL> -- In-Memory Column Store query with In-Memory Aggregation enabled
[…省略…]
```

- また、実行プランでVector Group Byが使用されていることを確認するために、SQLスクリプトstep6_2.sqlを実行します。

```
SQL> @step6_2.sql
SQL> connect ssb/ssb
接続されました。
[…続く…]
```

演習

Lesson3: Step6

```
[…省略…]
-----------------------------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name        | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |             |       |       |       | 28571 (100)|          |        |      |            |
|   1 |  TEMP TABLE TRANSFORMATION         |             |       |       |       |            |          |        |      |            |
|   2 |   LOAD AS SELECT                   |             |       |       |       |            |          |        |      |            |
|   3 |    PX COORDINATOR                  |             |       |       |       |            |          |        |      |            |
|   4 |     PX SEND QC (RANDOM)            | :TQ10001    |     7 |    98 |       |     8  (25)| 00:00:01 | Q1,01  | P->S | QC (RAND)  |
|   5 |      BUFFER SORT                   |             |     7 |    98 |       |     8  (25)| 00:00:01 | Q1,01  | PCWP |            |
|   6 |       VECTOR GROUP BY              |             |     7 |    98 |       |     8  (25)| 00:00:01 | Q1,01  | PCWP |            |
|   7 |        KEY VECTOR CREATE BUFFERED  | :KV0000     |  2556 | 35784 |       |     8  (25)| 00:00:01 | Q1,01  | PCWP |            |
|   8 |         PX RECEIVE                 |             |  2556 | 25560 |       |     7  (15)| 00:00:01 | Q1,01  | PCWP |            |
|   9 |          PX SEND HASH              | :TQ10000    |  2556 | 25560 |       |     7  (15)| 00:00:01 | Q1,00  | P->P | HASH       |
|  10 |           PX BLOCK ITERATOR        |             |  2556 | 25560 |       |     7  (15)| 00:00:01 | Q1,00  | PCWC |            |
|* 11 |            TABLE ACCESS INMEMORY FULL | DATE_DIM |  2556 | 25560 |       |     7  (15)| 00:00:01 | Q1,00  | PCWP |            |
|  12 |   LOAD AS SELECT                   |             |       |       |       |            |          |        |      |            |
|  13 |    PX COORDINATOR                  |             |       |       |       |            |          |        |      |            |
|  14 |     PX SEND QC (RANDOM)            | :TQ20001    |     1 |    10 |       |   154  (55)| 00:00:01 | Q2,01  | P->S | QC (RAND)  |
|  15 |      HASH GROUP BY                 |             |     1 |    10 |   13M |   154  (55)| 00:00:01 | Q2,01  | PCWP |            |
|  16 |       PX RECEIVE                   |             |     1 |    10 |       |   154  (55)| 00:00:01 | Q2,01  | PCWP |            |
|  17 |        PX SEND HASH                | :TQ20000    |     1 |    10 |       |   154  (55)| 00:00:01 | Q2,00  | P->P | HASH       |
|  18 |         VECTOR GROUP BY            |             |     1 |    10 |   13M |   154  (55)| 00:00:01 | Q2,00  | PCWP |            |
|  19 |          HASH GROUP BY             |             |     1 |    10 |       |   154  (55)| 00:00:01 | Q2,00  | PCWP |            |
|  20 |           KEY VECTOR CREATE BUFFERED | :KV0001   |  1200K|   11M |       |   149  (54)| 00:00:01 | Q2,00  | PCWC |            |
|  21 |            PX BLOCK ITERATOR       |             |  1200K|  7031K|       |    94  (26)| 00:00:01 | Q2,00  | PCWC |            |
|* 22 |             TABLE ACCESS INMEMORY FULL | PART    |  1200K|  7031K|       |    94  (26)| 00:00:01 | Q2,00  | PCWP |            |
[…続く…]
```

演習

Lesson3: Step6

[…続き…]

23	LOAD AS SELECT										
24	PX COORDINATOR										
25	PX SEND QC (RANDOM)	:TQ30001	1	22		17 (30)	00:00:01	Q3,01	P->S	QC (RAND)	
26	HASH GROUP BY		1	22		17 (30)	00:00:01	Q3,01	PCWP		
27	PX RECEIVE		1	22		17 (30)	00:00:01	Q3,01	PCWP		
28	PX SEND HASH	:TQ30000	1	22		17 (30)	00:00:01	Q3,00	P->P	HASH	
29	VECTOR GROUP BY		1	22		17 (30)	00:00:01	Q3,00	PCWP		
30	HASH GROUP BY		1	22		17 (30)	00:00:01	Q3,00	PCWP		
31	KEY VECTOR CREATE BUFFERED	:KV0002	20051	430K		16 (25)	00:00:01	Q3,00	PCWC		
32	PX BLOCK ITERATOR		20051	352K		15 (20)	00:00:01	Q3,00	PCWC		
* 33	TABLE ACCESS INMEMORY FULL	SUPPLIER	20051	352K		15 (20)	00:00:01	Q3,00	PCWP		
34	LOAD AS SELECT										
35	PX COORDINATOR										
36	PX SEND QC (RANDOM)	:TQ40001	25	975		165 (38)	00:00:01	Q4,01	P->S	QC (RAND)	
37	BUFFER SORT		25	975		165 (38)	00:00:01	Q4,01	PCWP		
38	VECTOR GROUP BY		25	975	13M	165 (38)	00:00:01	Q4,01	PCWP		
39	KEY VECTOR CREATE BUFFERED	:KV0003	299K	11M		164 (38)	00:00:01	Q4,01	PCWP		
40	PX RECEIVE		299K	9M		150 (32)	00:00:01	Q4,01	PCWP		
41	PX SEND HASH	:TQ40000	299K	9M		150 (32)	00:00:01	Q4,00	P->P	HASH	
42	PX BLOCK ITERATOR		299K	9M		150 (32)	00:00:01	Q4,00	PCWC		
* 43	TABLE ACCESS INMEMORY FULL	CUSTOMER	299K	9M		150 (32)	00:00:01	Q4,00	PCWP		

[…続く…]

Lesson3: Step6

演習

44	PX COORDINATOR									
45	PX SEND QC (ORDER)	:TQ50004	62	9300		28228 (54)	00:00:02	Q5,04	P->S	QC (ORDER)
46	SORT GROUP BY		62	9300		28228 (54)	00:00:02	Q5,04	PCWP	
47	PX RECEIVE		62	9300		28228 (54)	00:00:02	Q5,04	PCWP	
48	PX SEND RANGE	:TQ50003	62	9300		28228 (54)	00:00:02	Q5,03	P->P	RANGE
49	HASH GROUP BY		62	9300		28228 (54)	00:00:02	Q5,03	PCWP	
* 50	HASH JOIN		62	9300		28227 (54)	00:00:02	Q5,03	PCWP	
* 51	HASH JOIN		62	7130		28225 (54)	00:00:02	Q5,03	PCWP	
* 52	HASH JOIN		62	6510		28223 (54)	00:00:02	Q5,03	PCWP	
* 53	HASH JOIN		62	5394		28221 (54)	00:00:02	Q5,03	PCWP	
54	TABLE ACCESS FULL	SYS_TEMP_0FD9D6922_2518F0	1	6		2 (0)	00:00:01	Q5,03	PCWP	
55	VIEW	VW_VT_80F21617	62	5022		28218 (54)	00:00:02	Q5,03	PCWP	
56	HASH GROUP BY		62	3038		28218 (54)	00:00:02	Q5,03	PCWP	
57	PX RECEIVE		62	3038		28218 (54)	00:00:02	Q5,03	PCWP	
58	PX SEND HASH	:TQ50000	62	3038		28218 (54)	00:00:02	Q5,00	P->P	HASH
59	VECTOR GROUP BY		62	3038		28218 (54)	00:00:02	Q5,00	PCWP	
60	HASH GROUP BY		62	3038		28218 (54)	00:00:02	Q5,00	PCWP	
61	KEY VECTOR USE	:KV0000	19M	930M		26608 (51)	00:00:02	Q5,00	PCWC	
62	KEY VECTOR USE	:KV0001	19M	854M		26604 (51)	00:00:02	Q5,00	PCWC	
63	KEY VECTOR USE	:KV0003	19M	778M		26600 (51)	00:00:02	Q5,00	PCWC	
64	KEY VECTOR USE	:KV0002	60M	2146M		26596 (51)	00:00:02	Q5,00	PCWC	
65	PX BLOCK ITERATOR		300M	9441M		26183 (50)	00:00:02	Q5,00	PCWC	
* 66	TABLE ACCESS INMEMORY FULL	LINEORDER	300M	9441M		26183 (50)	00:00:02	Q5,00	PCWP	
67	TABLE ACCESS FULL	SYS_TEMP_0FD9D6923_2518F0	1	18		2 (0)	00:00:01	Q5,03	PCWP	
68	PX RECEIVE		7	70		2 (0)	00:00:01	Q5,03	PCWP	
69	PX SEND BROADCAST	:TQ50001	7	70		2 (0)	00:00:01	Q5,01	P->P	BROADCAST
70	PX BLOCK ITERATOR		7	70		2 (0)	00:00:01	Q5,01	PCWC	
* 71	TABLE ACCESS FULL	SYS_TEMP_0FD9D6921_2518F0	7	70		2 (0)	00:00:01	Q5,01	PCWP	
72	PX RECEIVE		25	875		2 (0)	00:00:01	Q5,03	PCWP	
73	PX SEND BROADCAST	:TQ50002	25	875		2 (0)	00:00:01	Q5,02	P->P	BROADCAST
74	PX BLOCK ITERATOR		25	875		2 (0)	00:00:01	Q5,02	PCWC	
* 75	TABLE ACCESS FULL	SYS_TEMP_0FD9D6924_2518F0	25	875		2 (0)	00:00:01	Q5,02	PCWP	

演習

Lesson3: Step6

```
[...続き...]
Predicate Information (identified by operation id):
---------------------------------------------------
 11 - inmemory(:Z>=:Z AND :Z<=:Z)
[...省略...]
 66 - inmemory(:Z>=:Z AND :Z<=:Z AND
  (SYS_OP_KEY_VECTOR_FILTER("L"."LO_SUPPKEY",:KV0002) AND
   SYS_OP_KEY_VECTOR_FILTER("L"."LO_CUSTKEY",:KV0003) AND
         SYS_OP_KEY_VECTOR_FILTER("L"."LO_PARTKEY",:KV0001) AND
   SYS_OP_KEY_VECTOR_FILTER("L"."LO_ORDERDATE",:KV0000)))
     filter((SYS_OP_KEY_VECTOR_FILTER("L"."LO_SUPPKEY",:KV0002) AND
   SYS_OP_KEY_VECTOR_FILTER("L"."LO_CUSTKEY",:KV0003) AND
         SYS_OP_KEY_VECTOR_FILTER("L"."LO_PARTKEY",:KV0001) AND
   SYS_OP_KEY_VECTOR_FILTER("L"."LO_ORDERDATE",:KV0000)))
 [...省略...]

Note
-----
   - Degree of Parallelism is 2 because of hint
   - vector transformation used for this statement
```

- LINEORDER表に複数のBloom Filterを適用するかわりに、”KEY VECTOR”が作成され、使用されていることが確認できます。また、KEY VECTORの作成の際に新しいVector Group Byが使用されていることも確認できます。

演習

Lesson3: Step7

- Vector Group By の動作をもう一つ確認しましょう。今回は、クエリーを拡張し、全てのカスタマーに対する6年間のビジネスで得られる収益を見ます。
- また、今回はIn-Memory Column StoreとBuffer Cacheのパフォーマンスの差を比較するかわりに、Vector Group Byの使用/非使用でパフォーマンスを比較します。
- さらに少し複雑になったクエリーと新しい比較の結果を確認するため、SQLスクリプトstep7.sqlを実行します。

```
SQL> @step7
接続されました。
SQL> -- In-Memory Column Store query with vector group by
[…以下省略…]
```

- 次に、SQLスクリプトstep7_2.sqlを実行することで、Vector Group Byの実行プランを確認できます。

Vector Group by – Step 1

- 各ディメンション表のスキャンとキー・ベクターの作成
- キー・ベクターはジョインを実行するために利用され「VECTOR GROUP BY」を可能にします

```
-------------------------------------------------------------------------------------
| Id  | Operation                                | Name                               |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                                    |
|   1 |  TEMP TABLE TRANSFORMATION               |                                    |
|   2 |   LOAD AS SELECT                         |                                    |
|   3 |    PX COORDINATOR                        |                                    |
|   4 |     PX SEND QC (RANDOM)                  | :TQ10001                           |
|   5 |      BUFFER SORT                         |                                    |
|   6 |       VECTOR GROUP BY                    |                                    |
|   7 |        KEY VECTOR CREATE BUFFERED        | :KV0000                            |
|   8 |         PX RECEIVE                       |                                    |
|   9 |          PX SEND HASH                    | :TQ10000                           |
|  10 |           PX BLOCK ITERATOR              |                                    |
|* 11 |            TABLE ACCESS INMEMORY FULL    | DATE_DIM                           |
|  12 |   LOAD AS SELECT                         |                                    |
|  13 |    PX COORDINATOR                        |                                    |
|  14 |     PX SEND QC (RANDOM)                  | :TQ20001                           |
|  15 |      HASH GROUP BY                       |                                    |
|  16 |       PX RECEIVE                         |                                    |
|  17 |        PX SEND HASH                      | :TQ20000                           |
|  18 |         VECTOR GROUP BY                  |                                    |
|  19 |          HASH GROUP BY                   |                                    |
|  20 |           KEY VECTOR CREATE BUFFERED     | :KV0001                            |
|  21 |            PX BLOCK ITERATOR             |                                    |
|* 22 |             TABLE ACCESS INMEMORY FULL   | PART                               |
```

Vector Group by – Step 2

- ディメンション表から、演算で利用するカラムで構成されるテンポラリ表を作成(TEMP TABLE TRANSFORMATION)
- このテンポラリ表により後の再ジョインをより効果的に実行することが可能

```
-------------------------------------------------------------------------------
| Id  | Operation                          | Name                   |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                        |
|   1 |  TEMP TABLE TRANSFORMATION         |                        |
|   2 |   LOAD AS SELECT                   |                        |
|   3 |    PX COORDINATOR                  |                        |
|   4 |     PX SEND QC (RANDOM)            | :TQ10001               |
|   5 |      BUFFER SORT                   |                        |
|   6 |       VECTOR GROUP BY              |                        |
|   7 |        KEY VECTOR CREATE BUFFERED  | :KV0000                |
|   8 |         PX RECEIVE                 |                        |
|   9 |          PX SEND HASH              | :TQ10000               |
|  10 |           PX BLOCK ITERATOR        |                        |
|* 11 |            TABLE ACCESS INMEMORY FULL | DATE_DIM            |
|  12 |   LOAD AS SELECT                   |                        |
|  13 |    PX COORDINATOR                  |                        |
|  14 |     PX SEND QC (RANDOM)            | :TQ20001               |
|  15 |      HASH GROUP BY                 |                        |
|  16 |       PX RECEIVE                   |                        |
|  17 |        PX SEND HASH                | :TQ20000               |
|  18 |         VECTOR GROUP BY            |                        |
|  19 |          HASH GROUP BY             |                        |
|  20 |           KEY VECTOR CREATE BUFFERED | :KV0001              |
|  21 |            PX BLOCK ITERATOR       |                        |
|* 22 |             TABLE ACCESS INMEMORY FULL | PART               |
```

Vector Group by – Step 3

- キー・ベクターを使って LINEORDER表をスキャン
- スキャン結果を集計対象カラムで作成されたテンポラリ表と結合

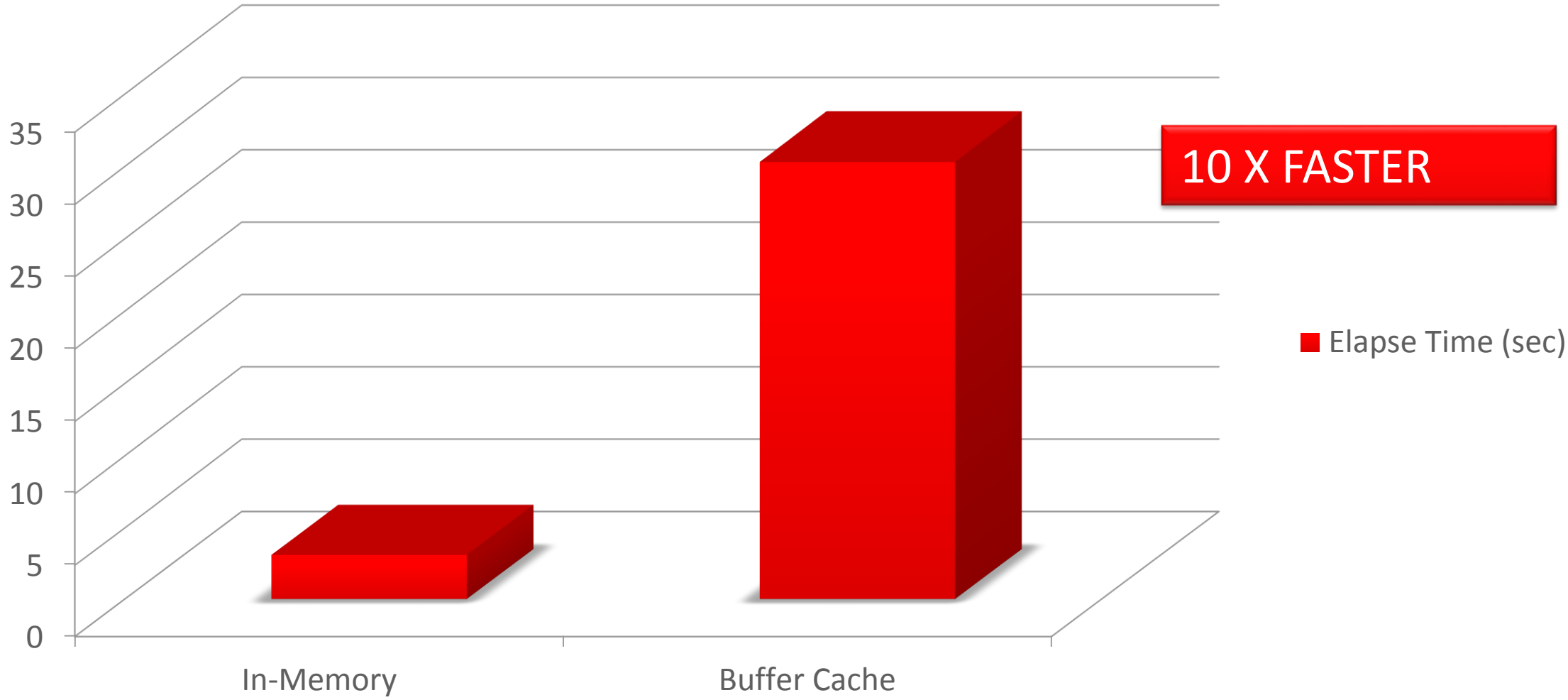
```
|* 50 |          HASH JOIN                          |                          |
|  51 |           PX RECEIVE                        |                          |
|  52 |            PX SEND BROADCAST                | :TQ50000                 |
|  53 |             PX BLOCK ITERATOR               |                          |
|* 54 |              TABLE ACCESS FULL              | SYS_TEMP_0FD9D6932_2518F0 |
|* 55 |           HASH JOIN                         |                          |
|  56 |            PX RECEIVE                       |                          |
|  57 |             PX SEND BROADCAST               | :TQ50001                 |
|  58 |              PX BLOCK ITERATOR              |                          |
|* 59 |               TABLE ACCESS FULL             | SYS_TEMP_0FD9D692F_2518F0 |
|* 60 |           HASH JOIN                         |                          |
|  61 |            TABLE ACCESS FULL                | SYS_TEMP_0FD9D6931_2518F0 |
|* 62 |            HASH JOIN                        |                          |
|  63 |             TABLE ACCESS FULL               | SYS_TEMP_0FD9D6930_2518F0 |
|  64 |             VIEW                            | VW_VT_80F21617           |
|  65 |              HASH GROUP BY                  |                          |
|  66 |               PX RECEIVE                    |                          |
|  67 |                PX SEND HASH                 | :TQ50002                 |
|  68 |                 VECTOR GROUP BY             |                          |
|  69 |                  HASH GROUP BY              |                          |
|  70 |                   KEY VECTOR USE            | :KV0000                  |
|  71 |                    KEY VECTOR USE           | :KV0001                  |
|  72 |                     KEY VECTOR USE          | :KV0003                  |
|  73 |                      KEY VECTOR USE         | :KV0002                  |
|  74 |                       PX BLOCK ITERATOR     |                          |
|* 75 |                        TABLE ACCESS INMEMORY FULL| LINEORDER            |
-------------------------------------------------------------------------------
```

Lesson 3 まとめ

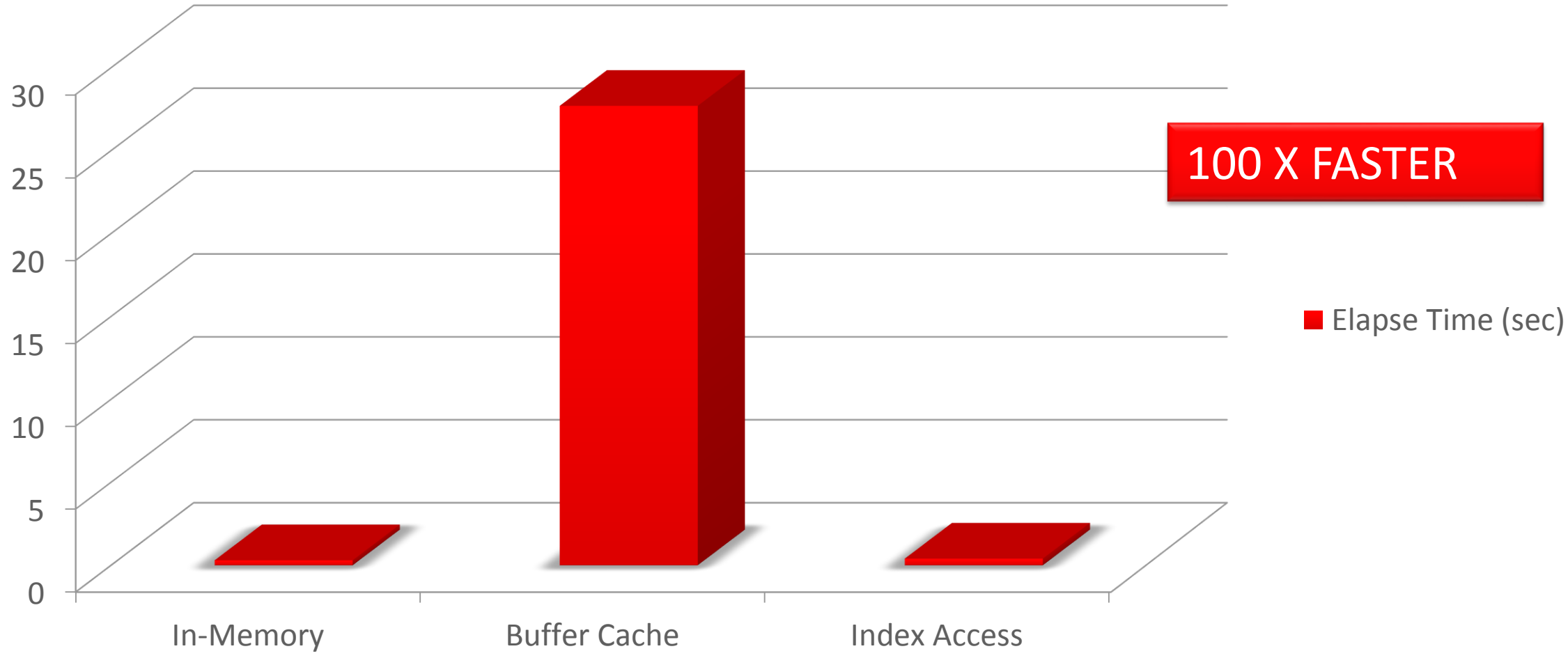
- ✓ ベクター結合（ブルーム・フィルタ）は、表の結合処理を
  ファクト表スキャン時のフィルタ処理に変換し、処理を高速化する
- ✓ ベクターGroup Byは Group Byで処理される行数が非常に
  多い場合に利用される
- ✓ ベクターGroup Byは ファクト表のスキャンと同時に集計演算が可能

Oracle Database In-Memory Recap

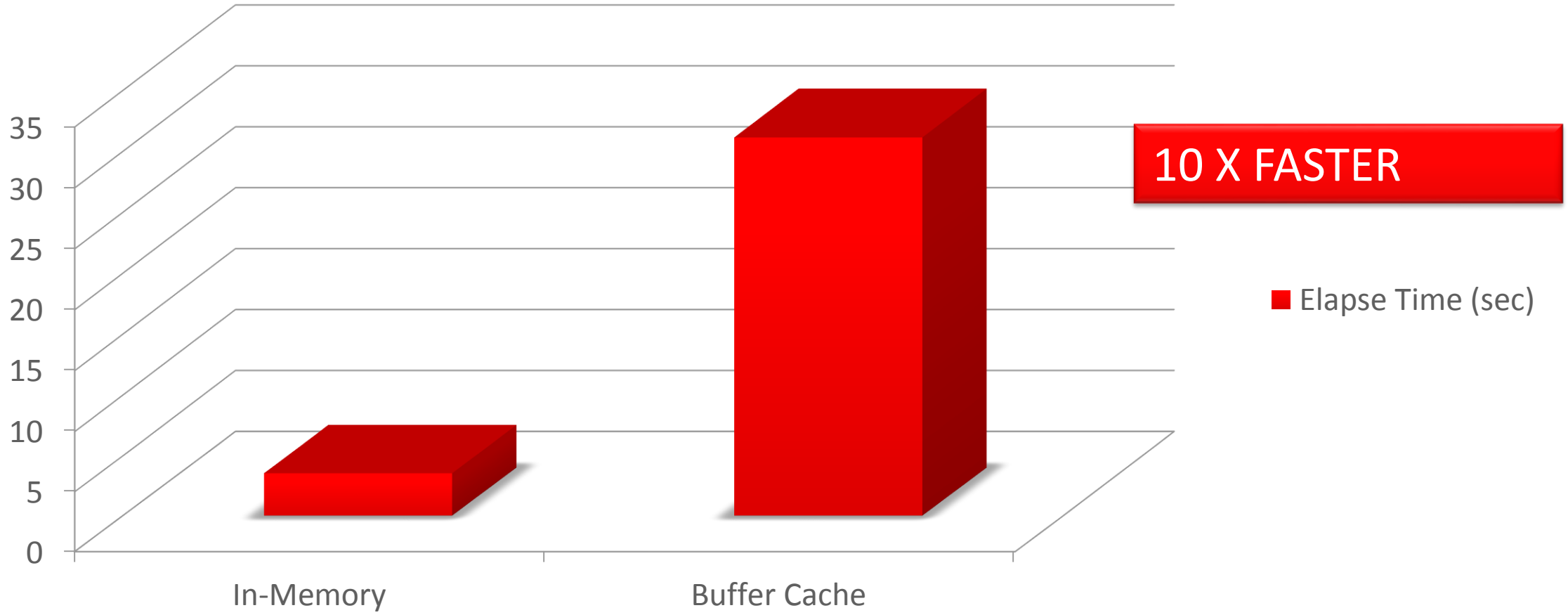
フルテーブル・スキャンのパフォーマンス



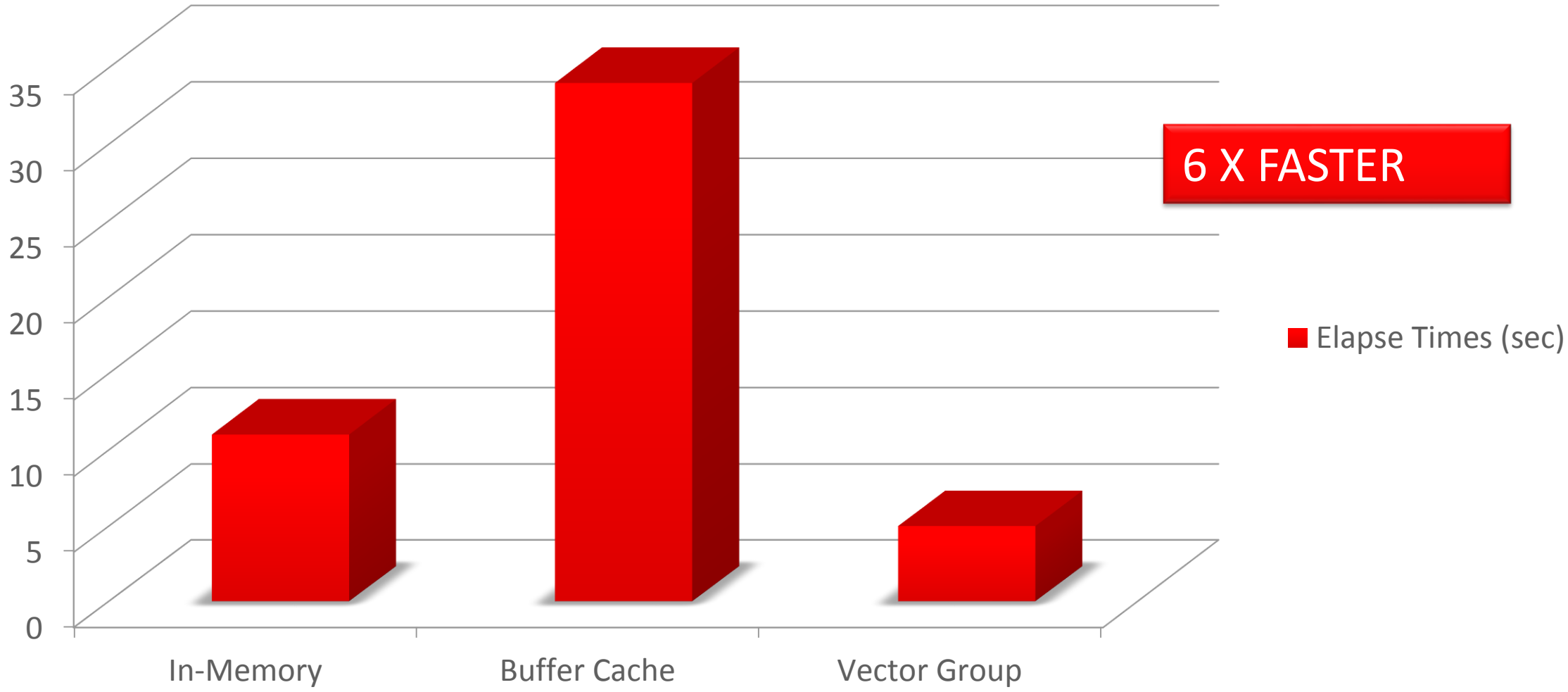
単一レコードの検索



複数テーブルのジョイン



複数カラムの集計演算



まとめ

- ✓カラム・ストアのメモリー・サイズは「INMEMORY_SIZE」で設定される
  - 圧縮アドバイザにより、必要な空き容量の提示が可能
- ✓カラム・ストアに格納する表には「INMEMORY」属性を設定する
  - ALTER TABLE <表名> **INMEMORY**;
- ✓カラム・ストアへのロードは「PRIORITY」属性で制御される
  - ワーカー・プロセス数を指定することにより、負荷の制御が可能
- ✓カラム・ストアを使わない場合は"INMEMORY_QUERY"パラメータ、または "NO_INMEMORY"ヒントを使用
- ✓V$IM_SEGMENT, V$MYSTATを使ってカラム・ストアの状況を監視

Hardware and Software
Engineered to Work Together

VISION 2020

#1 CLOUD

ORACLE JAPAN


ORACLE®