
ORACLE®

免責事項

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。

Oracle Database 12*c* Release 1 (12.1.0.2) CoreTech Seminar

Oracle Database In-Memory: 検索処理の詳細

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 Database & Exadata技術部
丹羽 勝久
2014/09/18

Agenda

1. カラム型データベースの検索処理の特徴
2. 結合処理(join)
3. 集計演算処理(aggregation)

概要再掲

高速な分析をリアルタイム化する新たな技術革新

DBにおける主要な2種類のフォーマット　– ロー型 vs カラム型

ロー (行)型

- OLTP処理を得意とするロー型
  - 例：注文データの挿入と検索
  - 少数の行(ロー)と多数の列(カラム)を高速処理

カラム (列)型

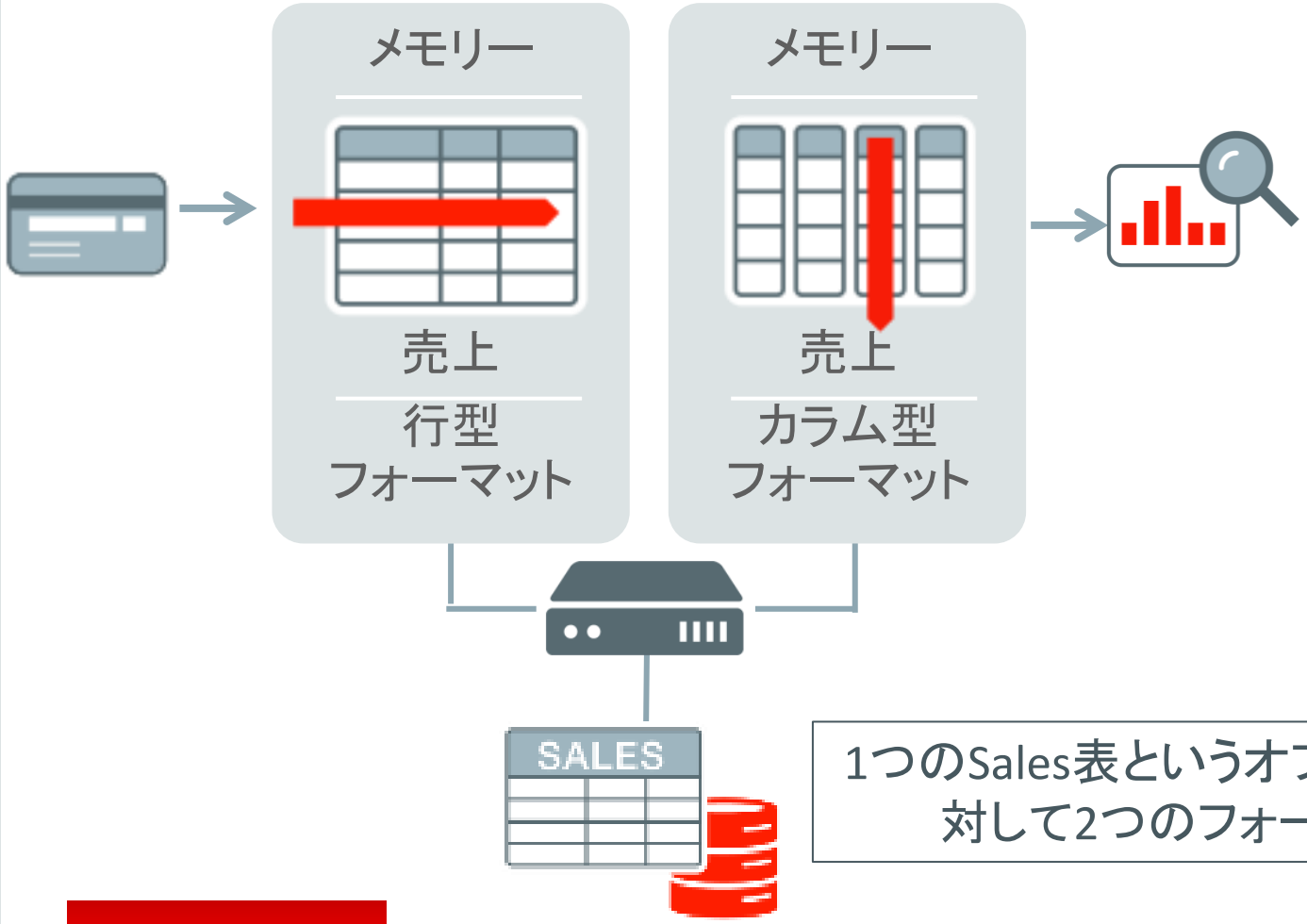
- 集計、分析処理を高速化するカラム型
  - 例：都道府県毎の売上合計のレポート
  - 少数の列(カラム)と多数の行(ロー)を高速処理

Oracle Database In-Memory テクノロジーは
各特性を持つ、2つのフォーマットを“両方同時に”メモリー上にロードし利用可能

概要再掲

高速な分析をリアルタイム化する新たな技術革新

インメモリ・デュアル・フォーマット



- 同一のデータを行型、カラム型双方のフォーマットで保持
  - ✓ インメモリ化指定したもののみ
- 双方のフォーマットを同時に利用可能
  トランザクションの一貫性も担保
- 集計、レポート処理はカラム型フォーマットに対して実行
- OLTP処理は行型フォーマットに対して実行

1つのSales表というオブジェクトに対して2つのフォーマット

行型の表とカラム型の表の構造イメージ

行ストア表

	PRODID	CUSTID	ORDATE	QTY	AMOUNT
行1	123	ABC	04/02	12	350
行2	789	XYX	12/01	43	720
行3	56	GHI	11/10	2	50
行4	432	SRE	2/22	8	143

行

カラム・ストア表

	行1	行2	行3	行4
PRODID	123	789	56	432
CUSTID	ABC	XYX	GHI	SRE
ORDATE	04/02	12/01	11/10	2/22
QTY	12	43	2	8
AMOUNT	350	720	50	143

列

行ストアとカラム・ストアの格納イメージ

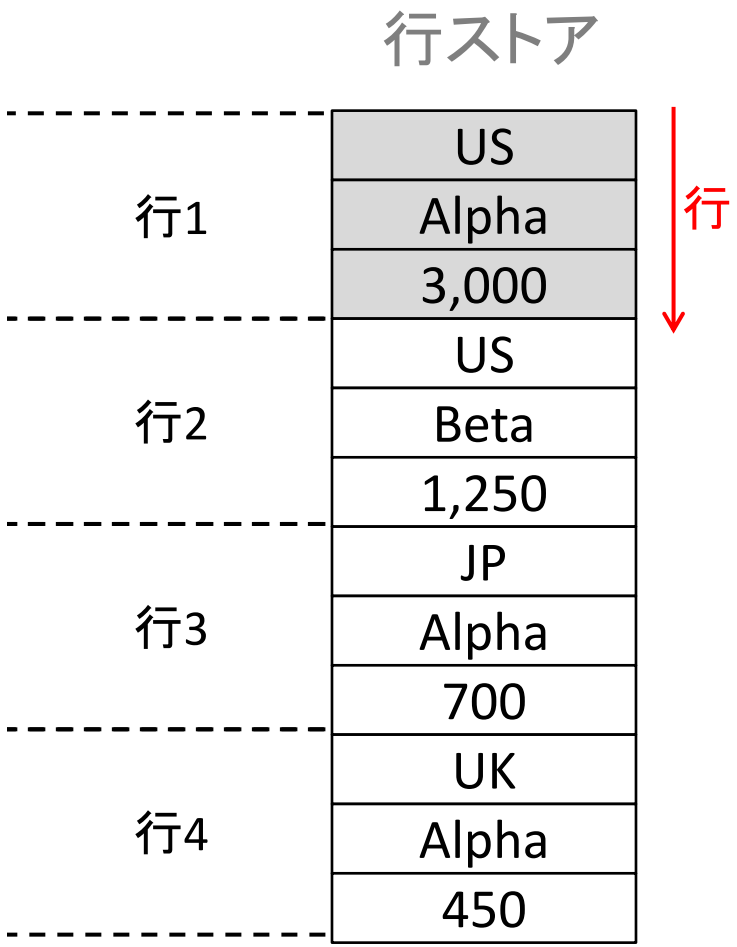
行アクセスのイメージ

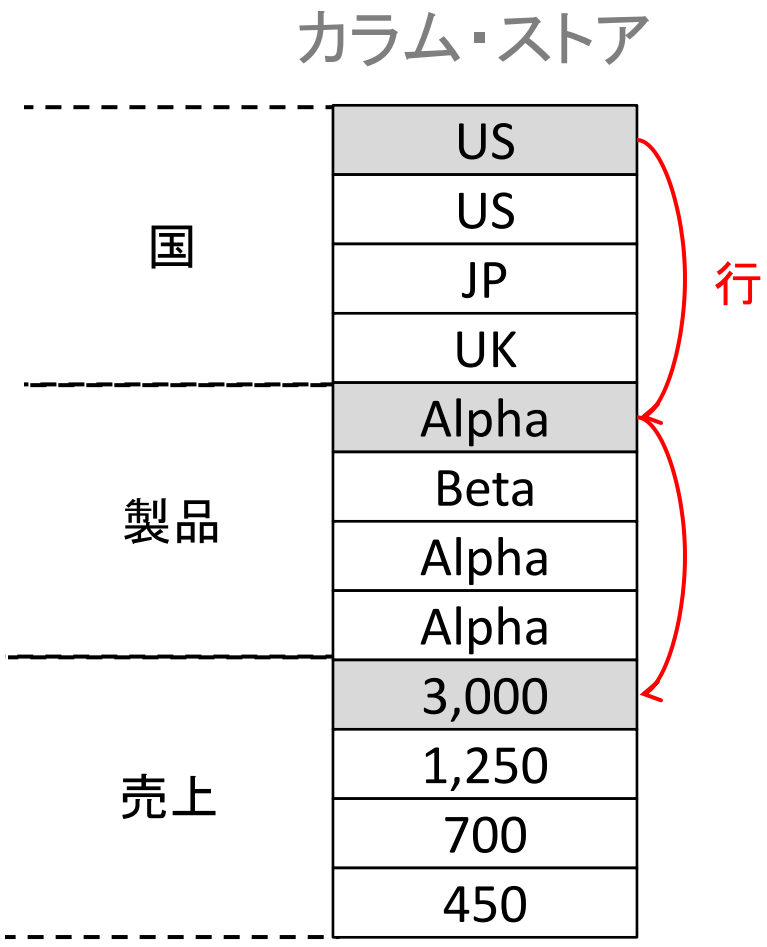
Select * from t1 where ……

表イメージ

国	製品	売上
US	Alpha	3,000
US	Beta	1,250
JP	Alpha	700
UK	Alpha	450

行ストアは行全体のアクセスが効率的





行ストアとカラム・ストアの格納イメージ

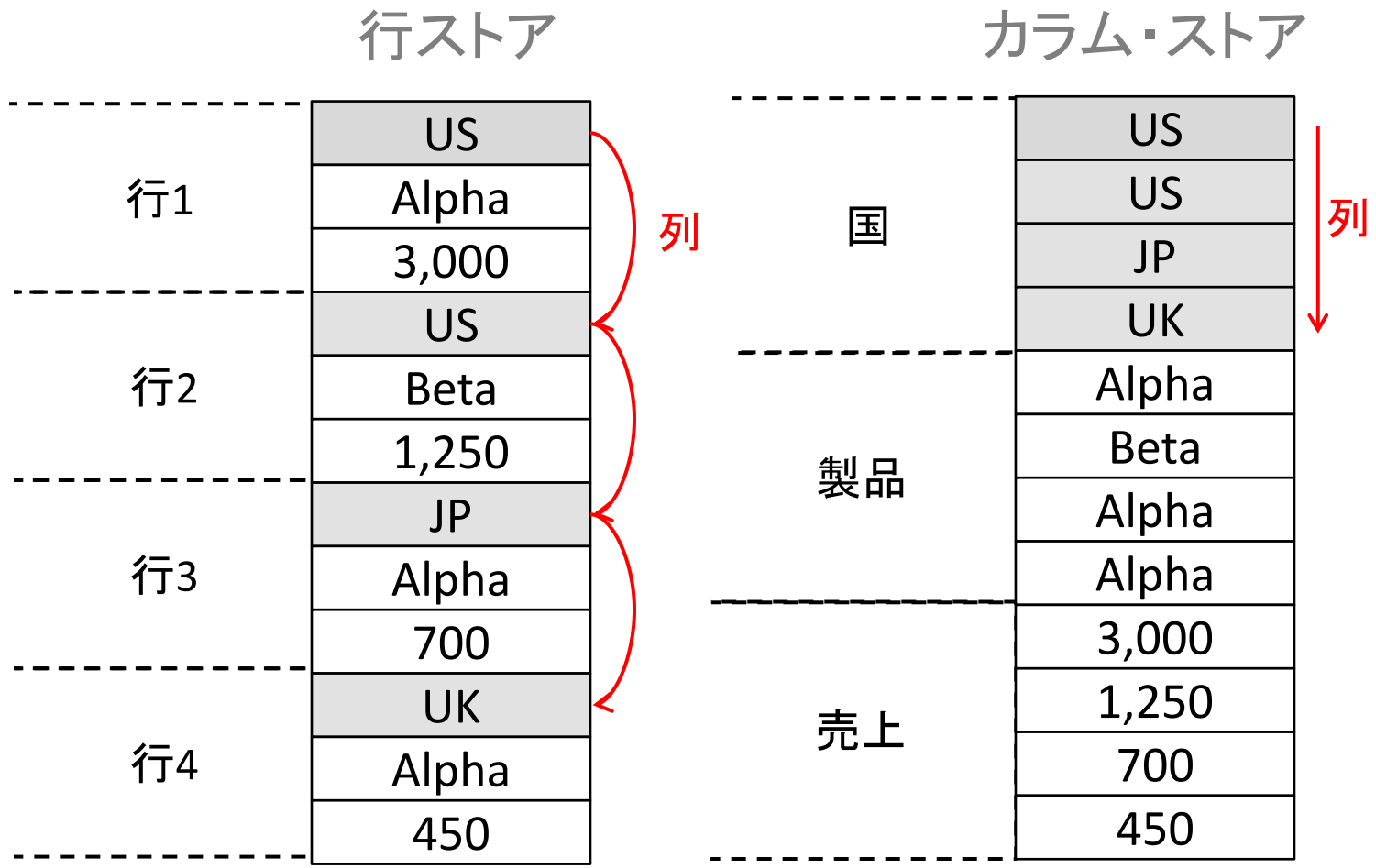
列アクセスのイメージ

表イメージ

国	製品	売上
US	Alpha	3,000
US	Beta	1,250
JP	Alpha	700
UK	Alpha	450

カラム・ストアは少数の
カラム・アクセスが効率的

Select col1 from t1 ;



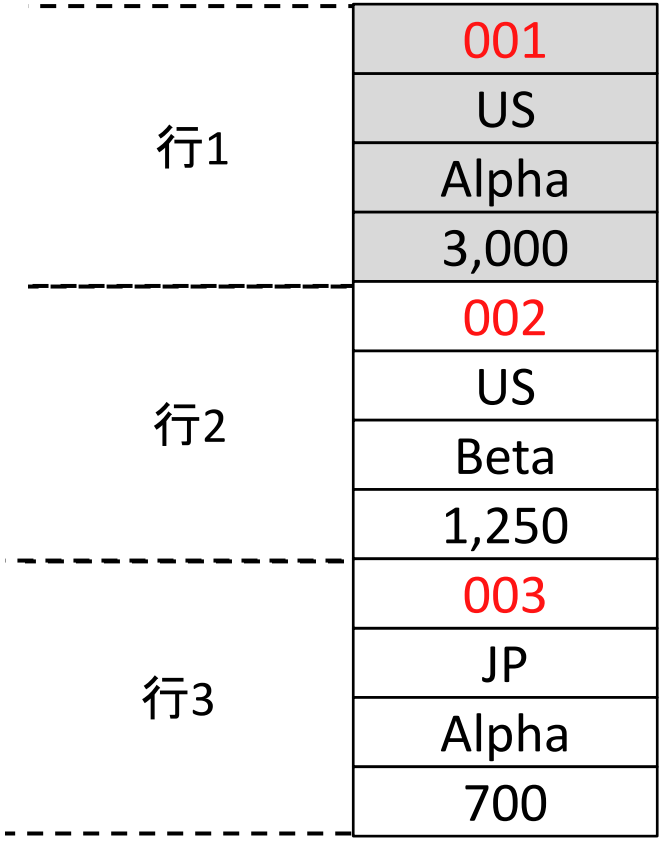
行ストアとカラム・ストアの格納イメージ

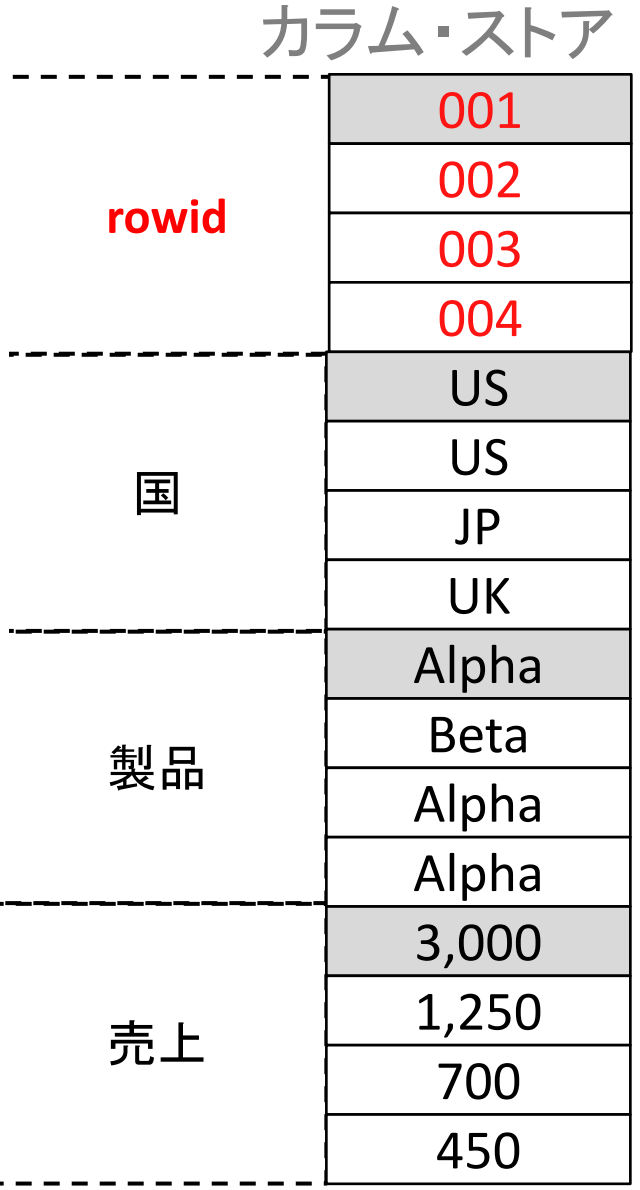
rowid付イメージ

rowid付表イメージ

rowid	国	製品	売上
001	US	Alpha	3,000
002	US	Beta	1,250
003	JP	Alpha	700
004	UK	Alpha	450

カラム・ストアも行の
認識にrowidを利用

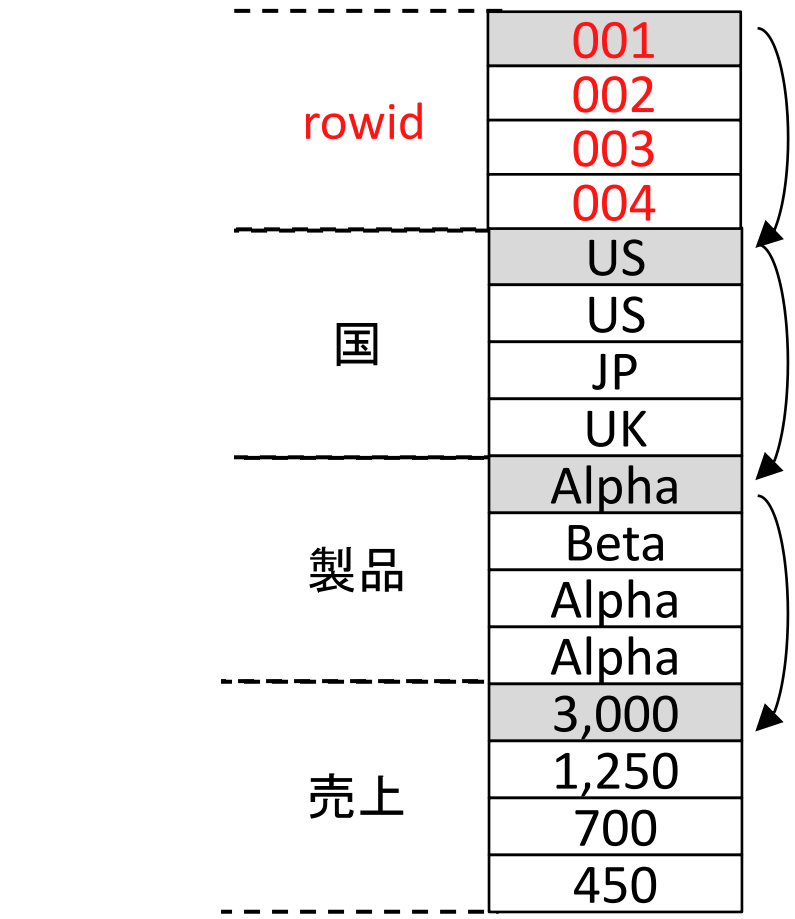




カラム型データベースの基本

カラム・ストアから行データの実体化

sales_t 表：カラム・ストア



select * from
sales_t ;

行データの
実体化

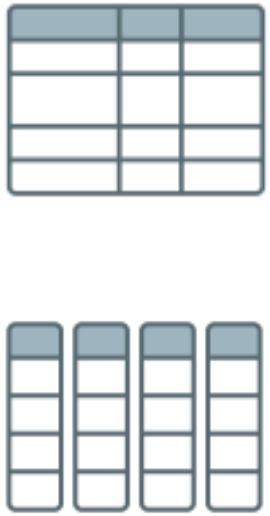
カラム・ストアも行を特定する
rowidを保有する

rowid付 行データ

rowid	国	製品	売上
001	US	Alpha	3,000
002	US	Beta	1,250
003	JP	Alpha	700
004	UK	Alpha	450

SQLから見ると行型もカラム型も透過的

行型もカラム型もどちらもリレーショナル・モデルを表現することに変わりはないため
SQLの変更は必要なく、行型とカラム型表同士の結合処理も可能



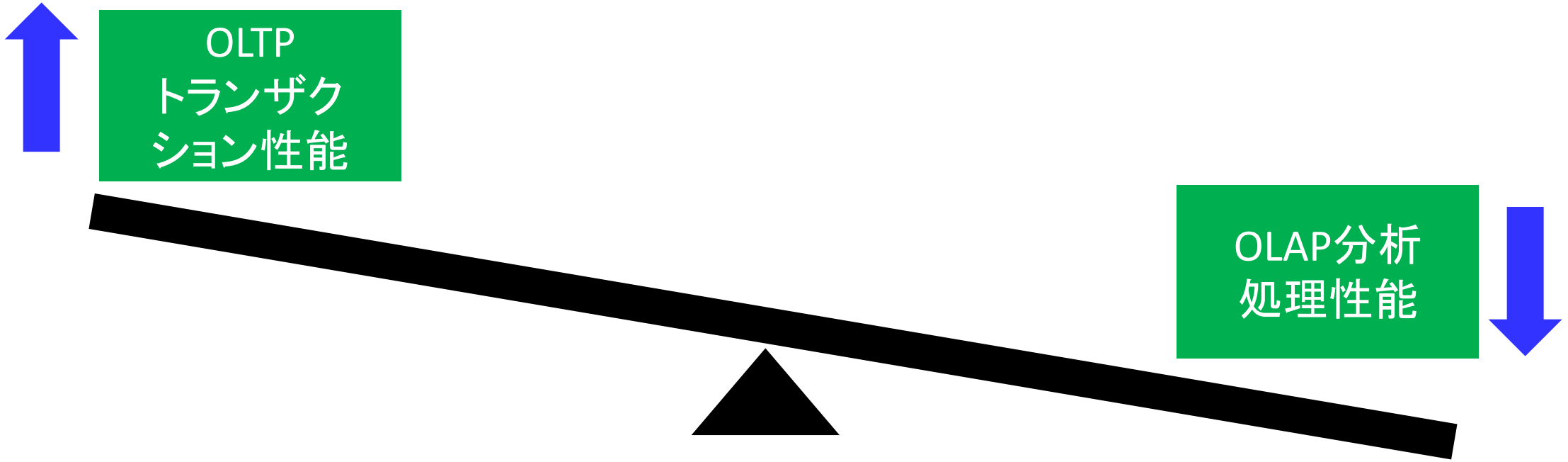
- select col1 from t1; ※列単位アクセス
- select * from t1; ※ 行全体アクセス
- select t1.region, t2.prod_type, ※ 結合、集計、ソート
  sum(t2.amount) from tab_row t1, tab_col t2
  where t1.col1 = t2.col1
  group by t1.region, t2.prod_type
  order by 1, 2;
- ….

行型とカラム型表との
結合処理も可能

OLTPとOLAPの性能向上はトレードオフ

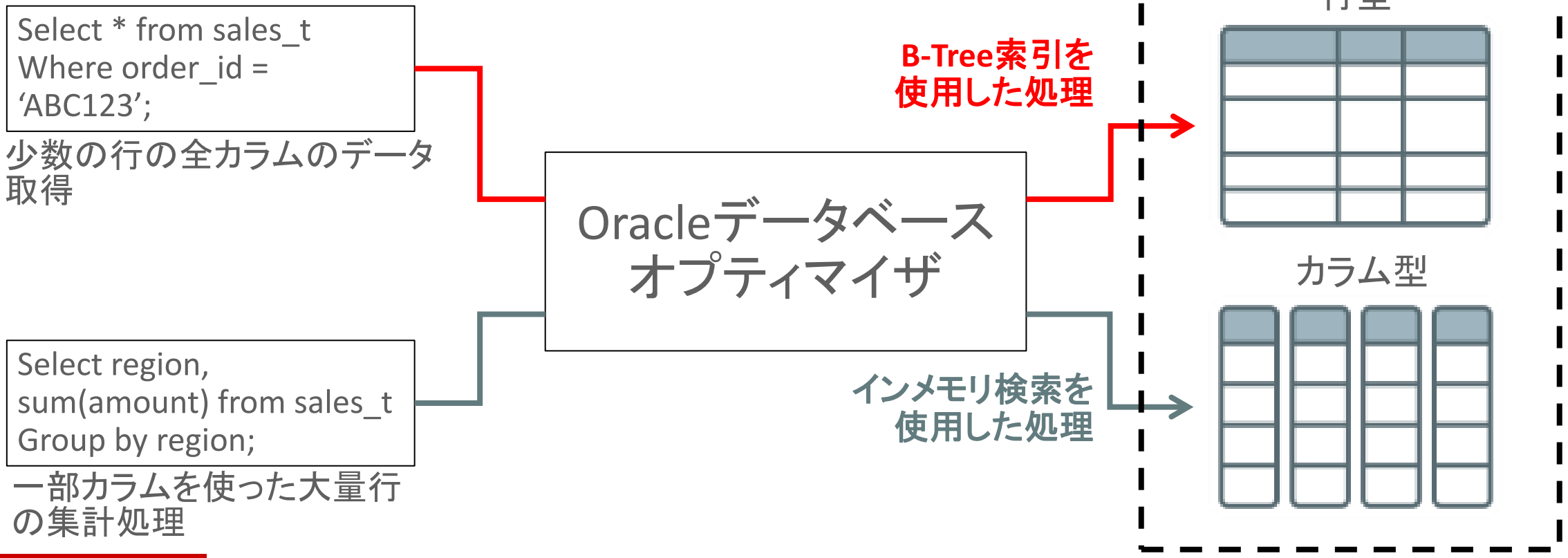
どちらかを性能向上するとどちらかにオーバーヘッドが発生

OLTPとOLAPを1つのデータベースで
共存することは難しい



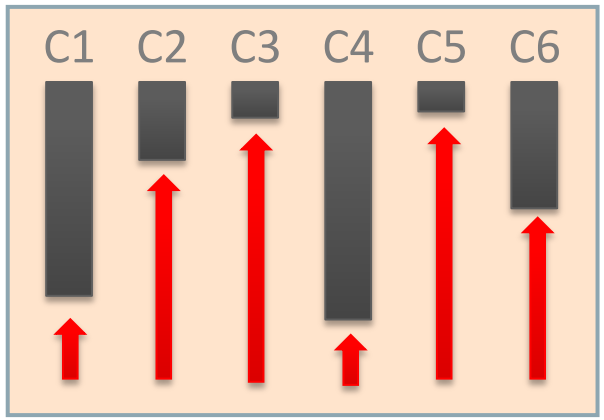
Oracle 12*c* Database In-Memory：デュアル・フォーマット

Oracle 12*c* Database In-Memoryはデュアル・フォーマットなので、データベースのオプティマイザがSQLにあわせて最適なフォーマットを選択してSQLを処理します。(他社のインメモリ機能はハイブリッド型：オブジェクトをどちらの方式にするか決定する必要あり)



カラム型表は何故分析用クエリーが高速か?

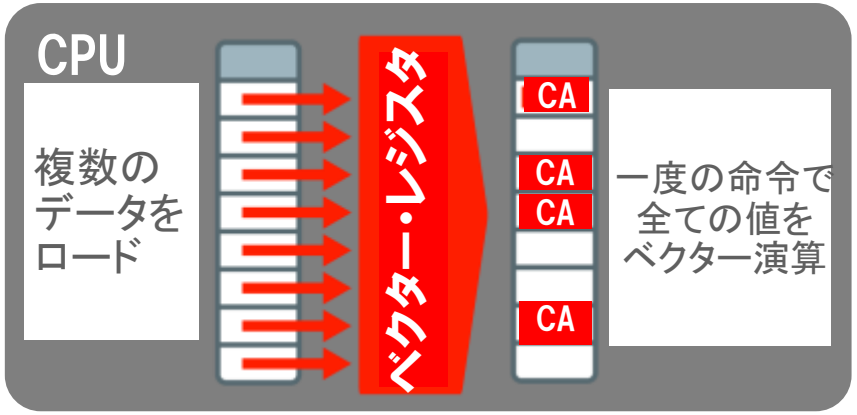
ポイント1:
集計に必要なカラムのみアクセス＋効果的な圧縮技術により圧縮した状態で検索が可能 (ディクショナリ圧縮)



ポイント2:
インメモリ・ストレージ索引により最小限のIMCUのみスキャン

例) where storeid > 8

ポイント3:
最新のプロセッサで搭載されているSIMDにより高速スキャン

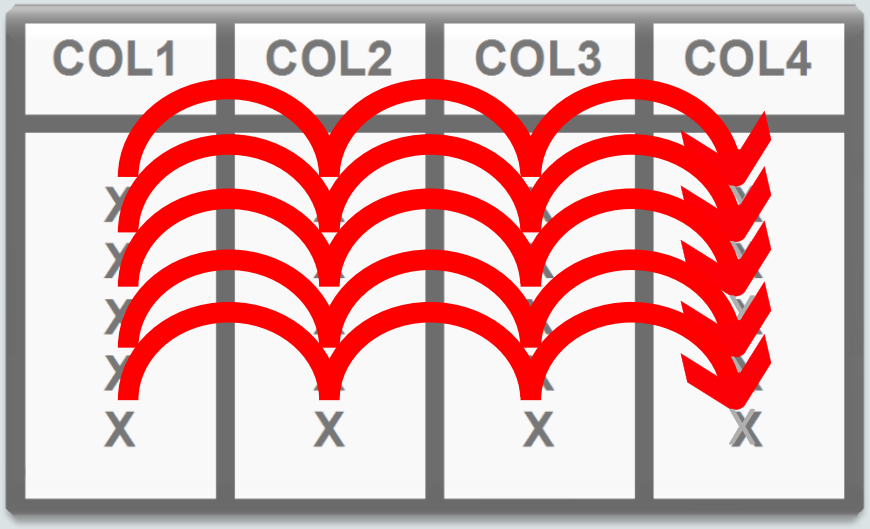


ポイント4:
パラレル・クエリーとパーティション表によりさらに高速化可能

カラム型表は何故分析用クエリーが高速か?

ポイント1-1: 必要なカラムのみアクセス

バッファ・キャッシュ



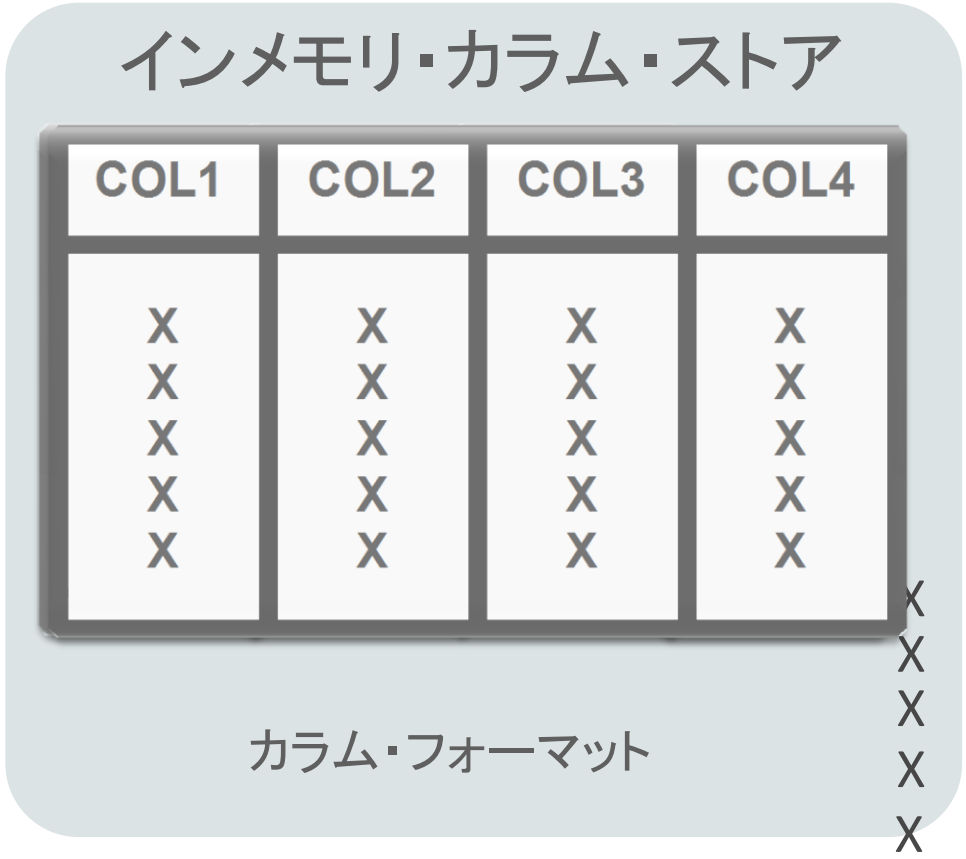
行フォーマット

```
SELECT COL4 FROM MYTABLE;
```

結果

カラム型表は何故分析用クエリーが高速か?

ポイント1-1: 必要なカラムのみアクセス




```
SELECT COL4 FROM MYTABLE;
```

結果

必要なカラムのみアクセス

データの読込量少ない

カラム型表は何故分析用クエリーが高速か?

ポイント1-2：ディクショナリ圧縮

非圧縮

EMP表のJOB列

CLERK
SALESMAN
SALESMAN
MANAGER
SALESMAN
MANAGER
MANAGER
ANALYST
PRESIDENT
SALESMAN
CLERK
CLERK
ANALYST
CLERK

97 bytes

ディクショナリ圧縮

ディクショナリ(distinctされた値)

ソートされた値

カラム値	ディクショナリ値	ビット表現
ANALYST	0	000
CLERK	1	001
MANAGER	2	010
PRESIDENT	3	011
SALESMAN	4	100

カラム値サイズ合計＋ビット値合計
→ 36 bytes + 3bit * 5 = 38 bytes

ディクショナリ圧縮は圧縮した状態で検索が可能

Where job = 'MANAGER'

Where job = 010
に内部的に変換
※圧縮状態で検索可能

エンコードされた各行の値 → 行

001	100	100	010	100	010	010	000	011	100	001	001	000	001

3bit * 14行 = 5.25bytes → 38 + 5.25 = **44 bytes (1/2.2 圧縮)**

ORACLE

カラム型表は何故分析用クエリーが高速か?

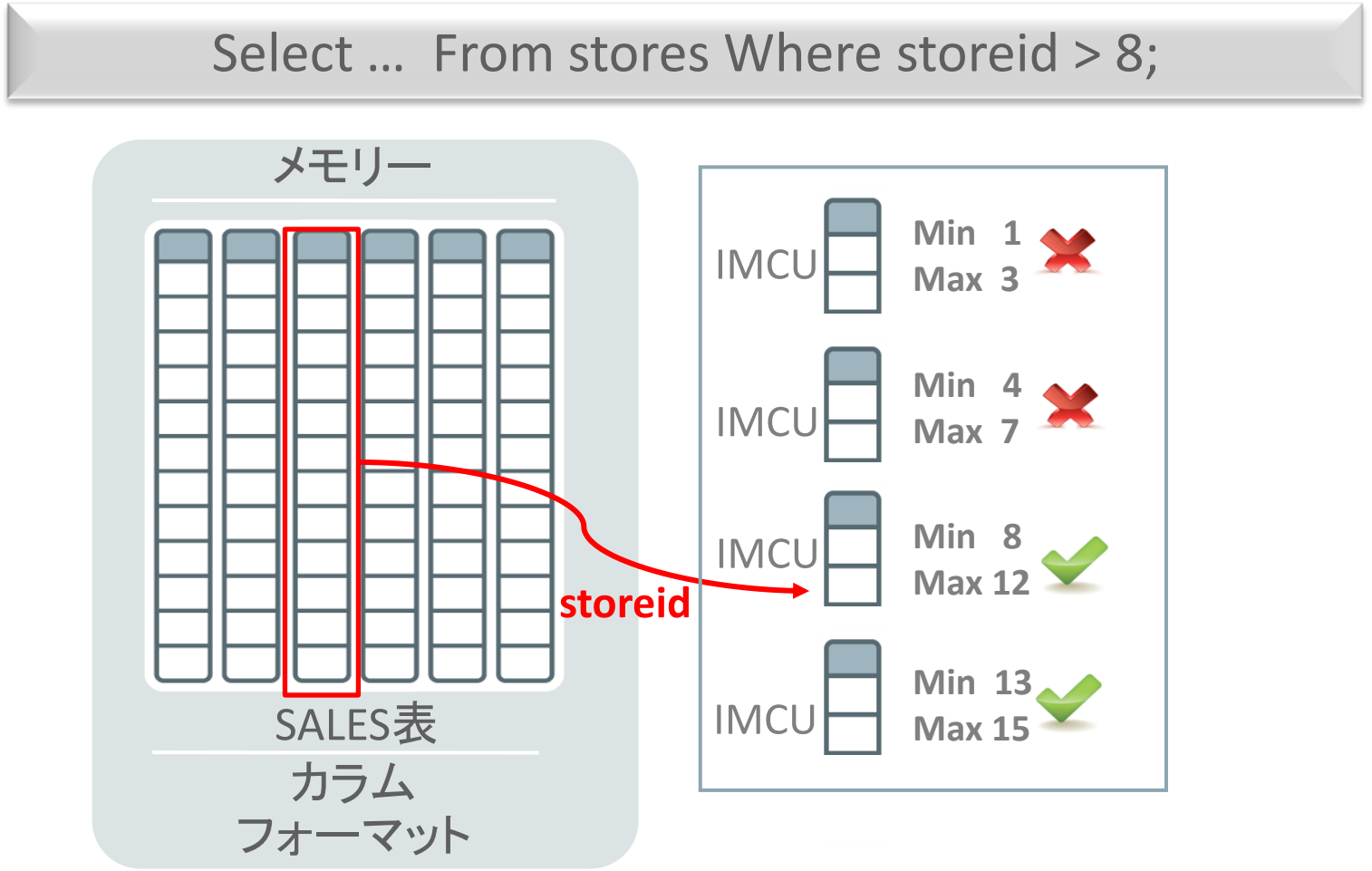
ポイント2: インメモリ・ストレージ索引 (※メモリー内に定義される)

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

SIMDによる効果的な演算

ポイント3：最新のプロセッサで搭載されているSIMD命令セットにより高速スキャン

SIMD： Single Instruction Multiple Data

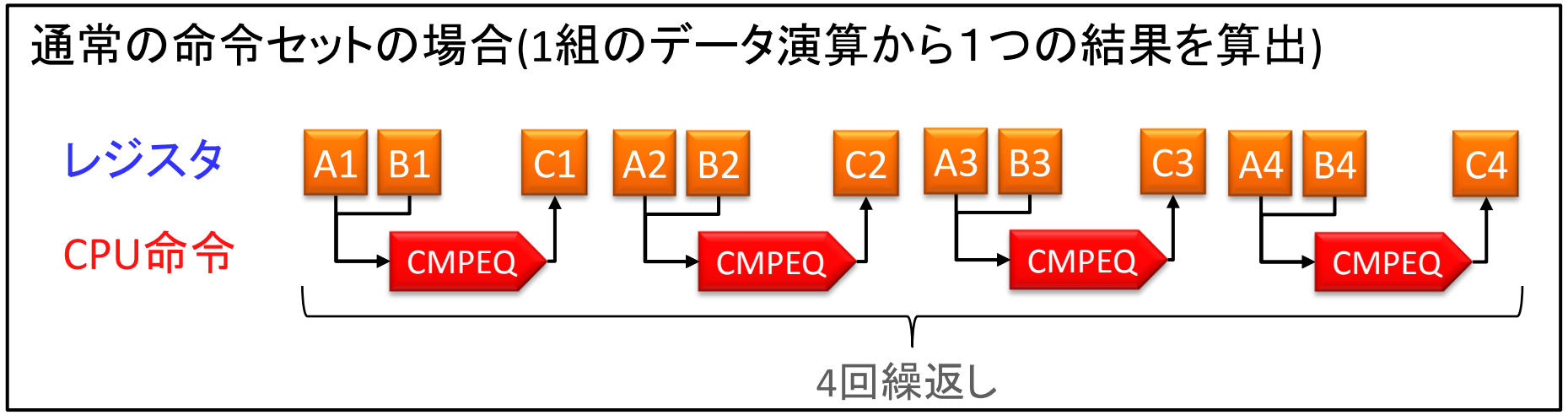
4回の一致比較の場合

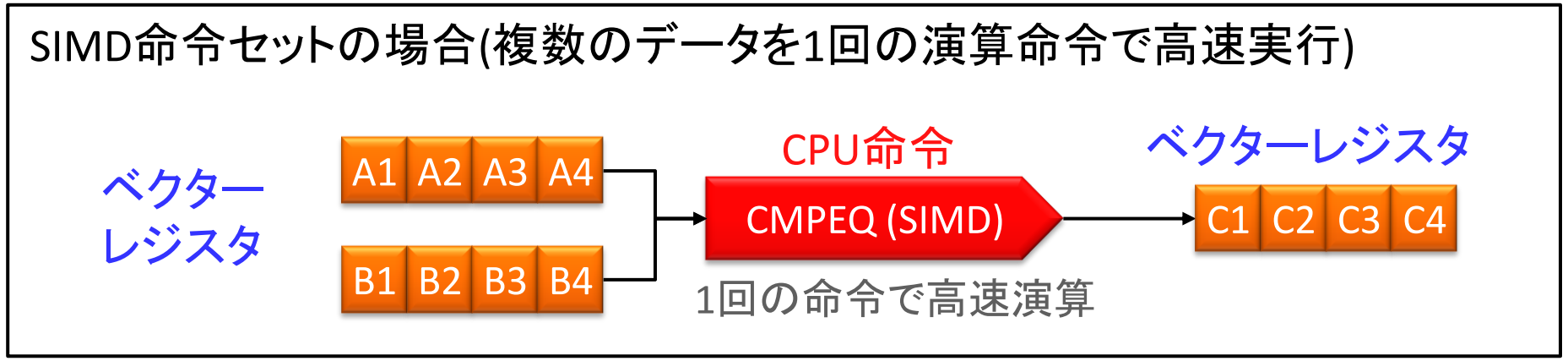
IF (A1 ＝ B1) → C1

IF (A2 ＝ B2) → C2

IF (A3 ＝ B3) → C3

IF (A4 ＝ B4) → C4





カラム型表は何故分析用クエリーが高速か?

ポイント3:最新のプロセッサで搭載されているSIMDにより高速スキャン

インメモリ・カラム・ストア

EMP表

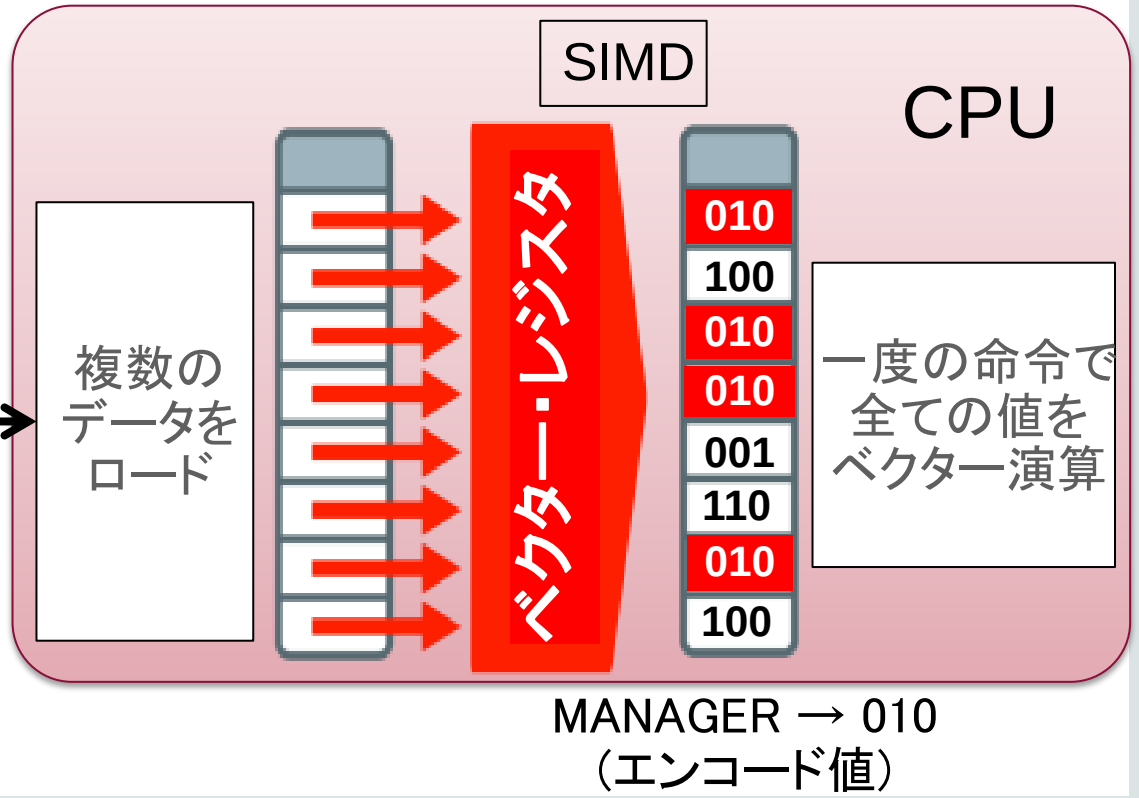
JOBカラム値	ディクショナリ値	ビット表現
ANALYST	0	000
CLERK	1	001
MANAGER	2	010
PRESIDENT	3	011
SALESMAN	4	100

JOB

001 | 100 | 100 | 010 | 100 | 010 | 010 | 000 | 011

ディクショナリ圧縮により
実データ値をビットデータとして扱うことで**より多くのデータをCPUレジスタにロード可能**

例:「MANAGER」職種を検索
(MANAGER → 010)



カラム型表は何故分析用クエリーが高速か?

ポイント4:インメモリ検索はパラレル・クエリー、パーティション表によりさらに高速化

インメモリ検索の実行プラン例

```
-----------------------------------------------------------
| Id  | Operation                          | Name         |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                   |              |
|   1 |  SORT AGGREGATE                    |              |
|   2 |   TABLE ACCESS INMEMORY FULL       | LINEORDER    |
-----------------------------------------------------------
```

- 新しいアクセス方法

  TABLE ACCESS INMEMORY FULL

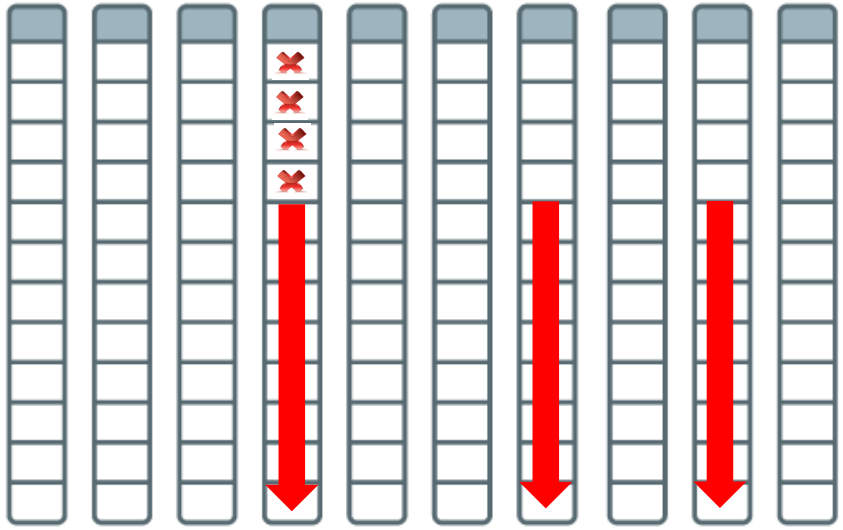
- インメモリ検索を有効／無効化するパラメータ

  INMEMORY_QUERY = {enable | disable}

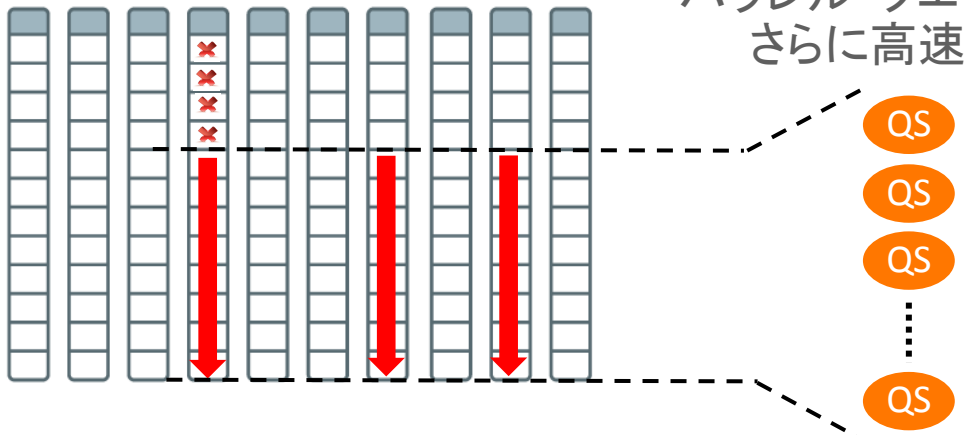
カラム型表は何故分析用クエリーが高速か?

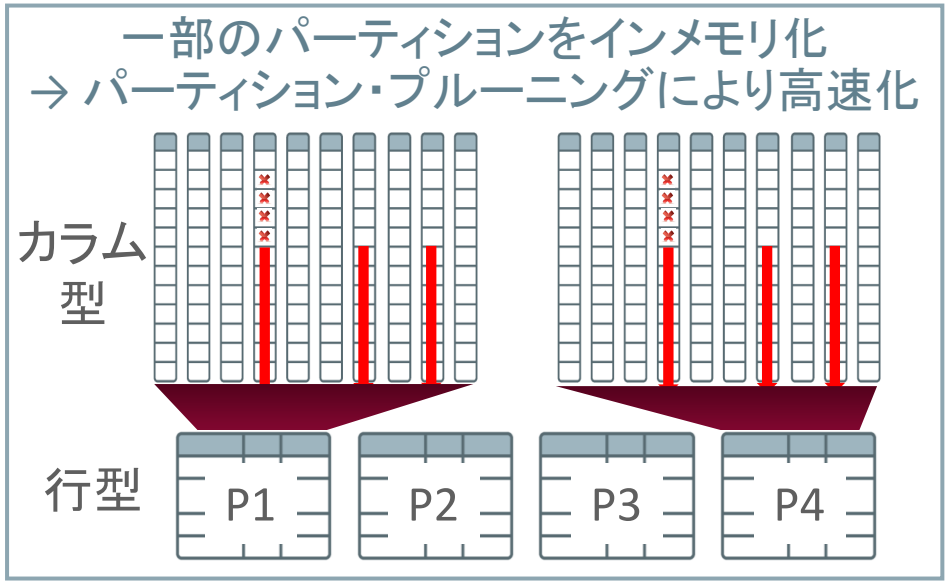
ポイント4:インメモリ検索はパラレル・クエリー、パーティション表によりさらに高速化

インメモリ・スキャン ＝ TABLE ACCESS INMEMORY FULL



基本的にFull Table Scanの発展系
→ データはインメモリ・カラム型で圧縮
→ 必要なカラムのみアクセス
→ インメモリ・ストレージ索引により
最低限のIMCUスキャン

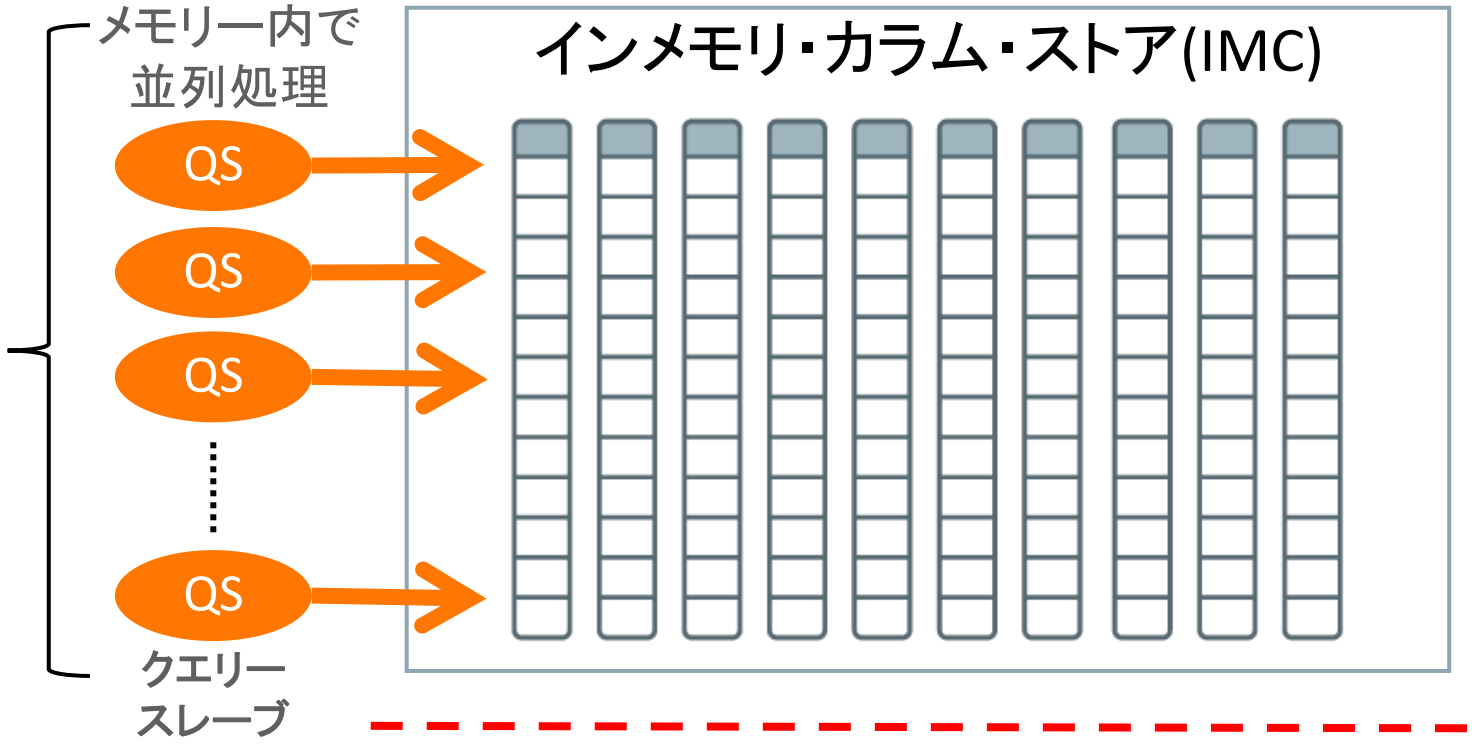




Database In-Memoryとパラレル・クエリー

autoDOPはインメモリ構成も
考慮してパラレル度を決定

インメモリ・カラム・ストアなので
対象データはメモリー内にある



In-Memory Parallel
Executionと同様の動き
(Buffer CacheではなくIMC利用)

＋

高速なインメモリ検索

- 必要なカラムのみアクセス
- 効果的な圧縮(高速検索)
- 効率的なSIMD利用
- インメモリ・ストレージ索引

基本的にディスク
読込は発生しない

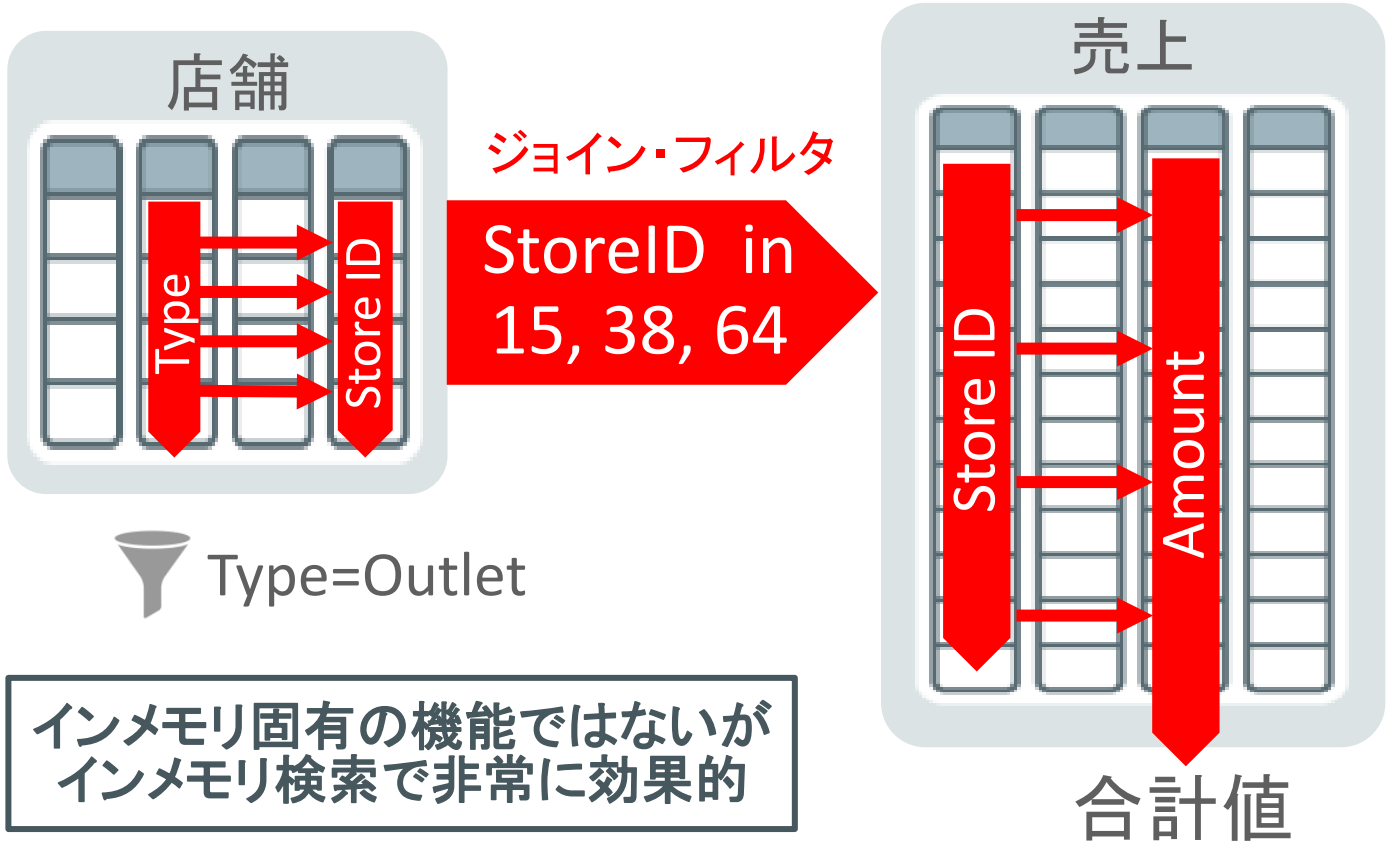
Agenda

1. カラム型データベースの検索処理の特徴
2. 結合処理(join)
3. 集計演算処理(aggregation)

インメモリ検索による表の結合処理の高速化

複数表の結合処理を内部的に高速カラム検索に変換 (ベクター結合)

例：直販店(outlet)の売上合計を集計



**インメモリ固有の機能ではないが
インメモリ検索で非常に効果的**

■ インメモリ・カラム・ストアにより複数表の結合処理を高速化

1. ジョイン・フィルタと呼ばれるフィルタを<u>カラム検索を使用して作成</u>
   - 店舗表のTYPE='OUTLET' に該当するStoreIDをリスト
2. 作成したジョイン・フィルタの条件にあう売上表のAMOUNTの合計値を計算
   - ジョイン・フィルタから以下の条件を生成「where StoreID in (15, 38, 64)」
   - 上記の条件にヒットする行の売上表単体のカラム検索により高速にAMOUNT列の合計値を算出(SUM(AMOUNT))

ベクター結合の実行計画例

```
-----------------------------------------------------------
| Id  | Operation                          | Name         |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                   |              |
|   1 |  SORT AGGREGATE                    |              |
|   2 |   PX COORDINATOR                   |              |
|   3 |    PX SEND QC (RANDOM)             | :TQ10000     |
|   4 |     SORT AGGREGATE                 |              |
|*  5 |      HASH JOIN                     |              |
|   6 |       JOIN FILTER CREATE           | :BF0000      |
|*  7 |        TABLE ACCESS INMEMORY FULL  | DATE_DIM     |
|   8 |       JOIN FILTER USE              | :BF0000      |
|   9 |        PX BLOCK ITERATOR           |              |
|* 10 |         TABLE ACCESS INMEMORY FULL | LINEORDER    |
-----------------------------------------------------------
```

実行SQL)

```
select
sum(lo_revenue*lo_discount)
from lineorder, date_dim
where lo_orderdate = d_datekey
and d_year = 1996;
```

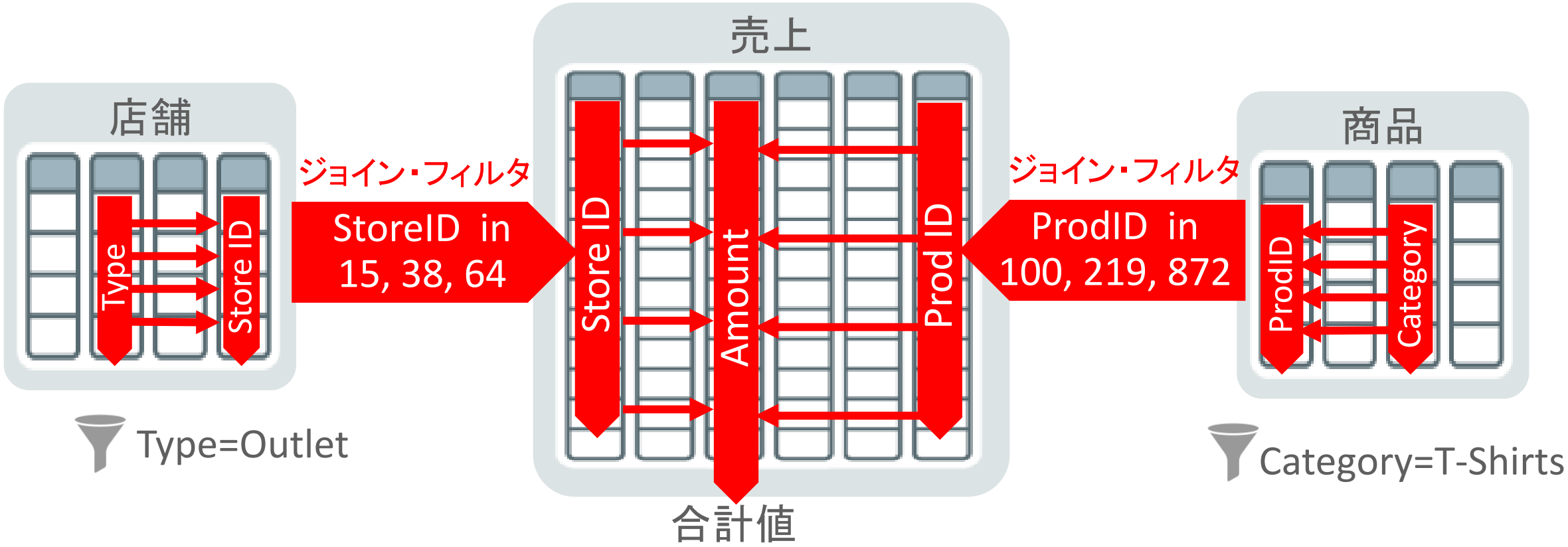
①ジョイン・フィルタ作成(DATE_DIM)
※:BF0000 (ブルーム・フィルタ)

②ジョイン・フィルタ利用した
LINEORDER表のカラム検索
※この例はパラレル・クエリー実行

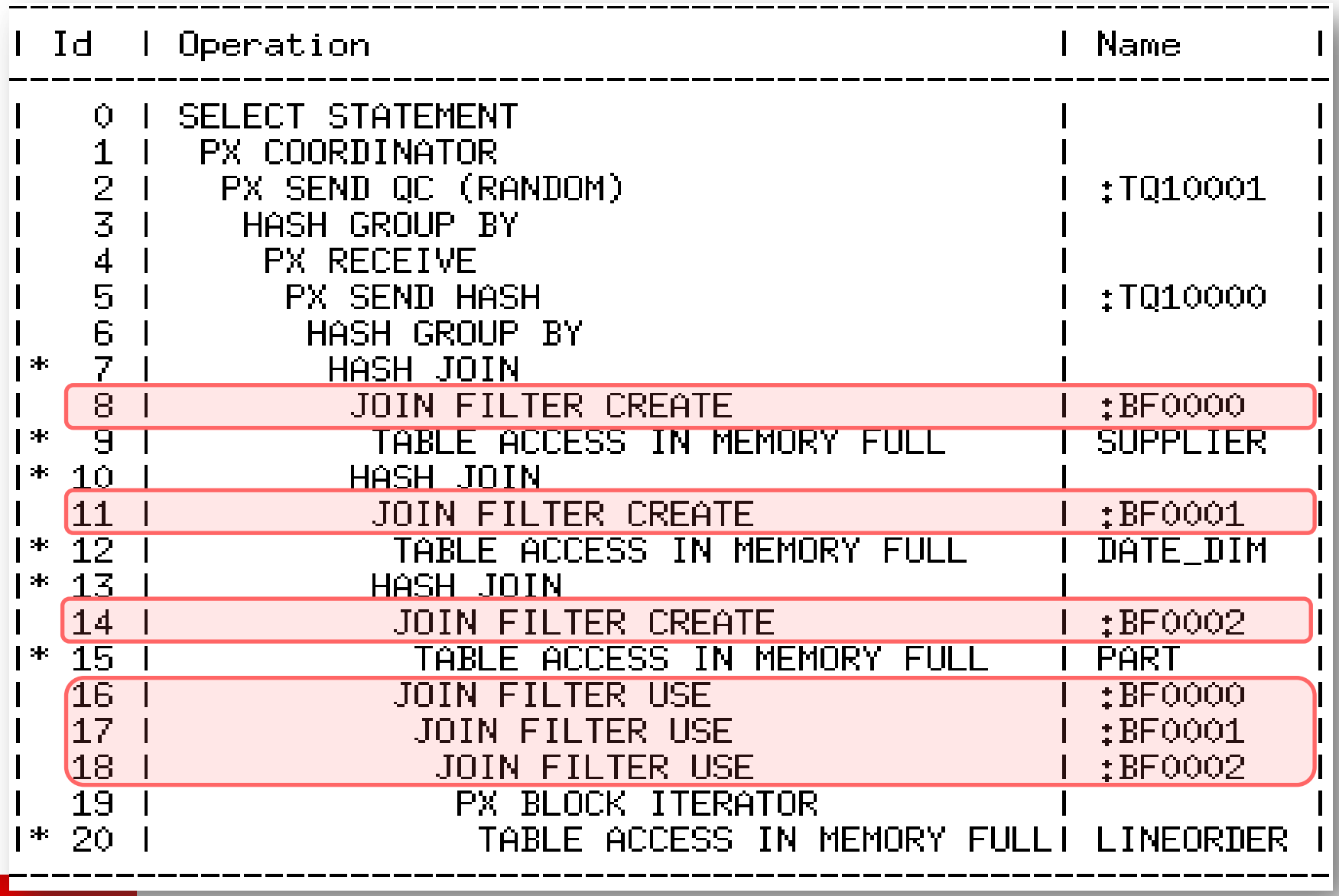
複数表のベクター結合の実行イメージ

例：直販店(outlet)のT-Shirtsの合計売上を集計

3つの表のジョイン処理を
売上表の単一のカラム検索に変換



複数表のベクター結合の実行計画例

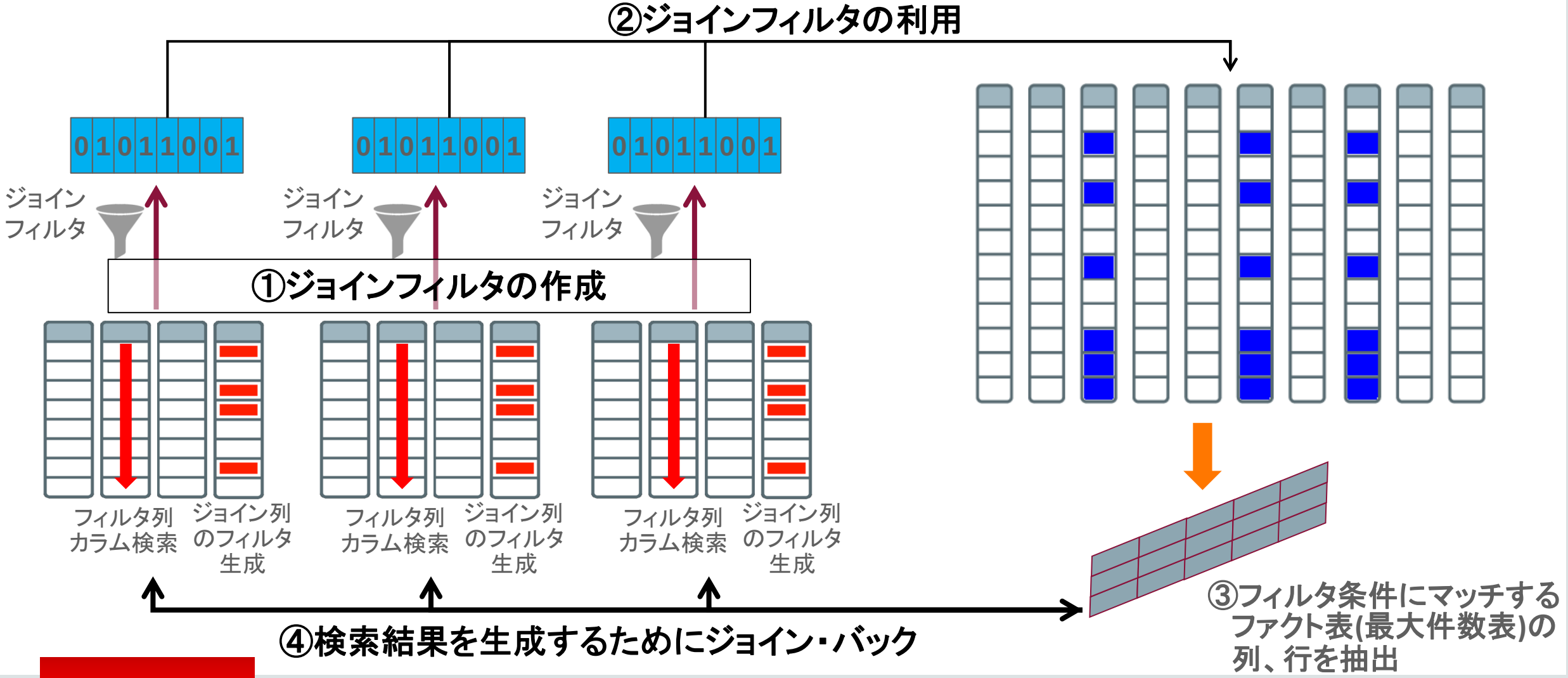


Id	Operation	Name
0	SELECT STATEMENT	
1	PX COORDINATOR	
2	PX SEND QC (RANDOM)	:TQ10001
3	HASH GROUP BY	
4	PX RECEIVE	
5	PX SEND HASH	:TQ10000
6	HASH GROUP BY	
*7	HASH JOIN	
8	JOIN FILTER CREATE	:BF0000
*9	TABLE ACCESS IN MEMORY FULL	SUPPLIER
*10	HASH JOIN	
11	JOIN FILTER CREATE	:BF0001
*12	TABLE ACCESS IN MEMORY FULL	DATE_DIM
*13	HASH JOIN	
14	JOIN FILTER CREATE	:BF0002
*15	TABLE ACCESS IN MEMORY FULL	PART
16	JOIN FILTER USE	:BF0000
17	JOIN FILTER USE	:BF0001
18	JOIN FILTER USE	:BF0002
19	PX BLOCK ITERATOR	
*20	TABLE ACCESS IN MEMORY FULL	LINEORDER

ジョイン・フィルタ作成

ジョイン・フィルタ利用

ジョイン・フィルタを使ったジョイン処理のイメージ



通常の結合処理との実行コスト比較

SQL例

```
Select   p.p_name, sum(l.lo_revenue*1.00212/3.12388832)
  From  PART p, LINEORDER l
where  l.lo_partkey = p.p_partkey
    and  p.p_name  in  (
                ‘hot lavender’ , ‘violet grey‘ , 'rose pink’,
                'yellow grey‘ , 'white snow‘ , 'spring olive‘
Group by p.p_name;
```

通常の結合処理との実行コスト比較

インメモリ検索の<u>ベクター結合を無効化</u>

```
SQL> /

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.12       0.12          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2     76.16      76.16          0          7          0           6
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4     76.28      76.28          0          7          0           6

Elapsed: 00:01:18.31

-------------------------------------------------------------------------------------------
| Id  | Operation                          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |             |     6 |   186 | 79134  (40)| 00:00:04 |
|   1 |  HASH GROUP BY                     |             |     6 |   186 | 79134  (40)| 00:00:04 |
|*  2 |   HASH JOIN                        |             |   586K|    17M| 79092  (40)| 00:00:04 |
|*  3 |    TABLE ACCESS INMEMORY FULL      | PART        |  1003 | 20060 |   206  (31)| 00:00:01 |
|   4 |    TABLE ACCESS INMEMORY FULL      | LINEORDER   |   600M|  6294M| 74051  (36)| 00:00:03 |
-------------------------------------------------------------------------------------------
```

通常の結合処理との実行コスト比較

インメモリ検索の<u>ベクター結合を有効化</u>

```
SQL> /

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.12       0.12          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2     12.48      12.48          0          7          0           6
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4     12.61      12.61          0          7          0           6

Elapsed: 00:00:13.02

-----------------------------------------------------------------------------------------
| Id  | Operation                          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |             |     6 |   186 | 79134  (40)| 00:00:04 |
|   1 |  HASH GROUP BY                     |             |     6 |   186 | 79134  (40)| 00:00:04 |
|*  2 |   HASH JOIN                        |             |   586K|    17M| 79092  (40)| 00:00:04 |
|   3 |    JOIN FILTER CREATE              | :BF0000     |  1003 | 20060 |   206  (31)| 00:00:01 |
|*  4 |     TABLE ACCESS INMEMORY FULL     | PART        |  1003 | 20060 |   206  (31)| 00:00:01 |
|   5 |    JOIN FILTER USE                 | :BF0000     |   600M|  6294M| 74051  (36)| 00:00:03 |
|*  6 |     TABLE ACCESS INMEMORY FULL     | LINEORDER   |   600M|  6294M| 74051  (36)| 00:00:03 |
-----------------------------------------------------------------------------------------
```

Agenda

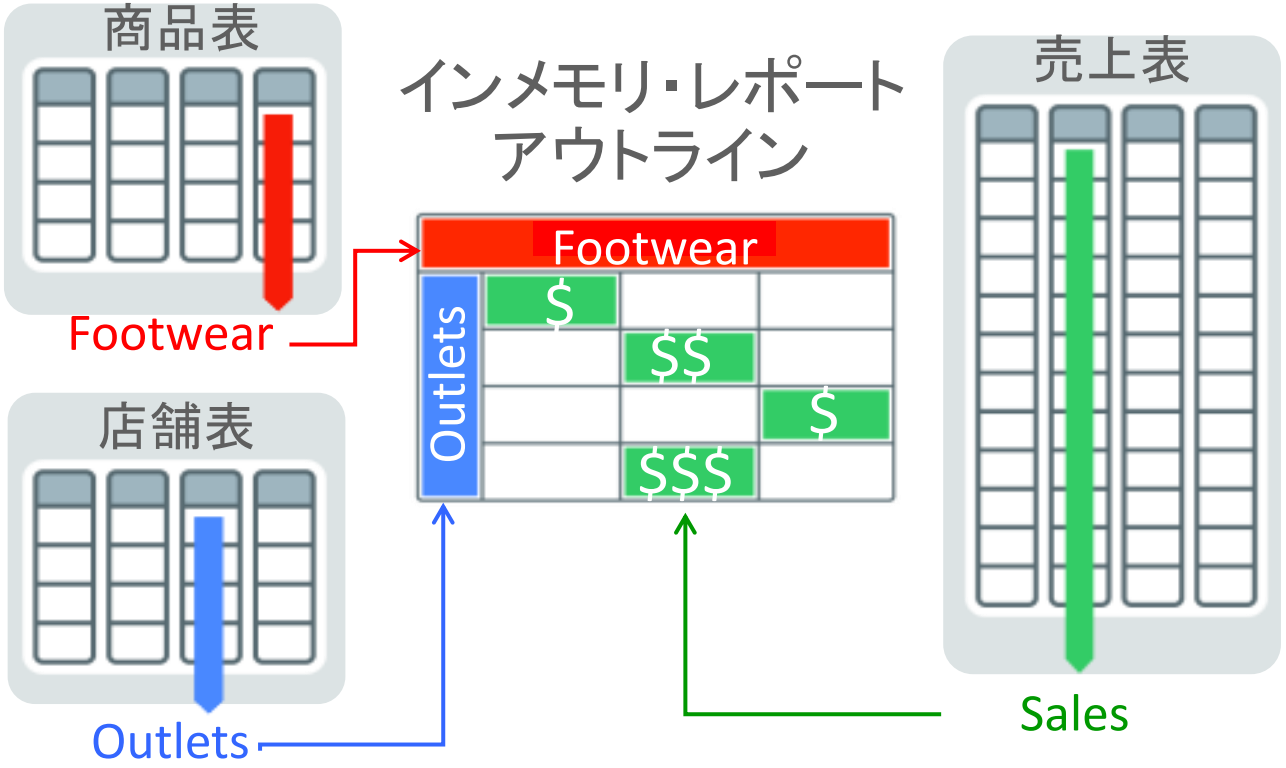
1. カラム型データベースの検索処理の特徴
2. 結合処理(join)
3. 集計演算処理(aggregation)

インメモリ検索による表の集計処理の高速化

ベクターGroup By(Vector Group By)

**インメモリ固有の機能ではないが
インメモリ検索で非常に効果的**

例: アウトレットでの靴の売上を集計



- レポート・アウトラインをメモリー上に動的に作成(インメモリ配列)
- レポート内の集計値はファクト表のスキャン中に展開
- 事前定義された多次元キューブを使わずに高速化

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

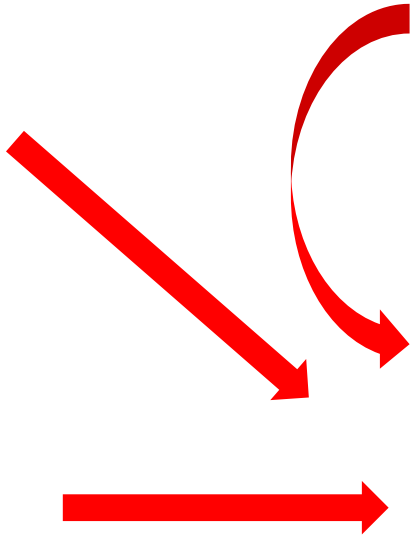
ストア表(Stores)

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
```

結合キー

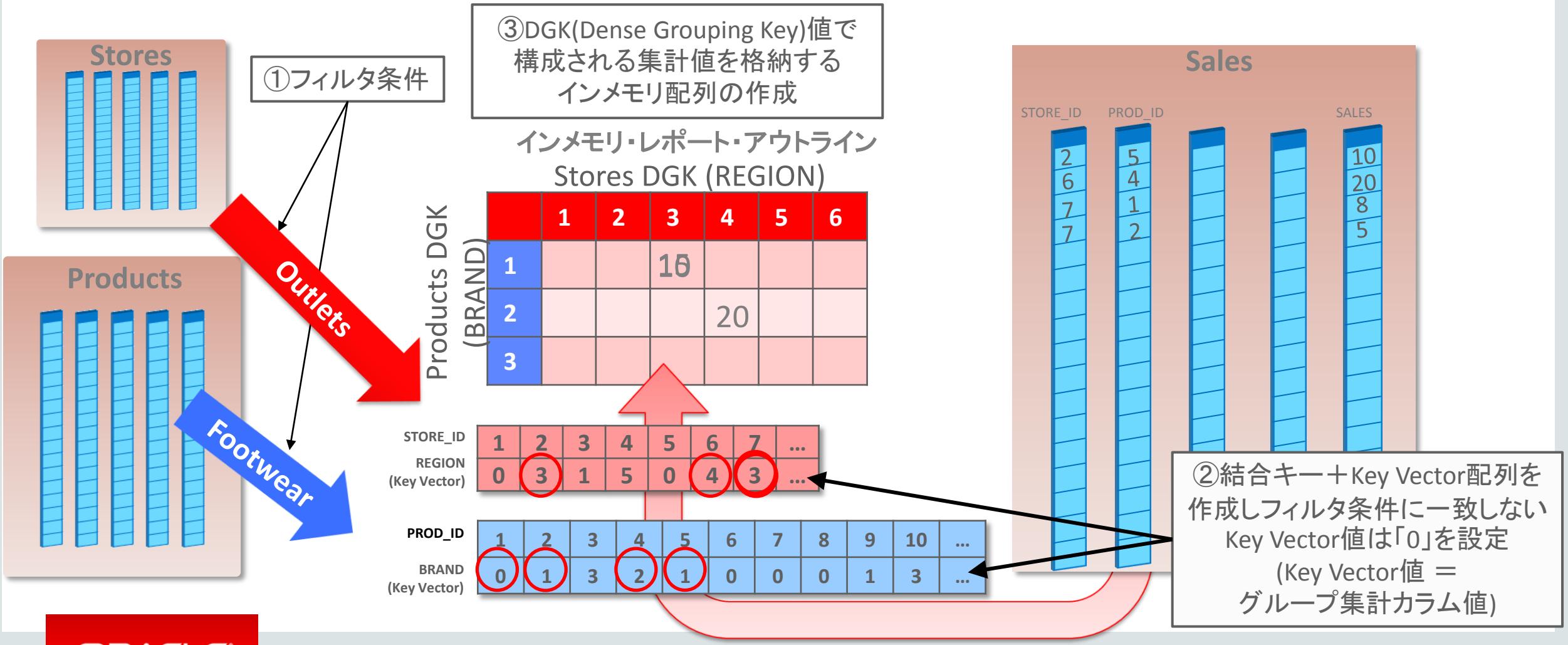
フィルタ条件

グループキー
(Key Vector)

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ー上の配列(インメモリ配列)を使う



ベクター Group By – 実行例 SQL

```
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;
```

通常の集計処理との実行コスト比較

比較結果

インメモリ検索 ― **ベクター Group By無効化**

Elapsed: 00:00:51.11

```
Statistics
----------------------------------------------------------
        456  recursive calls
         60  db block gets
      15602  consistent gets
        105  physical reads
          0  redo size
       8073  bytes sent via SQL*Net to client
        673  bytes received via SQL*Net from client
         13  SQL*Net roundtrips to/from client
          9  sorts (memory)
          0  sorts (disk)
        175  rows processed
```

通常の集計処理との実行コスト比較

比較結果

インメモリ検索 ― **ベクター Group By有効化**

Elapsed: 00:00:28.57

```
Statistics
----------------------------------------------------------
          90  recursive calls
          41  db block gets
         132  consistent gets
           4  physical reads
        3792  redo size
        8073  bytes sent via SQL*Net to client
         673  bytes received via SQL*Net from client
          13  SQL*Net roundtrips to/from client
          11  sorts (memory)
           0  sorts (disk)
         175  rows processed
```

まとめ

- インメモリ検索は分析クエリーが高速
  - 余分なカラム読込なし
  - 効果的な圧縮方法ー圧縮した状態で高速検索 (ディクショナリ検索)
  - 必要最低限の領域(IMCU)のみアクセスするインメモリ・ストレージ索引
  - SIMDによる高速スキャン
  - パラレル・クエリー、パーティション表との相性が良い
- インメモリ化することで結合処理、集計演算も高速
  - ベクター結合(ブルーム・フィルタ)による高速結合処理
  - ベクターGroup By(Key Vector、インメモリ配列)による高速集計処理

リファレンス

マニュアル・ドキュメント

- ベクター結合 (Vector Join)
  - Oracle® Databaseデータ・ウェアハウス・ガイド 12*c* リリース1 (12.1)
    - **ベクトル結合を使用した結合パフォーマンスの向上**
      http://docs.oracle.com/cd/E57425_01/121/DWHSG/ch2logdes.htm#CIHGBAFF
- ベクターGroup By (Vector Group By)
  - Oracle® Databaseデータ・ウェアハウス・ガイド 12*c* リリース1 (12.1)
    - インメモリ集計
      http://docs.oracle.com/cd/E57425_01/121/DWHSG/aggreg.htm#BCGFFGBA
  - Oracle® Database SQLチューニング・ガイド 12*c* リリース1 (12.1)
    - 5.7 インメモリー集計
      http://docs.oracle.com/cd/E57425_01/121/TGSQL/tgsql_transform.htm#BABFGEAE

Hardware and Software
Engineered to Work Together


ORACLE®