Aktuelle Entwicklungen im SAP BW für Oracle InMemory

Jörn Bartels
Oracle
München

Keywords
SAP Businesswarehouse Inmemory Optimizer Performance Starquery

Introduction
In SAP BW data was stored since recently in so called traditional data cubes. They were stored in a snowflake schema. The data was accessed using bitmapped indexes and the query was optimized by a "star transformation". With the introduction of the Inmemory column store the data is now stored in a so called "flatcube". The schema is simplified to a star schema. The paper explains the new approaches to access the data in the flat cubes, and shows how the bitmap indexes and the star-transformation are replaced by new, better performing functionality. At the end I will show actual customer queries using the new functionality and explain the performance benefits.

Replacing the Bitmap Indexes
The fact tables in the new flat cubes store a lot more data than in the traditional data cubes. The dimension tables are folded into the fact table. The fact tables have therefore a lot more key fields, which can no longer be indexed, as there are just too many. With Inmemory columnar storage this is no problem at all, as Inmemory scans data in a highly optimized way, which is explained at many other presentations.

Replacing the Star Transformation
The new functionality to replace the star-transformation is a little bit more complex. I will therefore show a series of queries that begin with a single join between a fact table, EUDFR2015, and a dimension table CUST_SALES and work up to a multi-table join.

Let's begin with the simplest join query, a join between the fact table EUDFR2015 and the dimension table CUST_SALES. This is a "What if" style query that calculates the amount of revenue increase that would have resulted from eliminating certain company-wide discounts in a given percentage range for products shipped on a given day (Christmas eve).

```
SELECT "X19"."S__CUST_H02" AS "S____223"
FROM "/BIC/8EUDFR2015" "F"
JOIN "/BI0/XCUST_SALES" "X19"
  ON  "F" . "SID_0CUST_SALES" = "X19" . "SID"
WHERE "X19"."SALESORG" BETWEEN 'IE00' AND 'IE09'
AND   "F"."SID_DIVREP" = 2
GROUP BY "X19"."S__CUST_H02"
```

The Optimizer has three join methods it could choose for this query;

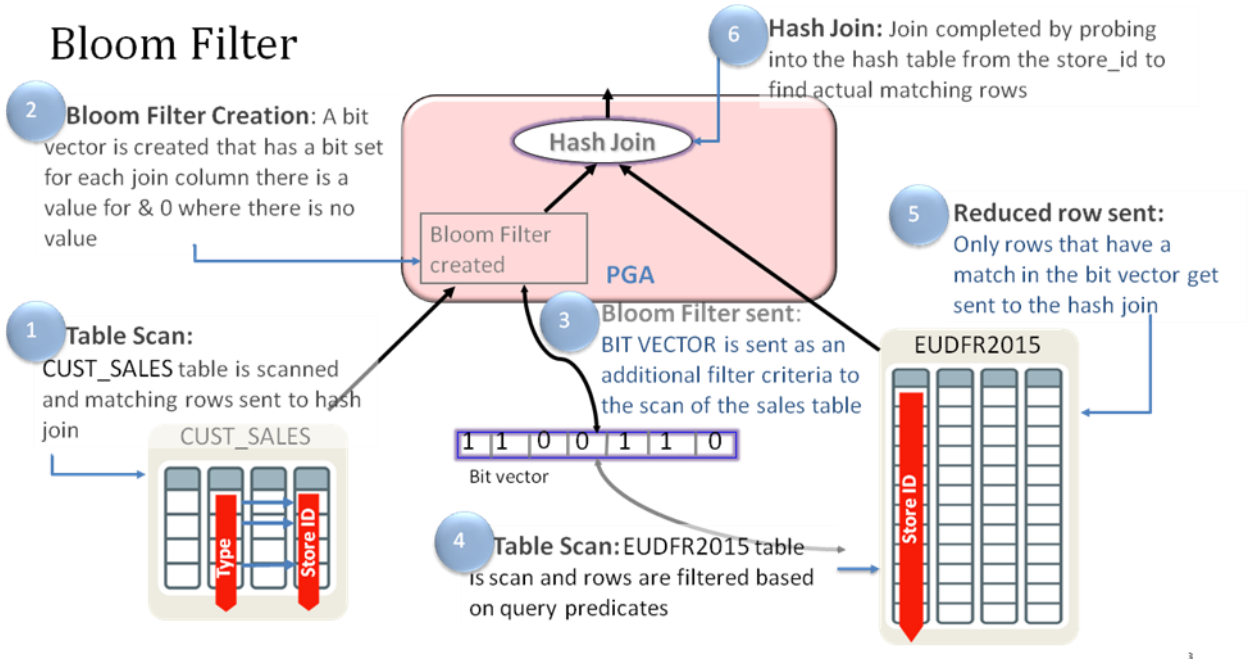
- Nest Loops
- Hash Join
- Sort Merge Join

Since there are no indexes on our tables the Optimizer picks a Hash Join. When two tables are joined via a Hash Join, the first table (typically the smaller table) is scanned and the rows that satisfy the 'where' clause predicates (for that table) are used to create a hash table, based on the join key, in memory. Then the larger table is scanned and for the rows that satisfy the 'where' clause predicates (for that table) the same hashing algorithm is performed on the join column(s). It then probes the previously built hash table for each value and if they match, it returns a row.

The In-Memory column store (IM column store) has no problem executing a query with a Hash Join efficiently because it is able to take advantage of Bloom Filters. A bloom filter transforms a join to a filter that can be applied as part of the scan of the fact table. Bloom filters were originally introduced in Oracle Database 10g to enhance hash join performance and are not specific to Oracle Database In-Memory. However, they are very efficiently applied to columnar data via SIMD vector processing.

So how does it work?

Join Filters
During the hash table creation, a bit vector or bloom filter is also created based on the join column. The bit vector is then sent as an additional predicate to the second table scan. After the where clause predicates have been applied to the second table scan, the resulting rows will have their join column hashed and it will be compared to values in the bit vector. If a match is found in the bit vector that row will be sent to the hash join. If no match is found then the row will be discarded.



Pic. 1: Bloom Filters

It's easy to identify a bloom filter in the execution plan. It appears in two places, at creation time and again when it is applied. Below is the plan for our query with the Bloom filter highlighted. The first step executed in this plan is actually line 4; a full table scan of the CUST_SALES table. The Bloom filter (:BF0000) is created immediately after the scan of the CUST_SALES table completes (line 3). The Bloom filter is then applied as part of the in-memory full table scan of the EUDFR2015 table (line

5 & 6). You can also see what join condition was used to build the bloom filter by looking at the predicate information under the plan. Look for 'SYS_OP_BLOOM_FILTER' in the filter predicates.

```
-----------------------------------------------------------
| Id  | Operation                     | Name              |
-----------------------------------------------------------
|   0 | SELECT STATEMENT              |                   |
|   1 |  HASH GROUP BY                |                   |
|*  2 |   HASH JOIN SEMI              |                   |
|   3 |    JOIN FILTER CREATE         | :BF0000           |←
|*  4 |     TABLE ACCESS FULL         | /BI0/XCUST_SALES  |
|   5 |    JOIN FILTER USE            | :BF0000           |←
|*  6 |     TABLE ACCESS INMEMORY FULL| /BIC/8EUDFR2015   |
-----------------------------------------------------------

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

   2 - access("F"."SID_0CUST_SALES"="X19"."SID")
   4 - filter("X19"."SALESORG"<='IE09' AND "X19"."SALESORG">='IE00')
   6 - inmemory("F"."SID_DIVREP"=2 AND
              SYS_OP_BLOOM_FILTER(:BF0000,"F"."SID_0CUST_SALES"))
```

Pic. 2: Access plan for a Two Table Join

Let's now try a more complex query that encompasses three joins and an aggregation of a lot more data.

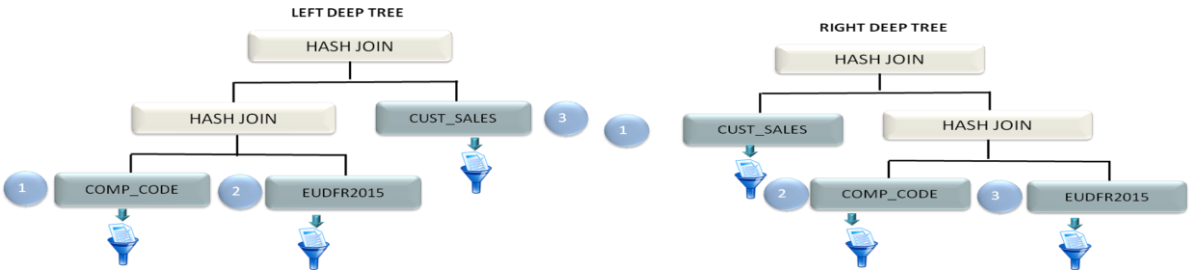
```
SELECT "X19"."S__CUST_H02" AS "S____223"
FROM "/BIC/8EUDFR2015" "F"
JOIN "/BI0/XCUST_SALES" "X19"
  ON "F" . "SID_0CUST_SALES" = "X19" . "SID"
JOIN "/BI0/XCOMP_CODE" "X20"
  ON "F" . "SID_0COMP_CODE" = "X20" . "SID"
JOIN "/BI0/SSALESORG" "S8"
  ON "F" . "SID_0SALESORG" = "S8" . "SID"
WHERE "X20"."S__CDC_ENT" = 41
AND   ( "S8"."SALESORG" BETWEEN 'GB00' AND 'GB99' AND
        "X19"."SALESORG" BETWEEN 'GB00' AND 'GB99'
     OR "S8"."SALESORG" BETWEEN 'IE00' AND 'IE99' )
AND   "X19"."SALESORG" BETWEEN 'IE00' AND 'IE99'
AND   "F"."SID_DIVREP" = 2
GROUP BY "X19"."S__CUST_H02"
```

If you examine the execution plan for this SQL statement (below) you will see that two bloom filters were created. The bloom filters are created on the SID_0CUST_SALES and the SID_0COMP_CODE column of the EUDFR2015 table. The Oracle Database is not limited to just one bloom filter per scan. It is possible to apply multiple bloom filters on a single table scan when appropriate.

```
-----------------------------------------------------------------------
| Id  | Operation                                | Name              |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                   |
|   1 |  HASH GROUP BY                           |                   |
|*  2 |   HASH JOIN RIGHT SEMI                   |                   |
|   3 |    TABLE ACCESS BY INDEX ROWID BATCHED| /BI0/SSALESORG   |
|*  4 |     INDEX RANGE SCAN                     | /BI0/SSALESORG~0 |
|*  5 |    HASH JOIN RIGHT SEMI                  |                   |
|   6 |     JOIN FILTER CREATE                   | :BF0000           |<-
|*  7 |      TABLE ACCESS FULL                   | /BI0/XCOMP_CODE   |
|*  8 |     HASH JOIN                            |                   |
|   9 |      JOIN FILTER CREATE                  | :BF0001           |<-
|* 10 |       TABLE ACCESS FULL                  | /BI0/XCUST_SALES  |
|  11 |      JOIN FILTER USE                     | :BF0000           |<-
|  12 |       JOIN FILTER USE                    | :BF0001           |<-
|* 13 |        TABLE ACCESS INMEMORY FULL        | /BIC/8EUDFR2015   |
```

Pic. 3: Accessplan for a Four Table Join

But how is Oracle able to apply two join filters when the join order would imply that the EUDFR2015 is accessed before the CUST_SALES table?



Pic. 4: Left deep and right deep tree

This is where Oracle's 30 years of database innovation kick in. By embedding the column store into the Oracle Database we can take advantage of all of the optimizations that have been added to the database over the last 11 releases. In this case, the Optimizer has switched from its typically left deep tree execution plan (shown above) to create a right deep tree using an optimization called 'swap_join_inputs'. So, instead of the EUDFR2015 table being the second table accessed in the plan, it actually becomes the third.

This allows multiple bloom filters to be generated before we scan the necessary columns for the EUDFR2015 table, meaning we are able to benefit by eliminating rows during the scan rather than waiting for the join to do it.

Multiple bloom filters replace the star transformation as they are able to restrict the fact table before the joins are actually executed.

Group By and Aggregation push down

Push-down of predicates and bloom filters is one of the optimizations that make scanning columns in the IM column store very efficient. Oracle Database In-Memory has the ability to push also aggregations and group-bys down into the scan of a column or columns. This ability to push-down

allows us to take advantage of other performance enhancing features of Database In-Memory like SIMD vector processing.

Let's take a look at a very simple BW query and see if we can find out what's happening. We'll use the following query:

```
SELECT  "S21"."MAT_SALES" AS "0MAT_SALES", SUM ("F"."/BIC/INV_QTY_T")
FROM "/BIC/8LADFRGACT" "F"
JOIN "/BI0/SMAT_SALES" "S21"
 ON  "F"."SID_0MAT_SALES" = "S21"."SID"
GROUP BY "S21"."MAT_SALES"
```

Pic. 5: Grouping BW query

To make it easier to explain we remove also the join (we will come back to this later):

```
SELECT  "F"."SID_0MAT_SALES", SUM ("F"."/BIC/INV_QTY_T")
FROM "/BIC/8LADFRGACT" "F"
GROUP BY "F"."SID_0MAT_SALES"
```

Pic. 6: Simple grouping BW query

The access plan we get is the following:

```
-------------------------------------------------------------------
| Id  | Operation                     | Name            | Rows  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                 | 46336 |
|   1 |  HASH GROUP BY                |                 | 46336 |
|   2 |   TABLE ACCESS INMEMORY FULL  | /BIC/8LADFRGACT |    93M|
-------------------------------------------------------------------
```

Pic. 7: Access plan for simple grouping BW query

In order to do a meaningful comparison we will use the IM session statistics to verify what is happening. In particular, the statistic "IM scan rows projected", which indicates the number of rows being returned from the IM column store. Two other statistics are also useful for this investigation, "IM scan rows", which is the total number of rows in all of the In-Memory Compression Units (IMCU) that were accessed, and "IM scan CUs memcompress for query low", which is the total number of IMCUs that were touched.

So let's take a look at what happens when the query is run in sqlplus. The session level statistics are listed by running the script imstats.sql which is simply a join between v$statname and v$mystat that selects any statistic that starts with 'IM '.

```
NAME                                                  VALUE
---------------------------------------------------- -----------
IM scan CUs no cleanout                                      227
IM scan CUs current                                          227
IM scan CUs readlist creation accumulated time               367
IM scan CUs readlist creation number                         227
IM scan CUs pcode aggregation pushdown                       227
IM scan rows pcode aggregated                           93541477
IM scan CUs memcompress for query low                        227 ←
IM scan bytes in-memory                                4277889507
IM scan bytes uncompressed                            46834367933
IM scan CUs columns accessed                                 454
IM scan CUs columns theoretical max                        10215
IM scan rows                                            93541477 ←
IM scan rows valid                                      93541477
IM scan rows projected                                   1147304 ←
IM scan CUs split pieces                                     232
```

Pic. 8: Inmemory statistics

From the statistic output we see that the query accessed 227 IMCUs (IM scan CUs memcompress for query low) and a total of 93,541,477 rows in those 227 IMCUs (IM scan rows), but only "returned" 1,147,304 rows (IM scan rows projected) from the IM column store. So how does Oracle return the correct aggregations when it only returns 1,147,304 rows from the query? The optimization that is happening is that Database In-Memory is returning the aggregations at the IMCU level back to the query layer. Most of the aggregation work occurs during the actual scan of the IMCUs and this is where considerable time is saved, especially when scanning very large objects. The query layer has much less work to do to complete the aggregation (i.e. over 93 million rows versus just a bit more than 1 million rows).

Group By Placement

The last optimization I like to show is easily visible with the original statement:

```
SELECT  "S21"."MAT_SALES" AS "0MAT_SALES", SUM ("F"."/BIC/INV_QTY_T")
FROM "/BIC/8LADFRGACT" "F"
JOIN "/BI0/SMAT_SALES" "S21"
 ON  "F"."SID_0MAT_SALES" = "S21"."SID"
GROUP BY "S21"."MAT_SALES"
```

Pic. 9: Grouping BW query

The grouping is in this query not defined on the fact table and would expect to get the aggregation and group-by not pushed down into the scan of a column.

But when we look at the plan, we see the group by pushed down:

```
-------------------------------------------------------------------------
| Id  | Operation                          | Name            | Rows  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                 | 46336 |
|   1 |  HASH GROUP BY                     |                 | 46336 |
|*  2 |   HASH JOIN                        |                 | 46336 |
|   3 |    JOIN FILTER CREATE              | :BF0000         | 46336 |
|   4 |     VIEW                           | VW_GBC_5        | 46336 |
|   5 |      HASH GROUP BY                 |                 | 46336 | ←
|   6 |       TABLE ACCESS INMEMORY FULL   | /BIC/8LADFRGACT |    93M|
|   7 |    JOIN FILTER USE                 | :BF0000         |  2609K|
|*  8 |     TABLE ACCESS INMEMORY FULL     | /BI0/SMAT_SALES |  2609K|
-------------------------------------------------------------------------
```

Pic. 10: Access plan for grouping BW query

We got an additional group by on the fact table. This feature is called Group By Placement (GBP)

In this case we have aggregated data from LADFRGACT before joining to MAT_SALES because the optimizer has decided the volume selected from LADFRGACT is so large that aggregating before joining would be cheaper than joining before aggregating. Even with the additional GROUP BY on step 1 the cost stays lower than without the GBP. GBP enables additionally the push down of the group by and aggregation into the scan of the IMCUs of the table LADFRGACT. This results in additionally performance increase.

Summary
Flatcubes cannot only replace the old star cubes with equivalent functionality, but enable also the use of advanced functionality to improve the performance.

Kontaktadresse:
Jörn Bartels
Oracle Deutschland
Riesstraße 25
D-82275 München

Telefon: +49 (0) 89-1430 1120
E-Mail joern.bartels@oracle.de
Internet: www.oracle.com