Turbocharging Data Visualization & Analyses
With Oracle In-Memory 12.2
Dr.-Ing. Holger Friedrich
Turbocharging Data Visualization and Analyses with Oracle In-Memory 12.2

Dr.-Ing. Holger Friedrich, CTO
sumIT AG
• Introduction
• Analytic Workloads & Requirements
• Oracle In-Memory (12.2)
• Conclusions
• Consulting and implementation services in Switzerland
• Experts for
  - Data Warehousing,
  - Business Intelligence,
  - Advanced Analytics
• Focussed on Oracle technology
• Oracle specialisations for BI Foundation, Database, Data Warehousing, and Data Integration
• Our motto: Get Value From Data
• Visit our web site: www.sumit.ch (in German)
Holger Friedrich

- Computer Science diploma of Karlsruhe Institute of Technology (KIT)
- Ph.D. in Robotics and Machine Learning
- More than 17 years experience with Oracle technology
- Expert for
  - Data Integration
  - Data Warehousing / Big Data,
  - Advanced Analytics and
  - Business Intelligence
- Technical Director of sumIT AG
- Oracle ACE for DWH/BI

© 2017 sumIT AG
On-Premises
Business Intelligence
Advanced Analytics
Machine Learning
Spatial Analytics
Graph Analytics
Data Mining
CMS Experiment at the LHC, CERN

Run / Event: 150431 / 541464

Data Visualisation
(Non-Streaming) Workload Characteristics

- Scans of large amounts of data
- Apply functions to single columns
- Compute complex expressions on multiple attributes,
- Aggregate data
- Join data sets through known attribute pairs
- Flexibly join data sets through arbitrary attribute pairs
- Data sets do range from small to very large
- Query sensitive data sets
- Query new data sets (schema on read, JSON, etc.)

- All of the above with high speed
Columnar Databases

- Best for queries that
  - scan large quantities of data
  - compute aggregates on the results

- Efficient storage use, due to high compression benefits on most columns
Features Around Since First Release

1. In-memory storage indexes
2. Filtering on binary compressed data
3. Columnar storage of selected columns
4. Transparent querying across storage hierarchy
5. Real-time background synching of columnar store
6. Parallel query execution on the columnar store
7. SIMD vector processing
8. In-memory fault tolerance on RAC
9. In-memory aggregation
SELECT /*+ no_parallel(t) no_parallel_index(t) dbms_stats cursor_sharing_exact
       use_weak_name_resl dynamic_sampling(0) no_monitoring
       xmlindex_sel_idx_tbl no_substrb_pad */
       COUNT(*),
       SUM(sys_op_opnsize("PROD_ID")),
       SUM(sys_op_opnsize("CUST_ID")),
       COUNT(DISTINCT "CHANNEL_ID"),
       SUM(sys_op_opnsize("CHANNEL_ID")),
       substrb(dump(MIN("CHANNEL_ID"),16,0,64),1,240),
       substrb(dump(MAX("CHANNEL_ID"),16,0,64),1,240),
       COUNT(DISTINCT "PROMO_ID"),
       SUM(sys_op_opnsize("PROMO_ID")),
       substrb(dump(MIN("PROMO_ID"),16,0,64),1,240),
       substrb(dump(MAX("PROMO_ID"),16,0,64),1,240),
       COUNT(DISTINCT "QUANTITY_SOLD"),
       SUM(sys_op_opnsize("QUANTITY_SOLD")),
       substrb(dump(MIN("QUANTITY_SOLD"),16,0,64),1,240),
       substrb(dump(MAX("QUANTITY_SOLD"),16,0,64),1,240),
       SUM(sys_op_opnsize("AMOUNT_SOLD"))
FROM "SH"."MY_SALES" t;
### In-Memory

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>24</td>
<td>1070 (27)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>SORT GROUP BY</td>
<td></td>
<td>1</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS INMEMORY FULL</td>
<td>MY_SALES</td>
<td>14M</td>
<td>336M</td>
<td>1070 (27)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

1. In-Memory chosen automatically

2. Switched to row storage SGA, Flash, or Disk

3. Huge cost advantage due to IM

### With explicit NO_INMEMORY Hint

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>24</td>
<td>19350 (1)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>SORT GROUP BY</td>
<td></td>
<td>1</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>MY_SALES</td>
<td>14M</td>
<td>336M</td>
<td>19350 (1)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

2. Switched to row storage SGA, Flash, or Disk

3. Huge cost advantage due to IM
Flexible Joins in 12.1

- Join tables on flexible criteria
- No special model required, e.g. star schema
- No indices required
- Speed up joins by use of bloom filters (not on all occasions)
- Small hash joins in PGA
- However data has to be decompressed for
  - bloom filter construction
  - hashing
What’s New in 12.2?

- **Real-Time Analytics**
  - 2X Faster Joins
  - 5X Faster Expressions

- **Mixed Workload**
  - Active Data Guard Support

- **Massive Capacity**
  - In-Memory on Exadata Flash

- **Multi-model**
  - Native support for JSON Data type

- **Automation**
  - Dynamic Data Movement Between Storage & Memory
• Join columns in both tables are compressed using the same dictionary
• Joins occur on dictionary values rather than on data
  - Saves on decompression of data
  - Save on hashing the data

Example: Find sales price of each Vehicle
Local Dictionary at CU level

- Contiguous storage per column in an IMCU
- All CUs automatically store Min/Max values
- Multiple formats: depends on data and chosen compression level
- Most CUs have a “Dictionary”
  - Sorted list of distinct values in the CU
  - Column values replaced with dictionary IDs
- Additional compression of value list possible (e.g. Run Length Encoding, OZIP)
Global Dictionary

<table>
<thead>
<tr>
<th>NAME</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUDI</td>
<td>0</td>
</tr>
<tr>
<td>BMW</td>
<td>1</td>
</tr>
<tr>
<td>CADILLAC</td>
<td>2</td>
</tr>
<tr>
<td>PORSCHE</td>
<td>3</td>
</tr>
<tr>
<td>TESLA</td>
<td>4</td>
</tr>
<tr>
<td>VW</td>
<td>5</td>
</tr>
</tbody>
</table>

A global dictionary is created:
• Contains all distinct values for the column in the table
• All CUs use a single Dictionary

- Global Dictionary created when 1st table is populated & used for join column in both tables
Hash Join With Join Group

2. Table Array of compressed values:
   An array of distinct values is created from the compressed values

3. Table access inmemory full:
   SALES table is scanned and data in the name column is sent to the hash join in its compressed format

4. Hash Join: Join completed by looking up compressed values in the array

1. Table access inmemory full:
   VEHICLES table is scanned and join column values sent to join in compressed format
CREATE INMEMORY JOIN GROUP vehicle_ales_jg
(VEHICLES(NAME), SALES(NAME));

- Create Join Group
- Populate tables into column store
- Make sure that the global dictionary exists
- Use dictionary view user_joingroup to monitor

```
SELECT o.object_name table_name,
       c.column_name column_name,
       gd.head_address "GD Address"
FROM user_objects o,
     user_tab_columns c,
     v$im_segdict gd
WHERE gd.objn = o.object_id
AND   o.object_name = c.table_name
AND   gd.column_number = c.column_id;
```
Managing and Monitoring (II)

Table Statistics:
- Plan Hash Value: 2520514737
- Line ID: 0
  - Operation: SELECT STATEMENT
- Line ID: 1
  - Operation: SORT AGGREGATE
- Line ID: 2
  - Operation: HASH JOIN
- Line ID: 3
  - Operation: TABLE ACCESS FULL
- Line ID: 4
  - Operation: PARTITION RANGE ALL
- Line ID: 5
  - Operation: TABLE ACCESS INMEMORY FULL

Other Plan Line Statistics:
- Build Size: 983K
- Build Row Count: 380
- Fan-out: 6
- Slot Size: 123K
- Total Build Partitions: 8
- Total Cached Partitions: 8
- Columnar Encodings Leveraged: 1

Time & Wait Statistics:
- Duration: 5.0s
- Database Time: 4.8s
- PL/SQL & Java: 0s
- Activity %: 100
In-Memory Virtual Columns

• Virtual Columns
  - do represent expressions involving columns in a table
  - are computed based on values within a record of a table
  - are not represented on disk
  - can be queried like any regular columns

• Syntax: Alter table Lineorder
  Add (net_price AS
  (l_extendedprice * (1 - l_discount)));
Rationale For In-Memory Virtual Columns

- Evaluating expressions repeatedly can be computationally expensive
- Repeated expressions are typical for analytic queries

```sql
select  l_returnflag, l_linestatus, sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as charge,
        avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from    lineorder
where   l_shipdate <= to_date ('1998-12-01','YYYY-MM-DD') - 90
group by l_returnflag,
         l_linestatus
order by l_returnflag, l_linestatus;
```
• Exist only in columnar store
• All performance features of In-Memory option are supported
  - IMCU pruning through storage indexes
  - SIMD
  - ...

Select `sum_disc_price`
From `Sales`
Where `region` = 'CA';
In-Memory Expressions

- Often used expressions that are automatically captured
- Capturing based on the Expression Statistics Store
- The database determines 'hot' expressions autonomously
- Top 50 expressions are added as virtual IM columns
- Populated automatically during next (re-)population of columnar store
SELECT employee_id, last_name, salary, commission_pct,
       ROUND(salary*12/52,2) as "weekly_sal",
       12*(salary*NVL(commission_pct,0)+salary) as "ann_comp"
FROM   employees
ORDER BY ann_comp;
12.2 - Massive Storage Capacity

- In-Memory columnar format in flash cache
- Massive extension of storage capacity for the In-memory option
Use Of The Available Flash Cache

- New smart columnar flash cache format
- Available on Exadata
- Additional optimisations, e.g. evaluation of multiple column values in single vector instruction
- Optimiser spreads queries across DRAM & flash
- Extends available storage by factor 10
- Currently unique to Oracle In-Memory
- HCC objects are automatically populated into flash in IM-Columnar format
12.2 - Accelerate JSON Access

- Make use of JSON binary representation in the columnar store
- Support for multiple JSON functions since 12.1
  - `json_table`
  - `json_value`
  - `json_exists`
- In-Memory Expressions speed up JSON analytics further in IM 12.2
Conclusions

• Today’s variety of analytical tools and tasks generate
  - more diverse workloads
  - require a very fast, flexible, agile, and robust query engine
• Oracle In-Memory boosts many of today’s analytical workloads
• The new 12.2 features
  - In-memory Expressions
  - Join Groups
  - JSON optimisations
  - In-Memory Columnar Storage in Flash
  provide unprecedented performance
• Big Data SQL extends its use and integrates it with HADOOP clusters
BIWA SUMMIT 2017
WITH SPATIAL SUMMIT

THE Big Data + Analytics + Spatial + Cloud + IoT + Everything Cool User Conference
January 31 - February 2, 2017