The Design and Implementation of In-Memory Databases

Shasank Chavan
Vice President, In-Memory Database Technologies Group
Safe Harbor

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.

Statements in this presentation relating to Oracle’s future plans, expectations, beliefs, intentions and prospects are “forward-looking statements” and are subject to material risks and uncertainties. A detailed discussion of these factors and other risks that affect our business is contained in Oracle’s Securities and Exchange Commission (SEC) filings, including our most recent reports on Form 10-K and Form 10-Q under the heading “Risk Factors.” These filings are available on the SEC's website or on Oracle’s website at http://www.oracle.com/investor. All information in this presentation is current as of September 2019 and Oracle undertakes no duty to update any statement in light of new information or future events.
Agenda

Motivation for In-Memory Databases: Real-Time Analytics
Hardware Trends Enabling In-Memory Processing
Row Store for Transactions, Column Store for Analytics
Columnar Data Formats and Compression Techniques
In-Memory SQL Operators
Data Reorganization
Specialized Database Machines

What's Next?
What is a Real-Time Enterprise?

- **Data Driven** – Uses real-time insights derived from current data
- **Agile** – Rapidly adjusts to changes
- **Efficient** – Continuously adjusts processes to maximize profitability

Copyright © 2019 Oracle and/or its affiliates.
Many Examples of **Real-Time Enterprises**

- **Insurance companies** improve portfolios and reduce cost with real-time analytics for pricing
- **Retailers** use location-based analytics to automate sending personalized mobile coupons to customers
- **Manufacturing Processes** use real-time analytics to monitor production quality and adjust assembly parameters
- **Financial Services** perform risk/fraud analysis across channels in real-time, not after the event occurs
- **Telecom and Broadband** vendors use real-time congestion metrics to optimize their networks
In-Memory Database Technology Facts
The Time for In-Memory is NOW

- Next Generation Enterprises must be real-time
  - In-Memory enables 10-100x performance gains
  - In-Memory is essential for real-time processing
  - In-Memory is mandatory in next-gen platforms

- Current memory sizes enable In-Memory today
  - Oracle X8-2: Up to 1.5TB of DRAM
  - In-Memory is no longer a special niche!
  - Not using in-memory like using a rotary phone or dial-up internet or Windows 98 … in 2019

Copyright © 2019 Oracle and/or its affiliates.
In-Memory Technologies Across Tiers

**Application-Tier**
- TimesTen-In-Memory Database
  - Latency Critical OLTP Applications
  - Microsecond response time
  - Standalone or Cache for Oracle Database

**Database-Tier**
- Database In-Memory
  - Dual Format In-Memory Database
  - Billions of Rows/sec analytic data processing
  - 2-3x Faster Mixed Workloads

**Storage-Tier**
- In-Memory on Exadata Storage
  - In-memory format on Exadata Flash Cache
  - 5-10x faster smart scan in storage
  - 15x increase in total columnar capacity
Hardware Trends

- Larger, Cheaper Memory (DRAM, PMEM)
- Larger CPU Caches (e.g. 1MB L2 Cache, 32MB Shared L3 Cache)
- Larger Multi-Core Processors (24 cores w/ Intel Cascade Lake)
- Larger Vector Processing Units (Intel AVX-512 has 512-bit SIMD registers)
- Faster Networks (100Gb/s RoCE vs 40Gb/s Infiniband)
- Persistent Memory (Game Changer for Data Management systems)
- NUMA Architectures (Local Memory faster than Remote)
- Distributed Systems for High Availability and Scalability
Row Store for Transactions, Column Store for Analytics
In-Memory Row Format: Slower for Analytics

- Transactions run faster on row format
  - Example: Insert or query a sales order
  - Fast processing for few rows, many columns

Buffer Cache

Row Format

SELECT COL4 FROM MYTABLE;
In-Memory Columnar Format: Faster for Analytics

- Analytics run faster on column format
  - Example: Report on sales totals by region
  - Fast accessing few columns, many rows

IM Column Store

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
<th>COL3</th>
<th>COL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

SELECT **COL4** FROM MYTABLE;

RESULT

Scans only the data required by the query
Row Format vs. Column Format Databases

Row Format

- **Transactions** run faster on row format
  - Example: Insert or query a sales order
  - Fast processing for few rows, many columns

Column Format

- **Analytics** run faster on column format
  - Example: Report on sales totals by region
  - Fast accessing few columns, many rows

Until Now Must Choose One Format and Suffer Tradeoffs
In-Memory Database Flavors

- Entire database must fit in memory (HANA)
- Column-store is an index on disk, loaded into memory (MSFT)
- In-Memory column-store embedded in application (TimesTen)
- Writes limited to tables that fit in memory (Memsql)
- No scale out support (MSFT)
- Hybrid (Oracle)
Database In-Memory: Optimal for Analytics and OLTP

- **Both** row and column format for same table
  - Simultaneously active and consistent
- OLTP uses existing row format
- Analytics uses In-Memory Column format
  - **Seamlessly** built into Oracle Database
  - All enterprise features work - RAC, Dataguard, Flashback, etc.
Accelerates Mixed Workloads (Hybrid OLTP)

- Inserting one row into a table requires updating 10-20 analytic indexes: **Slow**!
- Fast analytics **only on** indexed columns
- Analytic indexes **increase** database size

- Column Store not persistent so updates are: **Fast**!
- Fast analytics on **any** columns
- No analytic indexes: **Reduces** database size
Oracle Database In-Memory: How?

1. DRAM
   - SALES
   - Column Format

2. Memory
   - CPU
     - SIMD
     - Load multiple values
     - Vector Register
     - SIMD Compare Multiple values

3. DRAM
Oracle In-Memory Columnar Format – Under The Hood

In-Memory Compression Unit (IMCU)
- Unit of column store allocation
- Spans large number of rows (e.g. 0.5 million) on one or more table extents
- Each column stored as Column Compression Unit (column CU)

Multiple MEMCOMPRESS levels:
- FOR QUERY – fastest queries
- FOR CAPACITY – best compression
Improves All Aspects of Analytic Workloads ...

**Scans**

**Joins**

**Aggregations**

- **SALES**
  - **STATE = CA**

- **HASH JOIN**
  - **ITEMS**
  - **SALES**

- **Graphs**
Columnar Compression: Dictionary Encoding

- Dictionary Encoding is heavily used light-weight compressor with In-Memory Databases
- It gives excellent compression without requiring data to be decoded / decompressed for many operations
- Dictionary Encoding format:
  - Maintain an order-preserving dictionary of distinct symbols in a column.
  - Map those symbols to codes [0..N]
  - Replace symbols in column with the codes
- For additional compression, bit-pack the codes in the value stream (log(N) bits needed) and similarly pack symbols in dictionary.
- For more compression, run-length encode the value stream
In-Memory Compression

Uncompressed Data
- CAT
- CAT
- FISH
- FISH
- HORSE
- HORSE
- HORSE
- DOG
- DOG
- CAT
- CAT
- FISH
- HORSE
- HORSE
- DOG
- DOG

Dictionary Compressed
- CAT 0
- DOG 1
- FISH 2
- HORSE 3
- 00, 00, 10, 10
- 11, 11, 11, 01
- 01, 00, 00, 10
- 11, 11, 01, 01

Dict + RLE Compressed
- CAT 0
- DOG 1
- FISH 2
- HORSE 3
- Runs: 2,2,3,2,2,1,2,2
- 00, 10, 11, 01
- 01, 00, 10, 11
- 01

OZIP Compressed
- 0: 00101101
- 1: 01
- 010
Population Structures for Dictionary Encoding

- Hash Table
- CSB Tree
  - Up to 32 key prefixes
    - (8-byte each)
  - Up to 32 key pointers
- Radix Tree

Copyright © 2019 Oracle and/or its affiliates.
## Compression: Prefix Encoding

### No Compression

<table>
<thead>
<tr>
<th>COLUMN CU</th>
<th>DICTIONARY</th>
<th>LENGTHS</th>
<th>VALUE</th>
</tr>
</thead>
</table>

### Prefix Compression

<table>
<thead>
<tr>
<th>COLUMN CU</th>
<th>PREFIX-ENCODED DICTIONARY</th>
<th>LENGTHS</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>[USE</td>
<td></td>
<td>D]</td>
<td>[USEFUL</td>
</tr>
</tbody>
</table>

- **USE**
- **USED**
- **USEFUL**
- **USEFULLY**
- **USEFULNESS**
- **USELESS**
- **USELESSLY**
- **USELESSNESS**
In-Memory Expressions

Example: Compute total sales price
   \[
   \text{Net} = \text{Price} + \text{Price} \times \text{Tax}
   \]

SQL expression results can also be stored as additional inmemory columns

All In-Memory optimizations apply to expression columns (e.g. Vector processing, storage indexes)

Two modes:

- **Manual**: Declare virtual columns for desired inmemory expressions
- **Auto**: Auto detect frequent expressions

```sql
CREATE TABLE SALES (
    PRICE NUMBER, TAX NUMBER, ..., 
    NET AS (PRICE + PRICE \times TAX)
) INMEMORY;
```
OSON : Binary JSON for In-Memory

- BSON is a binary representation of JSON that’s not good enough
- OSON (for Oracle) provides the following:
  - Key name dictionary to encode key names
  - Jump offset table to access value based on name
  - Jump offset table to access array element based on index
  - Partial update support even if change size is larger than original one
In-Memory Spatial Analytics

- In-Memory only Spatial Summary column added to each spatial column
  - Compact approximation of complex spatial detail
  - Stored in optimized In-Memory format
  - Quickly filter using SIMD vector scans
  - Replace R-Tree Indexes for Spatial Analytics
  - No analytic R-tree index maintenance needed

<table>
<thead>
<tr>
<th>Parcel Number</th>
<th>Parcel Address</th>
<th>Spatial Details</th>
<th>Spatial Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>095040390</td>
<td>300 Oracle Pkwy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>095040310</td>
<td>400 Oracle Pkwy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>095040250</td>
<td>500 Oracle Pkwy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>095040260</td>
<td>600 Oracle Pkwy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Which parcel is the utility valve located in?

Search 140 Million US land parcels
SQL Operations

DML
Scans
Aggregations
Joins
DML Operations

• DML (Deletes, Inserts, and Updates) can be expensive with column stores
  • Inserting an entry in the middle of a sorted dictionary?
  • Inserting a code into a middle of the code vector?
  • Re-coding values
  • Expanding code vector (need an extra bit, for example)
  • How does Locking work?

```
<table>
<thead>
<tr>
<th></th>
<th>CAT</th>
<th>DOG</th>
<th>FISH</th>
<th>HORSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Code</th>
<th>CAT</th>
<th>DOG</th>
<th>FISH</th>
<th>HORSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>00, 00, 10, 10</td>
<td>11, 11, 11, 01</td>
<td>01, 00, 00, 10</td>
<td>11, 11, 01, 01</td>
</tr>
</tbody>
</table>
```

Diagram:
- EEL
- CAT
- DOG
- FISH
- HORSE
- 00, 00, 10, 10
- 11, 11, 11, 01
- 01, 00, 00, 10
- 11, 11, 01, 01

Copyright © 2019 Oracle and/or its affiliates.
In-Memory: Dual-Format Architecture

- In-Memory works for Analytics and Mixed Workloads
- Fast In-Memory DML because invalid row is logically removed from column store (just set a bit)
- Analytic query will ignore invalid rows in column store, and just vector process valid rows. Invalid rows are then processed.
  - Each IMCU is self-contained and processed independently, such that IMCUs not covering invalid rows are completely unaffected.
- Mixed workload queries can perform slowly if the number of invalid rows builds up in IMCUs, but fast repopulation techniques save the day!
**Foundational Features : Fast Background Repopulation**

- *Continuous intelligence* to track how dirty an IMCU is
- *Continuous intelligence* to track how frequently the IMCU is scanned.
- Action take to refresh / repopulate IMCU based on threshold
- Repopulation is done quickly by using *double buffering* and *incremental repopulation* techniques
  - **Double Buffering**: New IMCU is built while Old IMCU continues to serve queries. A *switcheroo* happens once New IMCU is ready. This keeps in-memory columnar processing online.
  - **Incremental Repopulation**: CUs in New IMCU are constructed quickly using the meta-data already present in the Old IMCU, allowing for quick formatting like Dictionary Encoding.
- Repopulation happens in the background by worker slaves, which are scaled elastically by Resource Manager. This enables DML to have minimal impact in the foreground.
Fine-Grain Invalidations

• Column-Level Invalidations
  • Once a row becomes invalid in IMCU, it must be fetched from disk/buffer cache when queried.
  • Often times DML statements only update a particular set of columns only.
    • The “invalid” row prevents the IMCU from being used, and nullifying the benefits of columnar data
  
  **SOLUTION:**
  • Invalidated rows do not prevent IMCU from servicing query unless affected columns are referenced.
SQL Operations

Scans
In-Memory Enables SIMD Vector Processing

- Column format benefit: Need to access only needed columns
- Process multiple values with a single SIMD Vector Instruction
- Billions of rows/sec scan rate per CPU core
  - Row format is millions/sec

Example:
Find sales in State of California

> 100x Faster
SIMD : History / Background

- SIMD (Single Instruction Multiple Data) technology in microprocessors allows one instruction to process multiple data items at the same time.
- SIMD technology was first used in the late 60s and early 70s, as the basis for vector supercomputers (Cray, CDC Star-100, in the ILLIAC IV (1966))
- SIMD processing shifted from super-computer market to desktop market
  - Real-time graphics and gaming, audio/video processing (e.g. MPEG decoding), etc.
    - For example, “Change the brightness of an image” (i.e. add/sub value from R-G-B fields of pixel)
  - VIS (Sun Microsystems), MMX (Intel), AltiVec (IBM), SSE (Intel) – 1999
- Initial adoption of SIMD systems in personal computers was slow
  - FP registers reused. Conversions from FP to MMX registers. Lack of compiler support.
SIMD : Intrinsics (Intel)

- Instruction-level *intrinsics* are C-style functions representing assembly instruction(s) which compilers inline into the source program.
  - E.g. `__popcnt()` maps to the `popcnt` instruction which returns number of set bits in reg
- With SSE came 128-bit SIMD registers and rich set of intrinsics, which started making SIMD programming more main-stream for performance
- Intel’s compiler-intrinsics are available online via an interactive guide

<table>
<thead>
<tr>
<th>MMX (~124 intrinsics)</th>
<th>SSE4.2 (~624 intrinsics)</th>
<th>AVX2 (~997 intrinsics)</th>
<th>AVX-512 (~4864 intrinsics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cmpeq_pi*</td>
<td>_mm_blendv_epi*</td>
<td>_mm256_srlv_epi*</td>
<td>_mm512_conflict_epi*</td>
</tr>
<tr>
<td>_mm_srlv_pi*</td>
<td>_mm_max_epi*</td>
<td>_mm256_i32_gather_epi*</td>
<td>_mm512_mask_cmp_epi* _mask</td>
</tr>
<tr>
<td>_mm_add_pi*</td>
<td>_mm_shuffle_epi8</td>
<td>_mm256_permute4x64_epi64</td>
<td>_mm512_maskz_compress_epi*</td>
</tr>
</tbody>
</table>
SIMD: **Early Adoption For Data Processing: Scan Filters**

- Parallelize predicate evaluation – load, eval, store/consume result
- Select `count(*)` from T where `a > 10` and `b < 20`
  - `[Load] A`
  - `[Load] Temp = 10`
  - `[Compare] A > Temp`
  - Load B, Compare 20
  - And
  - Mask, Store Bit-Map

```plaintext
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>95182</td>
<td>1</td>
<td>69</td>
<td></td>
</tr>
</tbody>
</table>
```

**0101**
Scans: Predicate Filtering Techniques

**Selective Evaluation**
Evaluate predicates selectively given results of previous predicate
...where A=1 and B=C

**Dictionary Filtering**
Evaluate predicates selectively on dictionary given results of previous predicate
...where A<20 and BLOOM(X)

**Predicate Reordering**
Rearrange predicates based on selectivity and cost
...where B=C and A<20 and BLOOM(X)
*Can turn into...*
...where A<20 and BLOOM(X) and B=C
SQL Operations

Aggregations
In-Memory Aggregations

- Aggregations & Group By Ops are heavily used with analytics
- Group-By Aggregation (e.g. `select sum(A) from T group by x,y`) typically use Hash Tables
  - Grouping sum needed for each distinct combination of x and y.
  - Locking needed to prevent 2 threads updating into same bucket.
  - Hash Table manages size when dealing with sparse number of groups
- With In-Memory, leverage Dictionary Encoding
  - Aggregate on codes, not on symbols
  - Replace Hash Table with Array Index (max code is size of array)
SIMD: Group-By and Aggregation

- Group-By and Aggregation clauses are highly suitable for SIMD processing
  - Select a, b, \( \text{sum}(c) \) from T where <blah> group by a, b
- Hash Group-By is the most common implementation method.
  - Grouping key is hashed to index into Hash Table where results are accumulated.
- Array-based Aggregation is preferable when grouping key cardinality is relatively low and space is available.
  - Conflict detection is still required in order to parallelize aggregation across groups
- Partition-based Aggregation is also suitable – contention is avoided at the cost of slow partitioning.
SQL Operations

Joins
Bloom Filters are used in Hash Joins for fast filtering during inner table scan.

Keys from build table are hashed and bits are set in a (segmented) bitmap. A match is possible ONLY IF the join key from outer (probe) table has a hit in the bloom filter bitmap.

Bloom filter evaluation can be vectorized with SIMD using a combination of instructions – loads, shuffles, shifts, etc.

```
<table>
<thead>
<tr>
<th>Join Col</th>
<th>10</th>
<th>20</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
0xCF29D1E9
```

```
Mod (N=20)
```

```
00000001010000100011
```

```
10
```

```
00000001011000100011
```
Joins: Encoding Aware Hash Joins

- Leverage IM data encodings to reduce number of ops in HJ.
- <Reminder> In-Memory Columns are usually dictionary-encoded
  - Distinct values of a column in an IMCU are stored separately (sorted) and the original column values are replaced by codes ([0..N], where N is number of distinct symbols).
- Rows are projected back from table scan row-sources using Rowsets
  - A rowset basically has the following structure:
    - Struct rowset { char* value, short* len, short flags)
  - With In-Memory, Rowsets were amended to return back data still encoded (dict)
- SQL operators above Scans can now engage in late materialization techniques for improved performance.
Joins: Encoding Aware Hash Joins

- Late Materialization Techniques
  - Only dictionary codes from Probe Table are processed by HJ row-source
  - If code X has not been seen, fetch symbol to Hash, and then use to probe
    - Hashing happens only once per distinct symbol
    - Probe happens only once per distinct symbol.
- No redistribution of hash join’s probe table among parallel query slaves. Why?
  - IM dictionary codes can be preserved to propagate up query execution tree
- No optimizations across IMCU. Why?
  - The column in IMCU 1 has a separate Local dictionary than that in IMCU 2
  - Codes from IMCU 1 do *not* map to IMCU 2
New object called *Join Group* used to specify the join columns in tables which may be joined against.

Create Join Group `sales_vehicle_jg`  
(Vehicles(v_id), Sales(v_id));

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  
Join becomes simple index into an array

**Example**

“Find sales price of each vehicle”
Join Group: Common Dictionary Used By Both Tables, In All IMCUUs

Common 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>

Common Dictionary created when first table is populated & used for join column in both tables – Must be defined with **CREATE INMEMORY JOIN GROUP** syntax
Join Processing and Aggregation Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

```
SELECT food.category, geography.state, sum(sales.amt)
FROM sales, food, geography
WHERE sales.f_id = food.f_id
AND sales.g_id = geog.g_id
AND food.category != 'Vegetable'
AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state
```
Join Processing and Aggregation
Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

SELECT food.category, geography.state, \textit{sum(sales.\textit{amt})}
FROM sales, food, geography
WHERE sales.f_id = food.f_id
AND sales.g_id = geography.g_id
AND food.category != 'Vegetable'
AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state

Scan & Filter
Join Processing and Aggregation
Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

```
SELECT food.category, geography.state, sum(sales.amt)
FROM sales, food, geography
WHERE sales.f_id = food.f_id
AND sales.g_id = geography.g_id
AND food.category != 'Vegetable'
AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state
```
Join Processing and Aggregation Overview

Calculate total *sales* of *non-veggies* from *WA* and *OR*, *group by* food category and state

```
SELECT food.category, geography.state, sum(sales.amt)
FROM sales, food, geography
WHERE sales.f_id = food.f_id
AND sales.g_id = geog.g_id
AND food.category != 'Vegetable'
AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state
```
Join Processing and Aggregation Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

```
SELECT food.category, geography.state, sum(sales.amt)
FROM sales, food, geography
WHERE sales.f_id = food.f_id
AND sales.g_id = geog.g_id
AND food.category != 'Vegetable'
AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state
```
Join Processing and Aggregation Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

SELECT food.category, geography.state, \text{sum(sales.amt)}
FROM sales, food, geography
WHERE \text{sales.f_id} = \text{food.f_id}
AND \text{sales.g_id} = \text{geog.g_id}
AND \text{food.category} \neq 'Vegetable'
AND \text{geography.state} \text{IN (}'WA', 'OR'\text{)}
GROUP BY \text{food.category, geography.state}
Join Processing and Aggregation Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

```
SELECT food.category, geography.state, sum(sales.amt)
FROM sales, food, geography
WHERE sales.f_id = food.f_id
  AND sales.g_id = geography.g_id
  AND food.category != 'Vegetable'
  AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state
```
Join Processing and Aggregation
Overview

Calculate total sales of non-eggies from WA and OR, group by food category and state

```
SELECT food.category, geography.state, sum(sales.amt)
FROM sales, food, geography
WHERE sales.f_id = food.f_id
AND sales.g_id = geography.g_id
AND food.category != 'Vegetable'
AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state
```
Join Processing and Aggregation Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

\[
\text{SELECT food.category, geography.state, } \sum(\text{sales.amt}) \\
\text{FROM sales, food, geography} \\
\text{WHERE sales.f_id = food.f_id} \\
\text{AND sales.g_id = geography.g_id} \\
\text{AND food.category \neq 'Vegetable'} \\
\text{AND geography.state IN ('WA', 'OR')} \\
\text{GROUP BY food.category, geography.state}
\]
Join Processing and Aggregation
Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

```sql
SELECT food.category, geography.state, sum(sales.amt)
FROM sales, food, geography
WHERE sales.f_id = food.f_id
AND sales.g_id = geography.g_id
AND food.category != 'Vegetable'
AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state
```

Probe Hash Table

Select food category, geography state, sum(sales amt)
From sales, food, geography
Where sales f_id = food f_id
And sales g_id = geography g_id
And food category != 'Vegetable'
And geography state IN ('WA', 'OR')
Group by food category, geography state
Join Processing and Aggregation Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

SELECT food.category, geography.state, sum(sales.amt) 
FROM sales, food, geography 
WHERE sales.f_id = food.f_id 
AND sales.g_id = geography.g_id 
AND food.category != 'Vegetable' 
AND geography.state IN ('WA', 'OR') 
GROUP BY food.category, geography.state
In-Memory Aggregation Concepts: **DGKs**

- Dense Grouping Keys (DGKs)
  - A dense surrogate key \([0..N]\) representing a unique combination of grouping keys. DGKs can be used as an efficient substitute of the original grouping key.

<table>
<thead>
<tr>
<th>Dept</th>
<th>Category</th>
<th>Item</th>
<th>f_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Banana</td>
<td>0</td>
</tr>
<tr>
<td>Produce</td>
<td>Vegetable</td>
<td>Eggplant</td>
<td>2</td>
</tr>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Date</td>
<td>3</td>
</tr>
<tr>
<td>Produce</td>
<td>Grain</td>
<td>Farro</td>
<td>5</td>
</tr>
</tbody>
</table>
In-Memory Aggregation Concepts: DGKs

- Dense Grouping Keys (DGKs)
  - A dense surrogate key \([0..N]\) representing a unique combination of grouping keys. DGKs can be used as an efficient substitute of the original grouping key.

<table>
<thead>
<tr>
<th>Dept</th>
<th>Category</th>
<th>Item</th>
<th>f_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Banana</td>
<td>0</td>
</tr>
<tr>
<td>Produce</td>
<td>Vegetable</td>
<td>Eggplant</td>
<td>2</td>
</tr>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Date</td>
<td>3</td>
</tr>
<tr>
<td>Produce</td>
<td>Grain</td>
<td>Farro</td>
<td>5</td>
</tr>
</tbody>
</table>

```
SELECT category, rownum-1 AS DGK_f
FROM (SELECT category as category
      FROM food
      WHERE category != 'Vegetable'
      GROUP BY category)
```

```
category  | DGK_f
-----------|-------
Fruit      | 0
Grain      | 1
```

“Joinback” temp table
In-Memory Aggregation Concepts: **Key Vectors**

- **Key Vectors (KVs)**
  - An in-memory array mapping dimension join keys to DGK values.
    - Two purposes: 1) Precise filter and 2) Quickly index into aggregation accumulator
  - Elements in KV are fixed-width and bit-packed (width = max DGK value)
  - Non-numeric join keys use a hash table. Numeric ones are offset by minimum value.

<table>
<thead>
<tr>
<th>Dept</th>
<th>Category</th>
<th>Item</th>
<th>f_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Banana</td>
<td>0</td>
</tr>
<tr>
<td>Produce</td>
<td>Vegetable</td>
<td>Eggplant</td>
<td>2</td>
</tr>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Date</td>
<td>3</td>
</tr>
<tr>
<td>Produce</td>
<td>Grain</td>
<td>Farro</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Food KV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>category</th>
<th>DGK_f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruit</td>
<td>0</td>
</tr>
<tr>
<td>Grain</td>
<td>1</td>
</tr>
</tbody>
</table>
Join Processing and Aggregation Overview

Calculate total sales of non-veggies from WA and OR, group by food category and state

```
SELECT food.category, geography.state, sum(sales.amt)
FROM sales, food, geography
WHERE sales.f_id = food.f_id
AND sales.g_id = geography.g_id
AND food.category != 'Vegetable'
AND geography.state IN ('WA', 'OR')
GROUP BY food.category, geography.state
```
In-Memory Aggregation Design: Key Vector Create

Calculate total sales of non-veggies from WA and OR, grouped by food category and state.

Key Vector Create (*Geography*)

(State IN ('WA', 'OR'), GBY State)

<table>
<thead>
<tr>
<th>Country</th>
<th>State</th>
<th>City</th>
<th>g_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>WA</td>
<td>Seattle</td>
<td>0</td>
</tr>
<tr>
<td>USA</td>
<td>WA</td>
<td>Spokane</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>OR</td>
<td>Salem</td>
<td>2</td>
</tr>
<tr>
<td>USA</td>
<td>CA</td>
<td>SF</td>
<td>3</td>
</tr>
<tr>
<td>USA</td>
<td>CA</td>
<td>LA</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>1</td>
</tr>
</tbody>
</table>

Join Back Table

<table>
<thead>
<tr>
<th>State</th>
<th>DGK_g</th>
</tr>
</thead>
<tbody>
<tr>
<td>WA</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
</tr>
</tbody>
</table>

Geo KV

<table>
<thead>
<tr>
<th>DGK_g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geo</th>
<th>Food</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In-Memory Aggregation Design: Key Vector Create

Calculate total sales of non-veggies from WA and OR, group by food category and state

Key Vector Create (Food)

(Category != “Vegetable”, GBY Category)

Join Back Table

<table>
<thead>
<tr>
<th>Category</th>
<th>DGK_f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruit</td>
<td>0</td>
</tr>
<tr>
<td>Grain</td>
<td>1</td>
</tr>
</tbody>
</table>

Food Table

<table>
<thead>
<tr>
<th>Dept</th>
<th>Category</th>
<th>Item</th>
<th>f_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Banana</td>
<td>0</td>
</tr>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Apple</td>
<td>1</td>
</tr>
<tr>
<td>Produce</td>
<td>Vegetable</td>
<td>Eggplant</td>
<td>2</td>
</tr>
<tr>
<td>Produce</td>
<td>Fruit</td>
<td>Date</td>
<td>3</td>
</tr>
<tr>
<td>Produce</td>
<td>Vegetable</td>
<td>Celery</td>
<td>4</td>
</tr>
<tr>
<td>Produce</td>
<td>Grain</td>
<td>Farro</td>
<td>5</td>
</tr>
</tbody>
</table>

Food KV

<table>
<thead>
<tr>
<th>DGK_f</th>
<th>0</th>
<th>0</th>
<th>NULL</th>
<th>0</th>
<th>NULL</th>
<th>1</th>
</tr>
</thead>
</table>
In-Memory Aggregation Design: Key Vector Use

- Key Vectors, like Bloom Filters, are used to filter rows during Fact table scan
  - Difference is KVs are *precise* dimension filters - not probabilistic / inexact.
  - Bloom filter *requires* passing rows to be reprobed in HT to remove false positives.
  - *KVs are pushed down to storage layer* for optimized filtering on compressed formats.

<table>
<thead>
<tr>
<th>Sales Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_id</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
In-Memory Aggregation Design: Key Vector Use

- Key Vectors, like Bloom Filters, are used to filter rows during Fact table scan
  - Difference is KVs are *precise* dimension filters - not probabilistic / inexact.
  - Bloom filter requires passing rows to be reprobed in HT to remove false positives.
  - KVs are pushed down to storage layer for optimized filtering on compressed formats.

**Sales Table**

<table>
<thead>
<tr>
<th>g_id</th>
<th>f_id</th>
<th>Amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>120</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>130</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>140</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>150</td>
</tr>
</tbody>
</table>

**Food KV**

<table>
<thead>
<tr>
<th>DGK_f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
In-Memory Aggregation Design: Accumulator

- After KV Filtering completes, aggregation into accumulator can occur. And because KV Filtering is exact, aggregation can be pushed down to scan.
- Accumulated results are projected back (\(g_{id}, f_{id}, \text{accum}_{\text{val}}\))
- We can also send back partially accumulated results (per parallel thread)

<table>
<thead>
<tr>
<th>Sales Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(g_{id})</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{DGK}_{f})</td>
</tr>
<tr>
<td>(\text{DGK}_{g})</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

For Sales table rows{
  \(\text{colDGK} = \) geography.KV[Sales.g_id];
  \(\text{rowDGK} = \) food.KV[Sales.f_id];
  \(\text{accumulator}[\text{colDGK}, \text{rowDGK}] +=\) sales.Amt;
}
In-Memory Aggregation Design: Join Back

- Final step is to map DGKs back to the dimension attributes.
  - Implemented in the execution plan as an equi-join using the DGKs in the projected rows and the grouping keys found in Join-Back tables created during DGK creation.
- Inexpensive operation because aggregation / group-by already reduced the number of total rows

```java
for (DGK_g <= 1) {
    for (DGK_f <= 1) {
        if (Accumulator[DGK_g, DGK_f] != NULL) {
            projectRow(DGK_g, DGK_f, Accumulator[DGK_g, DGK_f]);
        }
    }
}
```
In-Memory Aggregation: Join Scalability

Increase the number of joins
IMA has significant advantage because it coalesces the joins and applies as single operation on Fact table.

Example query w/ 4 joins

```sql
SELECT c_region, p_mfgr, ..., SUM(lo_tax)
FROM lineorder, customer, part, supplier, date
WHERE lo_custkey = c_custkey
    AND lo_partkey = p_partkey
    AND lo_suppkey = s_suppkey
    AND lo_orderdate = d_datekey
GROUP BY c_region, p_mfgr, ...
ORDER BY 1, 2, 3, 4
```
Data Reorganization / Workload Management
Manual In-Memory Management

- If entire database fits within in-memory area, no need for DBA involvement!
- Otherwise, need to intelligently select in-memory candidates
- **Desired outcome**: Keep hot objects in-memory, remove colder objects
  - Access patterns are not known in advance and change over time
  - Hard for DBAs to achieve manually
Automatic In-Memory

In-Memory Column Store

Observe Access Patterns

• Eliminates trial and error regarding in-memory area contents
• Constant background action:
  • Classifies data as hot, intermediate or cold
  • Hotter in-memory tables automatically populated
  • Colder in-memory tables automatically removed
  • Intelligent algorithm takes into account space-benefit tradeoffs
• Useful for autonomous cloud services since no user intervention required
Introducing **Automatic In-Memory**

- Eliminates trial and error regarding in-memory area contents
- Constant background action:
  - Classifies data as hot, intermediate or cold
  - Hotter in-memory tables automatically populated
  - Colder in-memory tables automatically removed
  - Intelligent algorithm takes into account space-benefit tradeoffs
- Useful for autonomous cloud services since no user intervention required
Introducing **Automatic In-Memory**

- Eliminates trial and error regarding in-memory area contents
- Constant background action:
  - Classifies data as hot, intermediate or cold
  - Hotter in-memory tables automatically populated
  - Colder in-memory tables automatically removed
  - Intelligent algorithm takes into account space-benefit tradeoffs
- Useful for autonomous cloud services since no user intervention required
Exadata Database Machine

In-Memory Columnar scans

In-Flash Columnar scans
Exadata X8 and X8M Hardware

- Scale-Out 2 or 8 Socket Database Servers
  - Latest 24 core Intel Cascade Lake
- 100 Gb/sec RDMA over Converged Ethernet RoCE internal fabric or 40Gb/s InfiniBand
- Scale-Out intelligent 2-Socket Storage Servers
  - 60% more cores to offload database processing
    - Latest 16 core Intel Cascade Lake CPUs
  - 40% higher capacity 14 TB disk drives
Latest Flash Creates **Giant Bottleneck** for Shared Storage

- NVMe Flash: 5.5 GB/sec
- 40 Gb/sec SAN link: 5 GB/sec

*Single Flash Drive is Faster than fast SAN*

Exadata Scales as Flash is Added, Flash Arrays Bottlenecked by Network

- Exadata
- All-Flash Array

90% of flash performance is lost
Exadata Uniquely Achieves Memory Speed with Shared Flash

- Architecturally, storage arrays can share flash capacity but not flash performance due to network bottlenecks
  - Even with next gen scale-out, PCIe networks, NVMe over fabric
- Must move compute to data to achieve full flash potential
  - Requires owning full stack; can’t be solved in storage alone
- X8M delivers 560 GB/sec flash bandwidth to any server
  - Approaches 800 GB/sec aggregate DRAM bandwidth of DB servers
In-Memory Analytics Performance in Shared Storage

- Exadata flash throughput approaches DRAM throughput
  - Analytics SQL bottleneck moves from I/O to CPU
- Exadata storage automatically transforms table data into In-Memory DB columnar formats in Exadata Flash Cache
  - Enables fast CPU vector processing in storage server queries
- Uniquely optimizes next generation flash as memory
Capacity: Extend DBIM Formats in Flash Cache

100s of TBs Possible on Full Rack X7 (with 10X Faster Queries)

Database Server

- SGA
- IMC

Up to 1.5 TB DRAM

Extends In-Memory Column Store into Flash

25.6 TB Flash x 3 = 76.8 TB (or more)

IMC (In-Memory Columnar) data

In-Memory Columnar scans

In-Flash Columnar scans

Row formatted or HCC Data

Storage Server
Exadata with Database In-Memory: Fault Tolerance

- IMCUs for a table are populated into 2 database servers in a RAC cluster
  - INMEMORY DUPLICATE mode
  - Zero application impact on failure
- **Fully Duplicate** all IMCUs for a table across all RAC instances
  - INMEMORY DUPLICATE ALL mode
  - Additionally provides *performance* because always local access to a duplicate IMCU
- Application transparent
What Next?
Today (Baseline)

- Not all data can fit into Memory
- Queries go against column store in DRAM and row store on DISK
- DRAM Dimms up to 128GB, and very expensive.

New: Intel® Optane™ DC Persistent Memory

- Entire workload can fit into Memory
- With Memory Mode, hottest tables are cached in DRAM for fastest access
- Apache Dimms up to 512GB

New: Intel® Optane™ DC Persistent Memory + Oracle 20c

- Oracle 20c introduces new Deep Vectorization framework that extends vector processing to all SQL operators

JOIN

ITEMS + SALES
Intel® Optane™ DC Persistent Memory with Oracle In-Memory Database

<table>
<thead>
<tr>
<th>Memory Mode</th>
<th>Query Time (sec)</th>
<th>Rows</th>
<th>Speed Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM (384GB)</td>
<td>130</td>
<td>18,002,448,306</td>
<td>~10x Faster !!</td>
</tr>
<tr>
<td>Memory Mode (1.5TB)</td>
<td>12</td>
<td>18,002,448,306</td>
<td></td>
</tr>
<tr>
<td>Oracle 20c with Memory Mode (1.5TB)</td>
<td>3</td>
<td>18,002,448,306</td>
<td>~40x Faster !!</td>
</tr>
</tbody>
</table>

START

About Oracle
Copyright © 2014, 2019 Oracle and/or its affiliates All rights reserved.