In-Memory Aggregation

Accelerating Joins and Aggregations on the Oracle In-Memory Database

UKOUG 2018 (Liverpool, England)

Shasank Chavan
Vice President, In-Memory Technologies
Safe Harbor Statement

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, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
In-Memory Aggregation: Presentation Agenda

1. Oracle Database In-Memory Option Overview
2. Join Processing and Aggregation Overview
3. In-Memory Aggregation Concepts
4. In-Memory Aggregation Example
5. In-Memory Aggregation Optimizations
6. Debugging / Triaging IMA Execution from User’s Perspective
7. Demo
Oracle Database In-Memory: Architecture

- **Dual-Format Architecture**
  - Both row and column formats for table
    - Transactions benefit with existing row format
    - Analytics benefit with In-Memory columnar format
  - Simultaneously active and consistent

- **Blazing Fast Analytic Scans**
  - SIMD on Compressed Columnar Data Formats
  - Fast Bloom Filter → Faster Joins
    - But Join Processing still not fast enough
In-Memory Compression Units (IMCU)

- Unit of column store allocation
  - Columnar representation of a large number of rows (e.g. 1 million)
- Created by background populate
- Actual size depends on size of rows, compression factor, etc.
- Separate CU for each column
- Rowids also stored as a CU
Vector Processing: **Additional Advantage of Column Format**

- Each CPU core scans only columns in local memory
- SIMD vector instructions used to process multiple values in each instruction
  - E.g. Intel AVX-512 instructions with 512 bit vector registers
- **Billions of rows/sec** scan rate per CPU core
  - Row format is millions/sec

---

**Example:** Find all sales in state of CA

- Load multiple region values
- Vector Compare all values an 1 cycle
- > 100x Faster
Oracle Database In-Memory: Features / Integration

- Analytics
- HTAP Workloads
- Massive Capacity
- Multi-Modal
- Automation
- Active Data Guard
- Dynamic Workload Parallelism
- Hardware Acceleration
- Compression
- Nonvolatile Memory
- Big Data Integration
- Spatial / Graph
- In-Memory Aggregation

Oracle Cloud: In-Memory Powers the Cloud
In-Memory Aggregation: Presentation Agenda

1. Oracle Database In-Memory Option Overview
2. Join Processing and Aggregation Overview
3. In-Memory Aggregation Concepts
4. In-Memory Aggregation Example
5. In-Memory Aggregation Optimizations
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
```
Join Processing and Aggregation Overview

Calculate total *sales* of non-*eggies* 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
```
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
```

2 Build Bloom Filter
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
```
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
```
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 = 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 = 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
```
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

```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
```
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*

```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
```
In-Memory Aggregation: Presentation Agenda

1. Oracle Database In-Memory Option Overview
2. Join Processing and Aggregation Overview
3. In-Memory Aggregation Concepts
4. In-Memory Aggregation Example
5. In-Memory Aggregation Optimizations
In-Memory Aggregation: Concepts

• **Observations**
  
  – Excessive time spent processing fact tables in joins vs dimension tables
  
  – Hashing can be expensive
  
  – Carrying payload columns up through $N$ level of joins is expensive.
  
  – Columnar data format encoding are not leveraged outside of scans, resulting in decompression overhead
In-Memory Aggregation: Concepts

• Observations
  – Excessive time spent processing fact tables in joins vs dimension tables
  – Hashing can be expensive
  – Carrying payload columns up through $N$ level of joins is expensive.
  – Dictionary codes are not leveraged outside of scans, resulting in decompression overhead

• Goals
  – Optimize for typical data-warehouse schemas
    • Some number of small dimension tables
    • Small number of very large fact tables
    • Pre-process dimensions to save per row on fact table scan
  – Adaptive
  – Offload capability
In-Memory Aggregation: Concepts

• Observations
  – Excessive time spent processing fact tables in joins vs dimension tables
  – Hashing can be expensive
  – Carrying payload columns up through N level of joins is expensive.
  – Dictionary codes are not leveraged outside of scans, resulting in decompression overhead

• Goals
  – Optimize for typical data-warehouse schemas
    • Some number of small dimension tables
    • Small number of very large fact tables
    • Pre-process dimensions to save per row on fact table scan
  – Adaptive
  – Offload capability

• Requirements
  – |Dim| <= |Fact| / 10
  – |Out Rows| <= |Fact| / 1M
    • Otherwise post-processing step becomes comparatively expensive
  – Grouping columns come from dimension tables *
  – Measure columns come from the fact table *
  – Dim-Fact tables have 1-to-many relationship *

* Most optimized, but not required
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.

### 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>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>

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

<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>

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

- Key Vectors (KVs)
  - An in-memory array mapping dimension join keys to corresponding DGK values.
    - Two purposes: 1) Precise filter and 2) Quickly index into aggregation accumulator
  - Elements in KV are fixed-width and bit-packed, with width equal to 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>
In-Memory Aggregation: Presentation Agenda

1. Oracle Database In-Memory Option Overview
2. Join Processing and Aggregation Overview
3. In-Memory Aggregation Concepts
4. In-Memory Aggregation Example
5. In-Memory Aggregation Optimizations
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
```
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 (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>...</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 (Geography)

(State IN (‘WA’, ‘OR’), GBY State)

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>

Geography Table

<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>
</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 (Geography)
(State IN (‘WA’, ‘OR’), GBY State)

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 Table

<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>...</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>
<tr>
<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)

<table>
<thead>
<tr>
<th>Food Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept</td>
</tr>
<tr>
<td>Produce</td>
</tr>
<tr>
<td>Produce</td>
</tr>
<tr>
<td>Produce</td>
</tr>
<tr>
<td>Produce</td>
</tr>
<tr>
<td>Produce</td>
</tr>
<tr>
<td>Produce</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)

<table>
<thead>
<tr>
<th>Join Back Table</th>
<th>Food Table</th>
<th>Food KV</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category</strong></td>
<td><strong>Dept</strong></td>
<td><strong>DGK f</strong></td>
</tr>
<tr>
<td>Fruit</td>
<td>Produce</td>
<td>0</td>
</tr>
<tr>
<td>Grain</td>
<td>Produce</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Produce</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>Produce</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Produce</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Produce</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Produce</td>
<td>5</td>
</tr>
</tbody>
</table>

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

![Diagram of a tree with nodes labeled Geo, Food, and Sales.](Image)

<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>
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.

### Geo KV

<table>
<thead>
<tr>
<th>DGK_g</th>
<th>f_id</th>
<th>Amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</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>NULL</td>
<td>2</td>
<td>140</td>
</tr>
<tr>
<td>NULL</td>
<td>1</td>
<td>150</td>
</tr>
</tbody>
</table>

### 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>
<th>g_id</th>
<th>f_id</th>
<th>Amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</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, accum_val*)
- We can also send back partially accumulated results (per parallel thread)

<table>
<thead>
<tr>
<th>Sales Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>g_id</strong></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><strong>DGK_f</strong></td>
</tr>
<tr>
<td><strong>DGK_g</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

For Sales table rows{
    colDGK = geography.KV[Sales.g_id];
    rowDGK = food.KV[Sales.f_id];
    accumulator[colDGK, 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 Design: Execution Plan

• Once query compiler determines the query is eligible for IMA:
  – New KV-Create operators inserted into plan to derive DGKs and create the temporary Join-Back tables (Late Materialization).
  – New KV-Use operators are then inserted above the fact table scan operator, to provide precise filtering of non-matching rows, using KVs.
  – Vector GBY operator will aggregate into accumulators using DGK maps.
### In-Memory Aggregation Design: Execution Plan

<table>
<thead>
<tr>
<th></th>
<th>SELECT STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TEMP TABLE TRANSFORMATION</td>
</tr>
<tr>
<td>2</td>
<td>LOAD AS SELECT</td>
</tr>
<tr>
<td>3</td>
<td>VECTOR GROUP BY</td>
</tr>
<tr>
<td>4</td>
<td>KEY VECTOR CREATE BUFFERED</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS INMEMORY FULL</td>
</tr>
<tr>
<td>6</td>
<td>LOAD AS SELECT</td>
</tr>
<tr>
<td>7</td>
<td>VECTOR GROUP BY</td>
</tr>
<tr>
<td>8</td>
<td>KEY VECTOR CREATE BUFFERED</td>
</tr>
<tr>
<td>9</td>
<td>TABLE ACCESS INMEMORY FULL</td>
</tr>
<tr>
<td>10</td>
<td>HASH GROUP BY</td>
</tr>
<tr>
<td>11</td>
<td>HASH JOIN</td>
</tr>
<tr>
<td>12</td>
<td>HASH JOIN</td>
</tr>
<tr>
<td>13</td>
<td>TABLE ACCESS FULL</td>
</tr>
<tr>
<td>14</td>
<td>VIEW</td>
</tr>
<tr>
<td>15</td>
<td>VECTOR GROUP BY</td>
</tr>
<tr>
<td>16</td>
<td>HASH GROUP BY</td>
</tr>
<tr>
<td>17</td>
<td>KEY VECTOR USE</td>
</tr>
<tr>
<td>18</td>
<td>KEY VECTOR USE</td>
</tr>
<tr>
<td>19</td>
<td>TABLE ACCESS INMEMORY FULL</td>
</tr>
<tr>
<td>20</td>
<td>TABLE ACCESS FULL</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
</tr>
<tr>
<td>1</td>
<td>TEMP TABLE TRANSFORMATION</td>
</tr>
<tr>
<td>2</td>
<td>LOAD AS SELECT</td>
</tr>
<tr>
<td>3</td>
<td>VECTOR GROUP BY</td>
</tr>
<tr>
<td>4</td>
<td>KEY VECTOR CREATE BUFFERED</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS INMEMORY FULL</td>
</tr>
<tr>
<td>6</td>
<td>LOAD AS SELECT</td>
</tr>
<tr>
<td>7</td>
<td>VECTOR GROUP BY</td>
</tr>
<tr>
<td>8</td>
<td>KEY VECTOR CREATE BUFFERED</td>
</tr>
<tr>
<td>9</td>
<td>TABLE ACCESS INMEMORY FULL</td>
</tr>
<tr>
<td>10</td>
<td>HASH GROUP BY</td>
</tr>
<tr>
<td>11</td>
<td>HASH JOIN</td>
</tr>
<tr>
<td>12</td>
<td>HASH JOIN</td>
</tr>
<tr>
<td>13</td>
<td>TABLE ACCESS FULL</td>
</tr>
<tr>
<td>14</td>
<td>VIEW</td>
</tr>
<tr>
<td>15</td>
<td>VECTOR GROUP BY</td>
</tr>
<tr>
<td>16</td>
<td>HASH GROUP BY</td>
</tr>
<tr>
<td>17</td>
<td>KEY VECTOR USE</td>
</tr>
<tr>
<td>18</td>
<td>KEY VECTOR USE</td>
</tr>
<tr>
<td>19</td>
<td>TABLE ACCESS INMEMORY FULL</td>
</tr>
<tr>
<td>20</td>
<td>TABLE ACCESS FULL</td>
</tr>
</tbody>
</table>

- `SYS_TEMP_0FD9DADAD_9873DD`<br>`:KV0000`<br>`FOOD`<br>`SYS_TEMP_0FD9DADA_E_9873DD`<br>`:KV0001`<br>`GEOGRAPHY`<br>`SYS_TEMP_0FD9DADAE_9873DD`<br>`VW_VT_AF278325`<br>`SYS_TEMP_0FD9DADA_E_9873DD`<br>`:KV0001`<br>`SALES`<br>`SYS_TEMP_0FD9DADAD_9873DD`
In-Memory Aggregation: Presentation Agenda

1. Oracle Database In-Memory Option Overview
2. Join Processing and Aggregation Overview
3. In-Memory Aggregation Concepts
4. In-Memory Aggregation Example
5. In-Memory Aggregation Optimizations
In-Memory Aggregation: Optimizations

- **Filtering Optimizations**
  - Large-scale pruning based on meta-data (e.g. upper/lower KV bounds)
  - Key Vector filtering pushed down to the table scan layer
  - SIMD techniques for Key Vector filtering.
  - Evaluate Key Vector Filtering once per dictionary symbol (dictionary encoding)
    - SIMD techniques to translate results from dictionary back to rows.

- **Aggregation / Group-By Optimizations**
  - SIMD GBY-SUM aggregation (highly tailored to CPU architecture).
Join keys from fact table loaded into a SIMD register (assuming join key is 4 byte value)

Example shows looking up join keys in a nibble packed segmented key vector
Join keys from fact table loaded into a SIMD register (assuming join key is 4 byte value)

Example shows looking up join keys in a nibble packed segmented key vector
SIMD KV Filtration

Join keys from fact table loaded into a SIMD register (assuming join key is 4 byte value)

Example shows looking up join keys in a nibble packed segmented key vector
SIMD KV Filtration

Join keys from fact table loaded into a SIMD register \(\text{(assuming join key is 4 byte value)}\)

Example shows looking up join keys in a nibble packed segmented key vector

Nibble Index in segment

Segment Index
SIMD KV Filtration

Join keys from fact table loaded into a SIMD register (assuming join key is 4 byte value)

Example shows looking up join keys in a nibble packed segmented key vector

Nibble Index in segment

Segment Index

Byte Index in segment
SIMD KV Filtration

Join keys from fact table loaded into a SIMD register (assuming join key is 4 byte value)

Example shows looking up join keys in a nibble packed segmented key vector

Nibble Index in segment

LSB of nibble index

Byte Index in segment

KV_seg[ + ] [ ]

Intel AVX-512 VPSRLVW VPSRLVD VPSRLVQ — Variable Bit Shift Right Logical + Mask

Dense key from KV
In-Memory Aggregation: Presentation Agenda

1. Oracle Database In-Memory Option Overview
2. Join Processing and Aggregation Overview
3. In-Memory Aggregation Concepts
4. In-Memory Aggregation Example
5. In-Memory Aggregation Optimizations
6. Performance Evaluation
In-Memory Aggregation: Performance Evaluations

• Demonstrate the benefits of IMA using SSB schema (SF 100)
  – Modified SSB schema (i.e. predicates are made less selective => more aggregation)

• Speedup is over previous techniques implemented on same IM store
  – i.e. Compared to traditional Hash-Join / Hash GBY implementation optimized for IM.

• 2-socket Intel XEON E5-2697 V3 @ 2.6Ghz (Haswell) w/ 36MB L3 cache and 14 cores per socket. Enabled HT – total 56 VCPUs.

• Tests Run
  – Aggregation Scalability – speedup as we increase number of measures.
  – Join Scalability – speedup seen as we increase the number of joins.
  – Grouping Key Scalability – speedup seen as we increase the number of grouping keys.
IMA: Performance: Aggregation Scalability

- Increase the number of measures
- Joins with low cardinality grouping key
- Example query w/ 2 joins and 6 measures:

```
SELECT p_mfgr, c_region, SUM(lo_tax),
     SUM(lo_discount), SUM(lo_quantity),
     SUM(lo_supplycost), SUM(lo_revenue),
     SUM(lo_ordtotalprice)
FROM lineorder, part, customer
WHERE lo_partkey = p_partkey
  AND lo_custkey = c_custkey
GROUP BY p_mfgr, c_region
ORDER BY 1, 2
```
IMA: Performance: 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, s_region,
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, s_region
ORDER BY 1, 2, 3, 4
```
IMA: Performance: **Grouping Key Scalability**

- Increase the number of grouping keys with low cardinality columns
- Benefit from using single DGK for all grouping keys from each dimension
  - HJ has to maintain additional keys/vals in hash table
- Expense from Key Create only.

**Example query:**

```
SELECT p.pl1, c.pl1
    p.pl2, c.pl2
    SUM(lo_tax)
FROM lineorder,
    widecust c,
    widepart p
WHERE
    lo_custkey = c_custkey
    AND
    lo_partkey = p_partkey
GROUP BY p.pl1, c.pl1,
    p.pl2, c.pl2
```
In-Memory Aggregation: Presentation Agenda

1. Oracle Database In-Memory Option Overview
2. Join Processing and Aggregation Overview
3. In-Memory Aggregation Concepts
4. In-Memory Aggregation Example
5. In-Memory Aggregation Optimizations
6. Performance Evaluation
7. Debugging / Triaging IMA Execution from User’s Perspective
IMA Performance Analysis: Tools

- Statistics – an AWR will show many statistics related to key vectors and IM
- Stats to the right taken from a Star Schema Benchmark Scale Factor=64 (384M fact rows)
  - 13 queries (KV queries)
  - 36 total joins (KVs created... also stats about them)
  - Scanned 2.9B rows
  - Evaluated 1.4B dictionary codes (across all 36 joins)
  - Aggregated 34M rows as part of the scan: 1% of all rows scanned
  - Scan actually produced only 3M rows: 0.1% of all rows

### KV Statistics

- **key vector CU codes processed**: 1,380,757,451
- **key vector CUs filtered**: 17,807
- **key vector cas merge locking retrial**: 76
- **key vector cas merge operations**: 9
- **key vector dblk batch parcels**: 41
- **key vector dblk range parcels**: 562
- **key vector hash cells scanned**: 1,652,058
- **key vector hash inserts**: 482
- **key vector hash probes**: 1,651,576
- **key vector non cas merge operations**: 14
- **key vector queries**: 13
- **key vector rows processed by code**: 456,724
- **key vectors created**: 36
- **key vectors created (bit wide)**: 12
- **key vectors created (byte wide)**: 2
- **key vectors created (indirect layout)**: 2
- **key vectors created (nibble wide)**: 22
- **key vectors created (offset layout)**: 13
- **key vectors created (simple layout)**: 21

### IM Statistics

- **IM scan bytes in-memory**: 156,817,446,697
- **IM scan bytes uncompressed**: 296,535,784,799
- **IM scan delta - only base scan**: 5,733
- **IM scan dict engine results reused**: 5,656
- **IM scan rows**: 2,895,285,128
- **IM scan rows optimized**: 0
- **IM scan rows pcode aggregated**: 34,007,113
- **IM scan rows projected**: 3,237,345
- **IM scan rows valid**: 2,895,285,128
IMA Performance Analysis: **Tools**

**SQL Monitor**

- The most comprehensive tool for viewing parallel queries’ timing
- Timeline a key first component for understanding IMA, since dims created first
- Example to right shows join of 100M row customer dim to 384M row fact table
  - Breaks 10X rule: IMA forced
  - Long time in KV creation
IMA Performance Analysis: Tools

• SQL Monitor binoculars have further details for IMA operations

• Key Vector Create has statistics about each key vector’s shape and size

• Key Vector Use has statistics about how many times it was checked, and how much filtering was done

• Vector Group by shows memory usage per process and total across PQ slaves
In-Memory Aggregation: Autonomy

• IMA does not require user input of any kind to improve performance
  – Selected by the optimizer for appropriate queries
  – In version 12.1, costing was very cautious
    • Strongly oriented towards not introducing performance regressions
  – In 12.2 or later, costing is more accurate

• As with all cost based optimizer features, it relies on good statistics!
In-Memory Aggregation: Adaptivity

• In case IMA engaged on the basis of poor statistics, there are a number of fallbacks we have to ensure the query still completes, and minimize performance issues
  – If a KV is too large to fit in memory, it is paged (Oracle18+), or bypassed (Oracle12)
  – If Vector Group by cannot get enough memory, we bypass it and use hash group by instead

• Optimizer Adaptive Features should also correct the plan and not choose a IMA again if the query is rerun
In-Memory Aggregation: Diagnosibility

• Typical reasons why IMA is not engaging:
  – In Memory option is disabled (inmemory_size=0)
  – CBO is disabled
  – Query constructs not supported by CBQT
  – No (simple) group by
  – No join condition specified
  – No aggregation functions
  – Fact and dimension size too similar
  – Conservative costing (V12.1)
In-Memory Aggregation: Hints

- In most situations, these should not be necessary
  - But can be useful to diagnose issues or experiment
- Hints that can be used to control IMA engagement

<table>
<thead>
<tr>
<th>Hint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECTOR_TRANSFORM</td>
<td>Force this query block to transform</td>
</tr>
<tr>
<td>NO_VECTOR_TRANSFORM</td>
<td>Prevent this query block from transforming</td>
</tr>
<tr>
<td>VECTOR_TRANSFORM_DIMS(t)</td>
<td>Treat table $t$ as a dimension</td>
</tr>
<tr>
<td>NO_VECTOR_TRANSFORM_DIMS(t)</td>
<td>Don’t treat table $t$ as a dimension</td>
</tr>
<tr>
<td>VECTOR_TRANSFORM_FACT(t)</td>
<td>Treat table $t$ as a fact</td>
</tr>
<tr>
<td>NO_VECTOR_TRANSFORM_FACT(t)</td>
<td>Don’t treat table $t$ as a fact</td>
</tr>
</tbody>
</table>
Thank You
In-Memory Aggregation Concepts: Late Materialization

• Late Materialization
  – Well-known optimization technique which replaces column values with codes
  – The codes are used throughout query execution and then replaced with column values as late as possible
  – DGKs are used as proxy for multiple grouping columns
  – Allows aggregations to happen at lowest level of execution plan, but delay projection of the actual grouping keys and dimension attributes until final aggregation.
    • At that time, replace DGK with the aggregated dimension information in join back table.