Toward Implementing Incremental View Maintenance on PostgreSQL

Yugo Nagata @ SRA OSS, Inc. Japan.

PGCon 2019
- May 31, 2019
Who am I

• Yugo Nagata
  - Engineer at SRA OSS, Inc. Japan
• PostgreSQL experiences
  - Technical support
  - consulting
  - Education
  - R&D
Outline

• Introduction
  – Views and materialize views
  – Incremental View Maintenance (IVM)
• Implementing IVM on PostgreSQL
  – What to be considered to implement IVM
  – Work-in-Progress patch
  – How it works
• Examples
  – Performance Evaluation
• Discussions
What is Incremental View Maintenance (IVM)
CREATE VIEW V AS
   SELECT device_name, pid, price
   FROM devices d
   JOIN parts p
       ON d.pid = p.pid;

- A view is a virtual relation defined by a query on base tables.
  - Only the definition query is stored.
- The result is computed when a query is issued to a view.
Materialized Views

CREATE MATERIALIZED VIEW V AS
SELECT device_name, pid, price
FROM devices d
JOIN parts p
ON d.pid = p.pid;

- Materialized views persist the results in a table-like form.
- No need to compute the result when a query is issued.
  - Enables faster access to data.
- The data is not always up to date.
  - Need maintenance.
Creating Materialized Views

```sql
CREATE MATERIALIZED VIEW V AS
    SELECT device_name, pid, price
    FROM devices d
    JOIN parts p
    ON d.pid = p.pid;
```

- The data of a materialized view is computed at definition time.
  - This is similar to “CREATE TABLE AS” statement.
  - The result of the definition query is inserted into the materialized view.
- Need maintenance to keep consistency between the materialized data and base tables.
Refreshing Materialized Views

- Need to re-compute the result of the definition query.
- Replacing the contents of a materialized view with the result.

```
REFRESH MATERIALIZED VIEW V;
```
Refreshing Materialized Views

REFRESH MATERIALIZED VIEW CONCURRENTLY V;

- With CONCURRENTLY option, the materialized view is refreshed without locking out concurrent selects on the view.
- Need to re-compute the result of the definition query, too.
Incremental View Maintenance

- Incremental View Maintenance (IVM)
  - Compute and apply only the incremental changes to the materialized views

\[
V = Q_v(D)
\]

\[
\delta^u(V)
\]

\[
V_{new} = Q_v(D')
\]

- Base relations
- Updated base relations
- Materialized view
- Updated materialized View
- View definition
- Incremental maintenance
- Changes of materialized view
- Changes of base tables
- Incremental maintenance
- Update query
- Refreshing
- Recomputation
- Update base relations

Copyright © 2019 SRA OSS, Inc. Japan All rights reserved.
Basic Theory of IVM

- **View definition**
  - Ex.) Natural join view
    \[
    V \overset{\text{def}}{=} R \bowtie S
    \]

- **Change on a base table**
  - \( R \leftarrow (R - \nabla R \cup \Delta R) \)

- **Calculation of change on view**
  - \( \nabla V = \nabla R \bowtie S \)
  - \( \Delta V = \Delta R \bowtie S \)

- **Apply the change to the view**
  - \( V \leftarrow (V - \nabla V \cup \Delta V) \)

SELECT * FROM R NATURAL JOIN S;

- **Symbols:**
  - \( R, S \) base tables
  - \( \nabla R \) deleted tuples
  - \( \Delta R \) inserted tuples
Basic Theory of IVM: Example (1)

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
<th>roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>two</td>
<td>II</td>
</tr>
<tr>
<td>3</td>
<td>three</td>
<td>III</td>
</tr>
</tbody>
</table>

\[ V \overset{\text{def}}{=} R \natural S \]

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
<th>roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>two</td>
<td>II</td>
</tr>
<tr>
<td>3</td>
<td>three</td>
<td>III</td>
</tr>
</tbody>
</table>
Basic Theory of IVM: Example (2)

Table R is changed

\[
R \leftarrow (R - \nabla R \cup \Delta R)
\]

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one → ONE</td>
</tr>
<tr>
<td>2</td>
<td>two</td>
</tr>
<tr>
<td>3</td>
<td>three</td>
</tr>
</tbody>
</table>

\[
\nabla R
\]

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
</tr>
</tbody>
</table>

\[
\Delta R
\]

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ONE</td>
</tr>
</tbody>
</table>

\[
\nabla V = \nabla R \bowtie S
\]

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
<th>roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
<td>I</td>
</tr>
</tbody>
</table>

\[
\Delta V = \Delta R \bowtie S
\]

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
<th>roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ONE</td>
<td>I</td>
</tr>
</tbody>
</table>
Basic Theory of IVM: Example (3)

\[ \nabla V \]

\[ \Delta V \]

\[ V \leftarrow (V - \nabla V \cup \Delta V) \]

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
<th>roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
<td>I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
<th>english</th>
<th>roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ONE</td>
<td>I</td>
</tr>
</tbody>
</table>

Delete

Insert

View V is updated by applying the calculated changes.
Implementing IVM on PostgreSQL
Considerations implementing IVM (1)

- How to extract changes on base tables
  - AFTER trigger and Transition Tables
  - Logical decoding of WAL is another idea.

- How to compute the delta to be applied to materialized views
  - Basically, based on relational algebra (or bag algebra).
  - Starting from a simple view definition:
    - Selection-Projection-Join views
Considerations implementing IVM (2)

- When to maintain materialized views
  - Immediate maintenance:
    - The materialized view is updated in the same transaction where the base table is updated.
  - Deferred maintenance:
    - The materialized view is updated after the transaction is committed
      - When view is accessed
      - As a response to user command (like REFRESH)
      - periodically
      - etc.

- How to handle views with tuple duplicates or DISTINCT clause
Views with Tuple Duplicates

```
SELECT english, roman
FROM R JOIN S USING (id);
```

<table>
<thead>
<tr>
<th>english</th>
<th>roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>I</td>
</tr>
<tr>
<td>two</td>
<td>II</td>
</tr>
<tr>
<td>two</td>
<td>II</td>
</tr>
<tr>
<td>three</td>
<td>III</td>
</tr>
</tbody>
</table>

- Only one tuple of duplicated two must be deleted.
- `DELETE` statement cannot be used because this delete two tuples.
Views with DISTINCT clause

```
SELECT DISTINCT english, greek
FROM R JOIN S USING (id);
```

<table>
<thead>
<tr>
<th>english</th>
<th>roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>I</td>
</tr>
<tr>
<td>two</td>
<td>II</td>
</tr>
<tr>
<td>three</td>
<td>III</td>
</tr>
</tbody>
</table>

- A tuple is deleted if and only if duplicity of the tuple becomes zero.
- Additional tuple can not be inserted if there is already the same one.
IVM Implementation using OIDs (PGConf.EU 2018)

• PoC (Proof of Concept) implementation
  - Using row OIDs as “primary keys” of tuples in a materialized view
  - This can handle views with tuple duplicates correctly.
    • DISTINCT is not supported.
  - Materialized views can be incrementally updated using REFRESH command. (a kind of Deferred Maintenance)

• Problems:
  - OID system column is removed since PostgreSQL 12.
  - Needs many changes in executor nodes.
New IVM Implementation

- Working-in-Progress patch has been submitted

- Provides a kind of Immediate Maintenance
  - Materialized views can be updated automatically and incrementally after base tables are updated.

- Supports views including duplicate tuples or DISTINCT clause in the view definition
  - "counting algorithm" is used
Counting algorithm (1)

• Algorithm for handling tuple duplicate or DISTINCT in IVM
  - The numbers of tuples are counted and this information is stored in materialized views.

<table>
<thead>
<tr>
<th>english</th>
<th>roman</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>two</td>
<td>II</td>
<td>2</td>
</tr>
<tr>
<td>three</td>
<td>III</td>
<td>1</td>
</tr>
</tbody>
</table>
Counting algorithm (2)

- Algorithm for handling tuple duplicate or DISTINCT in IVM
  - The numbers of tuples are counted and this information is stored in materialized views.
    - When tuples are to be inserted into the view, the count increases.
    - When tuples are to be deleted from the view, the count decreases.
    - If the count becomes zero, this tuple is deleted.

<table>
<thead>
<tr>
<th>V</th>
<th>delete</th>
<th>∇V</th>
<th>insert</th>
<th>∆V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>english</td>
<td>roman</td>
<td>count</td>
<td>english</td>
<td>roman</td>
</tr>
<tr>
<td>one</td>
<td>I</td>
<td>1</td>
<td>two</td>
<td>II</td>
</tr>
<tr>
<td>two</td>
<td>II</td>
<td>2 → 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>three</td>
<td>III</td>
<td>1 → 2</td>
<td>three</td>
<td>III</td>
</tr>
</tbody>
</table>
How it works
Creating materialized views (1)

- CREATE INCREMENTAL MATERIALIZED VIEW
  - Creates materialized views which is updated automatically and incrementally after base tables are changed
  - This syntax is just tentative, so it may be changed.

```sql
CREATE INCREMENTAL MATERIALIZED VIEW MV AS
    SELECT device_name, pid, price
    FROM devices d
    JOIN parts p
    ON d.pid = p.pid;
```
Creating materialized views (2)

- When populating the materialized view:
  - The number of tuples are counted by adding `count(*)` and `GROUP BY` to the query.
  - The result of count is stored in the matview as a special column named "`__ivm_count__`".

```sql
CREATE INCREMENTAL MATERIALIZED VIEW MV AS
    SELECT count(*) AS __ivm_count__,
           device_name, pid, price
    FROM devices d
    JOIN parts p
    ON d.pid = p.pid
    GROUP BY device_name, pid, price;
```
Creating materialized views (3)

- AFTER triggers are created on the all base tables.
  - For INSERT, DELETE, and UPDATE
  - Statement level trigger
  - With Transition Tables

- Triggers are Created automatically and internally rather than issuing CREATE TRIGGER statement.
  - Similar to the implementation of foreign key constrains

Example of an equivalent query:

```sql
CREATE TRIGGER IVM_trigger_upd_16598
  AFTER UPDATE ON devises
  REFERENCING NEW TABLE AS ivm_newtable OLD TABLE AS ivm_oldtable
  FOR EACH STATEMENT
  EXECUTE FUNCTION IVM_immediate_maintenance('public.mv');
```
Transition Tables

CREATE TRIGGER IVM_trigger_upd_16598
AFTER UPDATE ON devises
REFERENCING NEW TABLE AS ivm_newtable OLD TABLE AS ivm_oldtable
FOR EACH STATEMENT
EXECUTE FUNCTION IVM_immediate_maintenance('public.mv');

- This is a feature of AFTER trigger since PostgreSQL 10.
- Changes on tables can be referred in the trigger function using table names specified by REFERENCING clause.
  - `ivm_oldtable` contains tuples deleted from the table in a statement.
  - `ivm_newtable` contains tuples newly inserted into the table.
  - In theory, corresponding $\nabla R$ and $\Delta R$ respectively.
Calculating Changes on Views

• Calculate the changes on the materialized view by:
  – Replacing the base table in the view definition query with the transition table.
  – Using count(*) and GROUP BY to count the duplicity of tuples.

• The results are stored into temporary tables. (as $\nabla V$ and $\Delta V$)

```sql
CREATE TEMPORARY TABLE tempname_old AS
  SELECT count(*) AS __ivm_count__, device_name, pid, price
  FROM ivm_oldtable d
  JOIN parts p
    ON d.pid = p.pid
  GROUP BY device_name, pid, price;

CREATE TEMPORARY TABLE tempname_new AS
  SELECT count(*) AS __ivm_count__, device_name, pid, price
  FROM ivm_newtable d
  JOIN parts p
    ON d.pid = p.pid
  GROUP BY device_name, pid, price;
```
Applying Changes to View (1)

• The materialized view is updated by merging the calculated changes.
  - For each tuple in the change:
    • If the corresponding tuple already exists, the value of `__ivm_count__` column in the view is updated
    • Rather than executing DELETE or INSERT simply
  - When the values becomes zero, the corresponding tuple is deleted.
  - Using modifying CTE (WITH clause)
Applying Changes to View (2)

- Decrease `__ivm_count__`, or delete an old tuple

```sql
WITH t AS (  
  SELECT diff.__ivm_count__,  
      (diff.__ivm_count__ = mv.__ivm_count__) AS for_dlt,  
      mv.ctid  
  FROM matview_name AS mv, tempname_old AS diff  
  WHERE (mv.device_name, mv.pid, mv.price)  
      = (diff.device_name, diff.pid, diff.price)  
  ),
updt AS (  
  UPDATE mateview_name AS mv  
  SET __ivm_count__ = mv.__ivm_count__ - t.__ivm_count__  
  FROM t  
  WHERE mv.ctid = t.ctid AND NOT for_dlt
)

DELETE FROM matview_name AS mv  
USING t  
WHERE mv.ctid = t.ctid AND for_dlt;
```
Applying Changes to View (3)

- Increase `__ivm_count__`, or Insert a new tuple

```sql
WITH updt AS ( 
    UPDATE matview_name AS mv 
    SET `__ivm_count__` = mv.`__ivm_count__` + diff.`__ivm_count__` 
    FROM temptable_new AS diff 
    WHERE (mv.device_name, mv.pid, mv.price) 
      = (diff.device_name, diff.pid, diff.price) 
    RETURNING diff.device_name, diff.pid, diff.price
)

INSERT INTO matview_name 
(SELECT * FROM temptable_new AS diff 
 WHERE (diff.device_name, diff.pid, diff.price) 
 NOT IN (SELECT * FROM updt));
```
Access to materialized views

- When SELECT is issued for materialized views with IVM:
  - case 1: Defined with DISTINCT:
    - All columns (except to __ivm_count__) of each tuple are returned.
    - Duplicity of tuples are already eliminated by GROUP BY.
  - case 2: DISTINCT is not used:
    - Returns each tuple __ivm_count__ times.
    - By rewriting the SELECT query to replace the view with a sub-query which joins the view and generate_series function.

```sql
SELECT mv.* FROM mv, generate_series(1, mv.__ivm_count__);
```
Examples
Example 1

postgres=# CREATE INCREMENTAL MATERIALIZED VIEW m AS SELECT * FROM t0;
SELECT 3
postgres=# SELECT * FROM m;
---
  3
  2
  1
(3 rows)

postgres=# INSERT INTO t0 VALUES (4);
INSERT 0 1
postgres=# SELECT * FROM m;
---
  3
  2
  1
  4
(4 rows)

Creating a materialized view with IVM option

Insert a tuple into the base table.

The view is automatically updated.
Example 2-1

```sql
postgres=# SELECT * FROM t1;
 id | t
----+---
  1 | A
  2 | B
  3 | C
  4 | A
(4 rows)

postgres=# CREATE INCREMENTAL MATERIALIZED VIEW m1 AS SELECT t FROM t1;
SELECT 3

postgres=# SELECT * FROM m1;
 t
---
 A
 A
 C
 B
(4 rows)
```

Creating a materialized view with tuple duplicates
Example 2-2

postgres=# INSERT INTO t1 VALUES (5, 'B');
INSERT 0 1
postgres=# DELETE FROM t1 WHERE id IN (1,3);
DELETE 2
postgres=# SELECT * FROM m1;
t---
B
B
A
(3 rows)

The view with tuple duplicates is correctly updated.

Before:

<table>
<thead>
<tr>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>
(3 rows)

Inserting (5,B) into and deleting (1, A), (3, C) from the base table.
Example 3

```sql
postgres=# SELECT *, __ivm_count__ FROM m1;
| t | __ivm_count__ |
|---+---------------|
| B |             2 |
| B |             2 |
| A |             1 |
(3 rows)
```

__ivm_count__ column is invisible for users when "SELECT * FROM ..." is issued, but users can see this by specifying it explicitly.

```sql
postgres=# EXPLAIN SELECT * FROM m1;

QUERY PLAN

------------------------------------------------------------------------------
| Nested Loop (cost=0.00..61.03 rows=3000 width=2) |
  | -> Seq Scan on m1 mv (cost=0.00..1.03 rows=3 width=10) |
  |   -> Function Scan on generate_series (cost=0.00..10.00 rows=1000 width=0) |
(3 rows)
```

The internal usage of generate_series function is visible in the EXPLAIN result.
Simple Performance Evaluation (1)

• Materialized views of a simple join using pgbench tables:

```
CREATE MATERIALIZED VIEW mv_normal AS
    SELECT aid, bid, abalance, bbalance
    FROM pgbench_accounts JOIN pgbench_branches
    USING (bid)
    WHERE abalance > 0 OR bbalance > 0;
```

```
CREATE INCREMENTAL MATERIALIZED VIEW mv_ivm AS
    SELECT aid, bid, abalance, bbalance
    FROM pgbench_accounts JOIN pgbench_branches
    USING (bid)
    WHERE abalance > 0 OR bbalance > 0;
```
Simple Performance Evaluation (2)

```
test=# REFRESH MATERIALIZED VIEW mv_normal;
REFRESH MATERIALIZED VIEW
Time: 11210.563 ms (00:11.211)

CREATE INDEX on mv_ivm (aid,bid);

SELECT * FROM mv_ivm WHERE aid = 1;

<table>
<thead>
<tr>
<th>aid</th>
<th>bid</th>
<th>abalance</th>
<th>bbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1000</td>
<td>10</td>
</tr>
</tbody>
</table>

(1 row)

UPDATE pgbench_accounts SET abalance = 1000 WHERE aid = 1;

Time: 18.634 ms

SELECT * FROM mv_ivm WHERE aid = 1;

<table>
<thead>
<tr>
<th>aid</th>
<th>bid</th>
<th>abalance</th>
<th>bbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1000</td>
<td>10</td>
</tr>
</tbody>
</table>

(1 row)
```

The standard REFRESH of mv_normal took more than 10 seconds.

Creating an index on mv_ivm

Updating a tuple in pgbench_accounts took 18ms.

mv_ivm was updated automatically and correctly.
Simple Performance Evaluation (3)

```
test=# DROP INDEX mv_ivm__aid_bid_idx;
DROP INDEX
Time: 10.613 ms

test=# UPDATE pgbench_accounts SET abalance = 2000 WHERE aid = 1;
UPDATE 1
Time: 3931.274 ms (00:03.931)
```

However, if there are not indexes on mv_ivm, it took about 4 sec.

Although this is faster than normal REFRESH, appropriate indexes are needed on materialized views for efficient IVM.
Current Restrictions

• Supported:
  - selection, projection, inner join, DISTINCT

• Not supported:
  - Aggregation and GROUP BY
  - Self-join, sub-queries, OUTER JOIN, CTE, window functions
  - Set operations (UNION, EXCEPT, INTERSECT)

• I plan to deal with some aggregations by the first release.
Timing of View Maintenance

• Currently, only Immediate Maintenance is supported:
  – Materialized views are updated immediately when a base table is modified.

• Deferred Maintenance:
  – Materialized views are updated after the transaction, for example, by the user command like REFRESH.
  – Need to implement a mechanism to maintain “logs” for recording changes of base tables and another algorithm to update materialized views.

• There could be another implementation of Immediate Maintenance
  – Materialized views are updated at the end of a transaction that modified base tables, rather than in AFTER trigger.
  – Needs “logs” mechanism as well as Deferred.
About counting algorithm

- "__ivm_count__" is treated as a special column name.
  - Maybe this name has to be inhibited in user tables.
  - Is it acceptable to use such columns for IVM, or is there other better way?

- `generate_series` function is used when materialized views with tuple duplicates is accessed:
  - We can make a new set returning function instead of `generate_series`.
  - Performance issues:
    - Planner’s estimation of rows number is wrong.
    - The cost of join with this function could be high.
  → We might have to add a new plan node for IVM materialized views rather than using a set returning function.
Other issues

● Concurrent transactions
  - When concurrent transactions modify base tables under the same materialized view, lock waiting and race condition could occur.
  - Need more investigation

● Optimization
  - “counting” is unnecessary if a view doesn’t have DISTINCT or duplicates.
  - When overhead of IVM is higher than normal REFRESH, we should use the latter.
    - Using cost estimated by optimizer
Summary

• Our implementation of IVM on PostgreSQL
  - Immediate View Maintenance using AFTER trigger
  - Views with tuple duplicates or DISTINCT
    • counting algorithm
• To do:
  - Aggregation and GROUP BY (for the first release of IVM)
  - Deferred Maintenance
  - Concurrent transaction issues
  - Optimizations

• Working-in-Progress patch has been submitted to pgsql-hackers
  - Subject: Implementing Incremental View Maintenance
Thank you