Toward Implementing Incremental View Maintenance on PostgreSQL

Yugo Nagata @ SRA OSS, Inc. Japan.

PGConf.ASIA 2019
- Sep 9, 2019

About Me

- Yugo Nagata
  - Software Engineer at SRA OSS, Inc. Japan
  - R&D
- PostgreSQL experiences
  - Technical support
  - Consulting
  - Education

Outline

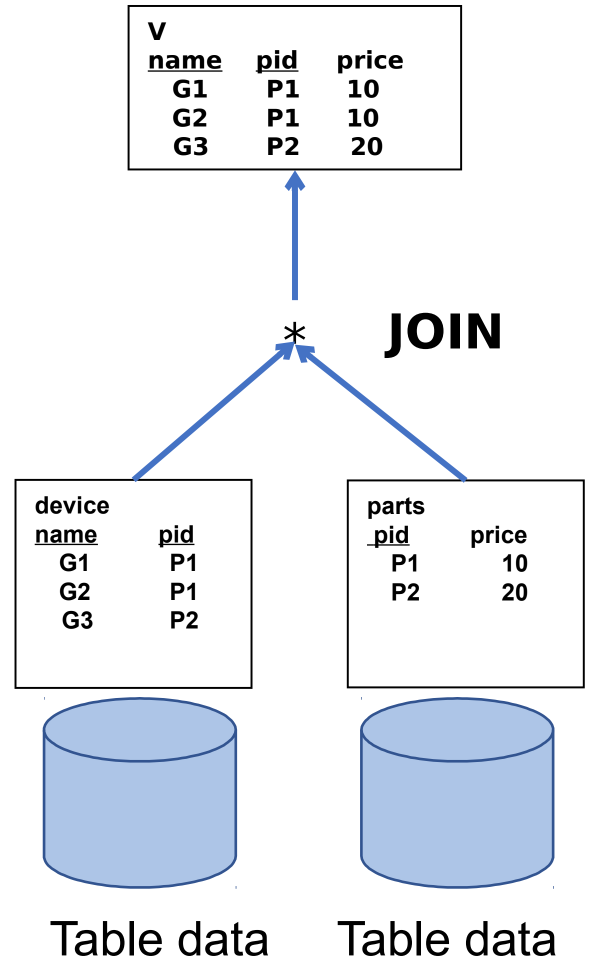
- Introduction
  - Views and materialize views
  - Incremental View Maintenance (IVM)
- Implementing IVM on PostgreSQL
  - What to be considered to implement IVM
  - Our implementation and its details
- Examples
  - Performance Evaluation
- Discussions

What is Incremental View Maintenance (IVM)

Views

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

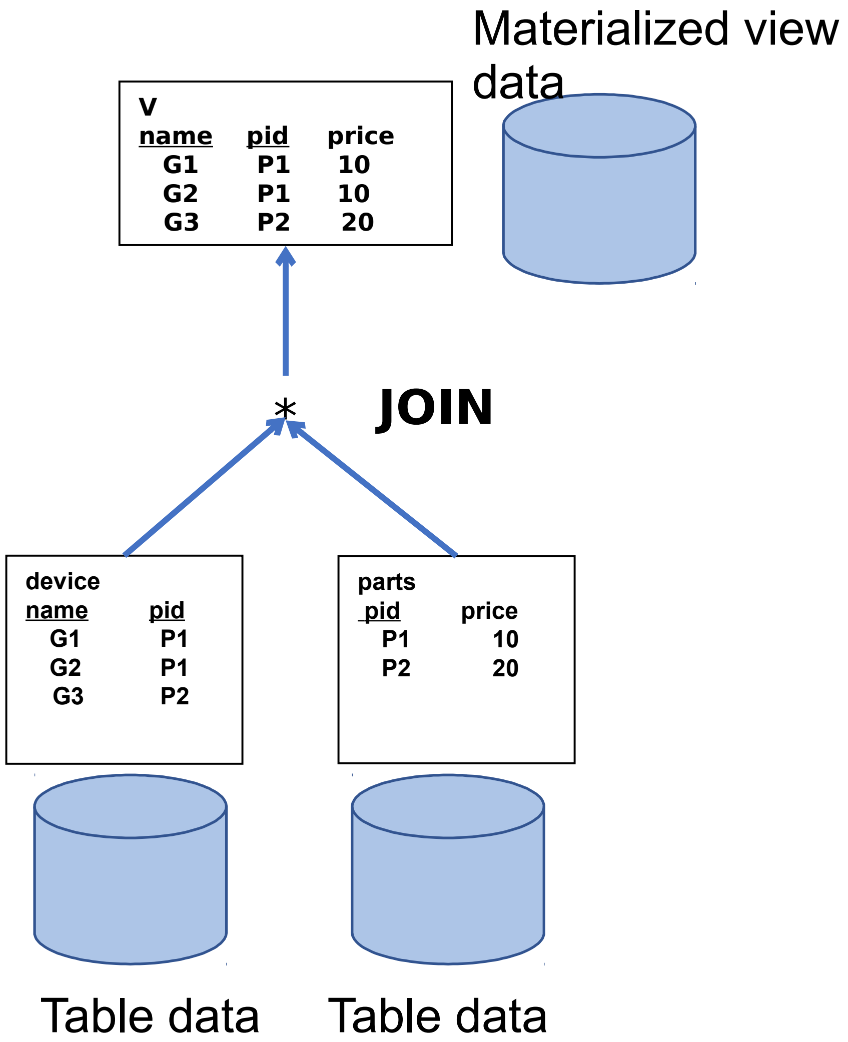
- A view is a virtual relation defined by a query on base tables.
  - Only the definition query is stored.
- The result is computed when a query is issued to a view.



Materialized Views

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

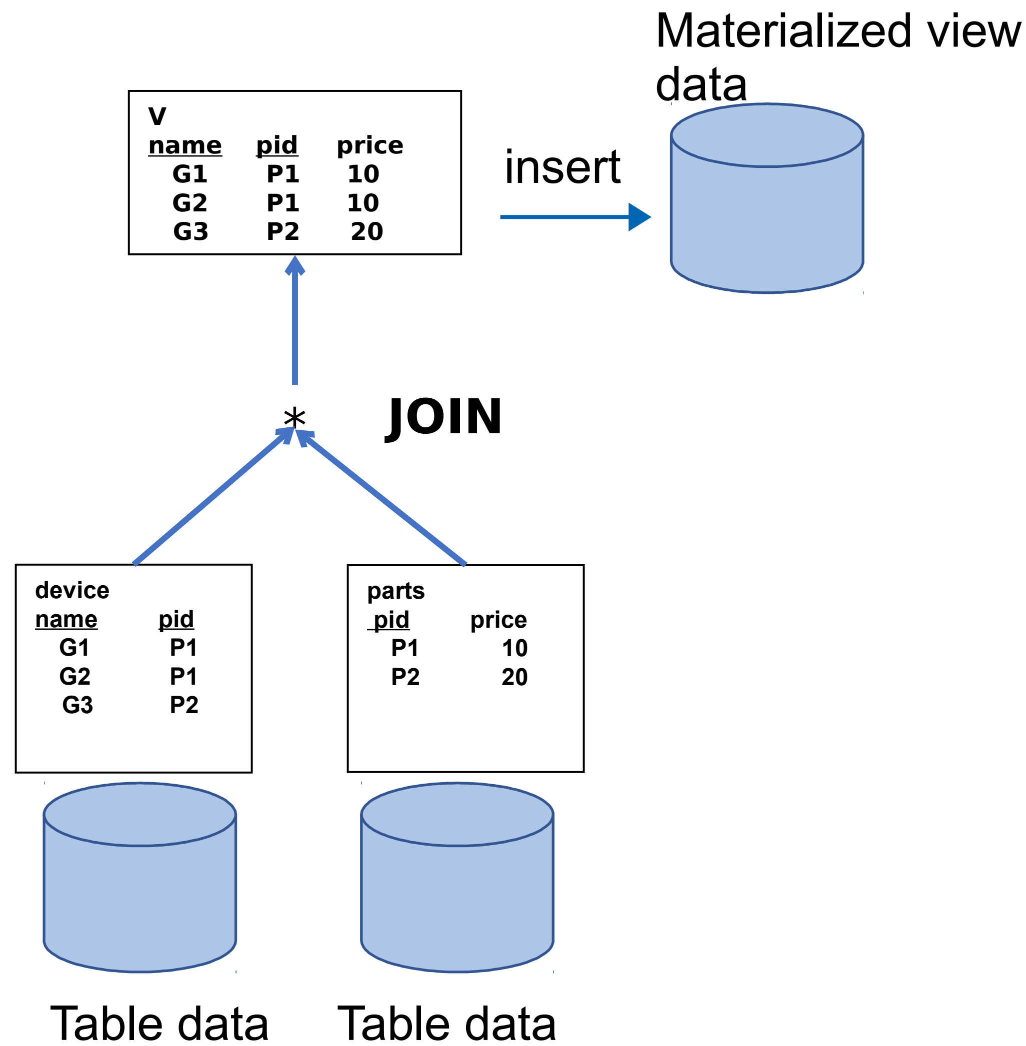
- Materialized views persist the results in a table-like form.
- No need to compute the result when a query is issued.
  - Enables faster access to data.
- The data is not always up to date.
  - Need maintenance.



Creating Materialized Views

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

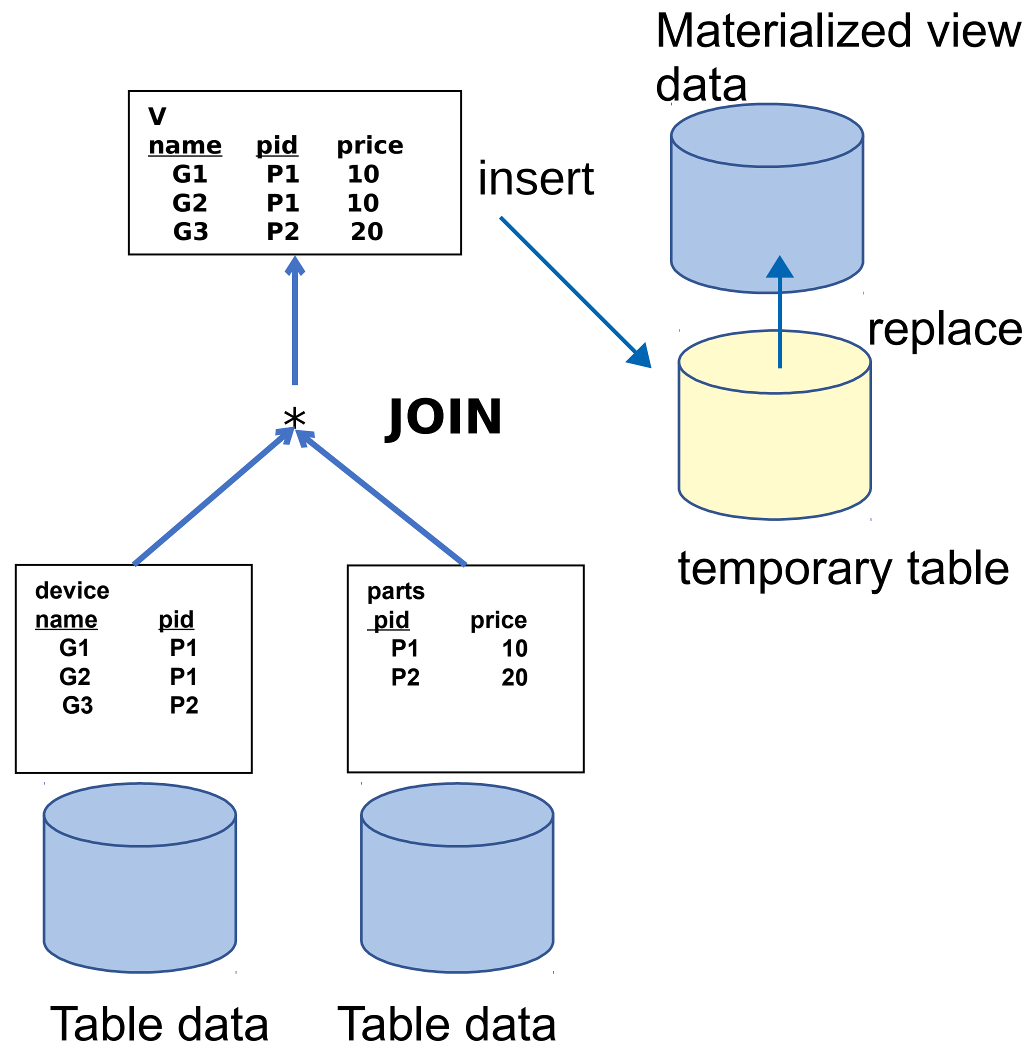
- The data of a materialized view is computed at definition time.
  - This is similar to “CREATE TABLE AS” statement.
  - The result of the definition query is inserted into the materialized view.
- Need maintenance to keep consistency between the materialized data and base tables.



Refreshing Materialized Views

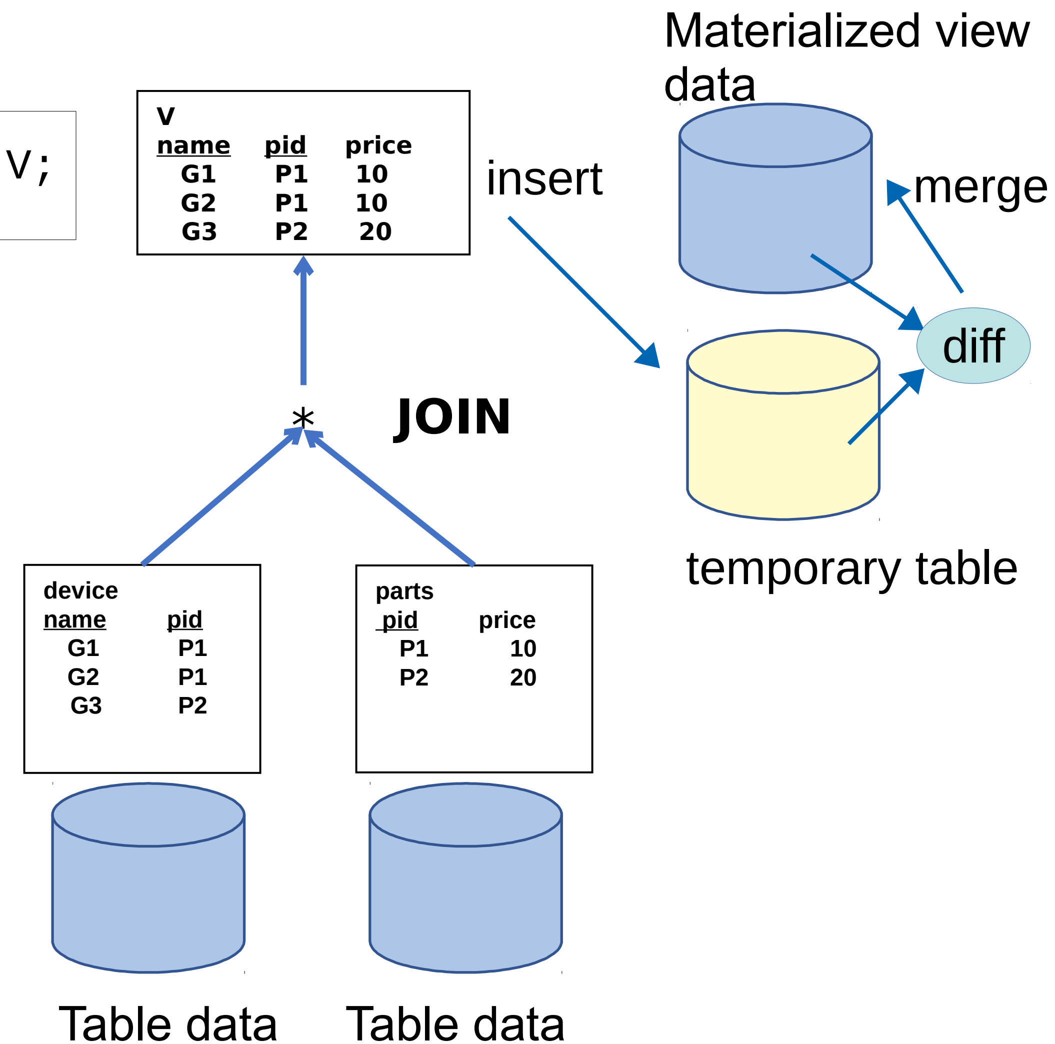
```
REFRESH MATERIALIZED VIEW V;
```

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



Refreshing Materialized Views

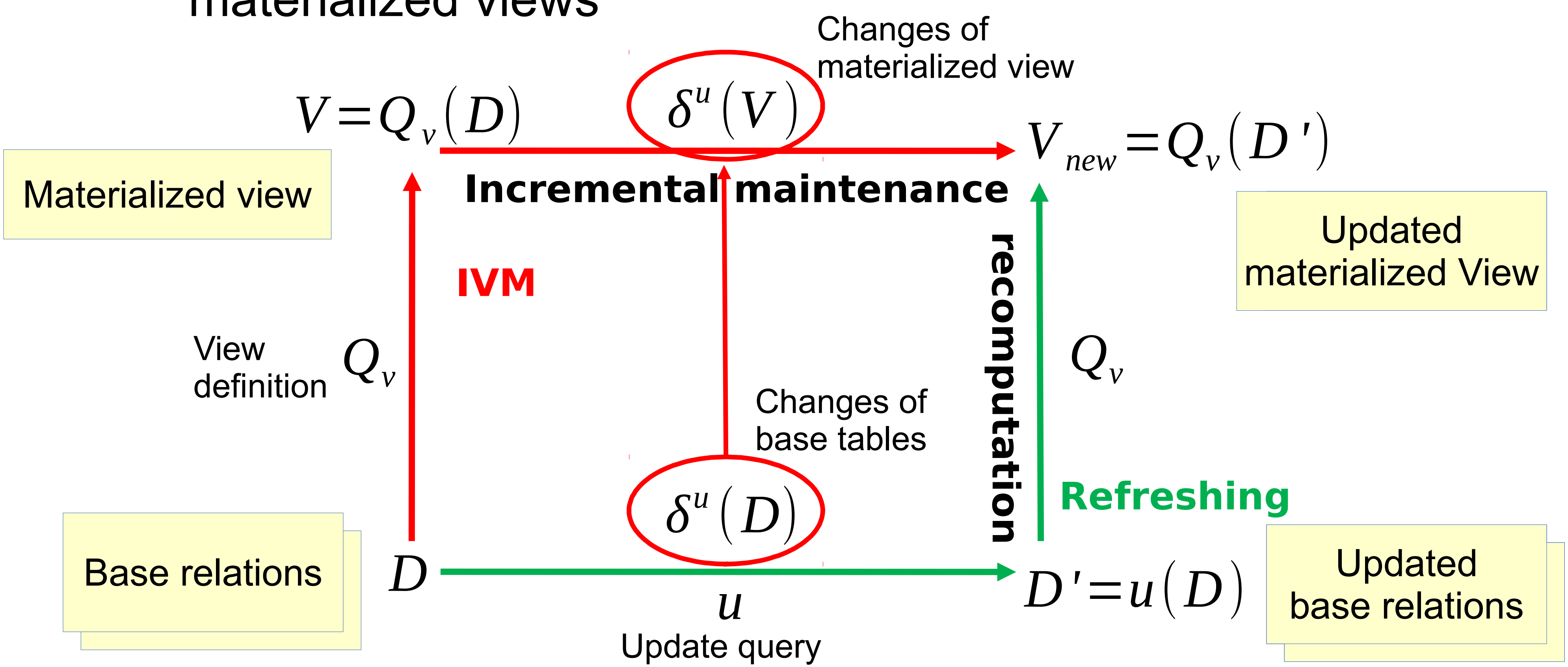
```
REFRESH MATERIALIZED VIEW CONCURRENTLY V;
```



- 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



Basic Theory of IVM

- View definition

```
SELECT * FROM R NATURAL JOIN S;
```

  - Ex.) Natural join view

    $$V \stackrel{\text{def}}{=} R \bowtie S$$

R, S	base tables
∇R	deleted tuples
ΔR	inserted tuples

- 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

  $$V \leftarrow (V - \nabla V \cup \Delta V)$$

Basic Theory of IVM: Example (1)

R

number	english
1	one
2	two
3	three

S

number	roman
1	I
2	II
3	III

natural join

$$V \stackrel{\text{def}}{=} R \bowtie S$$

number	english	roman
1	one	I
2	two	II
3	three	III

Basic Theory of IVM: Example (2)

Table R is changed

$R \leftarrow (R - \nabla R \cup \Delta R)$

number	english
1	one → ONE
2	two
3	three

S

number	roman
1	I
2	II
3	III

Calculate changes on view V

∇R

number	english
1	one

natural join

$\nabla V = \nabla R \bowtie S$

number	english	roman
1	one	I

ΔR

number	english
1	ONE

natural join

$\Delta V = \Delta R \bowtie S$

number	english	roman
1	ONE	I

Basic Theory of IVM: Example (3)

∇V

number	english	roman
1	one	I

ΔV

number	english	roman
1	ONE	I

delete

insert

$$V \leftarrow (V - \nabla V \cup \Delta V)$$

number	english	roman
1	one → ONE	I
2	two	II
3	three	III

View V is update by applying the calculated changes

Implementing IVM on PostgreSQL

Considerations on IVM Implementation(1)

- How to extract changes on base tables
  - AFTER trigger and Transition Tables
  - Another idea is logical decoding of WAL

- How to compute the delta to be applied to materialized views
  - Basically, based on relational algebra (or bag algebra).
  - Starting from simpler view definitions:
    - Selection-Projection-Join views
    - Some aggregate functions and GROUP BY

Considerations on IVM Implementation(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.

- Views with tuple duplicates or DISTINCT clause

Views with Tuple Duplicates

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

V

english	roman
one	I
two	II
two	II
three	III

delete

∇V

english	roman
two	II

- Only one tuple of duplicated two must be deleted.
- DELETE statement can not be used because this delete two tuples.

Views with DISTINCT Clause

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

V

english	roman
one	I
two	II
three	III

delete?

insert?

∇V

english	roman
two	II

ΔV

english	roman
three	III

- 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.

Our 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
  - By using "counting algorithm"

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.

V

english	roman	count
one	I	1
two	II	2
three	III	1

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.
    - Tuples to be inserted into the view → increment the count
    - Tuples to be deleted from the view → decrement the count
    - If the count becomes zero, this tuple can be completely deleted.

V

english	roman	count
one	I	1
two	II	2 → 1
three	III	1 → 2

delete

insert

∇V

english	roman	count
two	II	1

ΔV

english	roman	count
three	III	1

Implementation Details

Creating Materialized Views (1)

- CREATE INCREMENTAL MATERIALIZED VIEW
  - The tentative syntax to creates materialized views with IVM support
    - Views are updated automatically and incrementally after base tables are changed

```
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 materialized views, rewritten view definition query is used.
  - 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__".

```
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 all base tables.
  - For INSERT, DELETE, and UPDATE command
  - Statement level
  - With Transition Tables
- Triggers are Created automatically and internally rather than issuing CREATE TRIGGER statement directly.
  - Similar to the implementation of foreign key constrains

Example of an equivalent query:

```
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 to $\nabla \mathrm{R}$ and $\Delta \mathrm{R}$ respectively.

Calculating Delta on Views

- Calculate the delta on materialized views by rewriting view query
  - Replacing the base table with the transition table.
  - Using count(*) and GROUP BY in order to count the duplicity of tuples.
- The results are stored into temporary tables .
  - “old delta” and “new delta” corresponding to ∇V and ΔV, respectively.

```
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 Delta to View (1)

- Update the view by merging calculated delta tables.
  - For each tuple in delta tables :
    - If the corresponding tuple already exists, the value of __ivm_count__ is updated
      - decrement for old delta, increment for new delta
    - When the values becomes zero, the corresponding tuple should be deleted.
    - If a tuple in new delta doesn’t exist in the view, insert this into the view.
  - Using modifying CTE (WITH clause)
    - Building SQL strings and execute these via SPI.

Applying Delta to View (2)

- Old delta: decrement __ivm_count__, or delete an old tuple

```
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 Delta to View (3)

- New delta: increment __ivm_count__, or insert a new tuple

```
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.pric)
         NOT IN (SELECT * FROM updt));
```

Aggregate Functions Support

- Supporting, count, sum, min, max, avg
  - with or without GROUP BY
- Expressions specified in GROUP BY must appear in the target list of views.
- In addition to __ivm_count__, one or more extra hidden columns are added to the view.
  - For example, __ivm_count_avg__ and __ivm_sum_avg__ are added for avg function.
- Aggregates are performed on delta tables, and aggregated values in the view are updated using the results
  - The way of updating depends on the kind of aggregate function.

Updating Aggregated Values

- count(x) ← count(x) ± [count(x) from delta table]
- sum(x) ← sum(x) ± [sum(x) from delta table]
  - However, this becomes NULL if count(x) results in 0.
- avg(x) ← (sum(x) ± [sum(x) from delta]) / (count(x) ± [count(x) from delta])
  - NULL if count(x) results in 0.
- min(x)
  - When tuples are inserted:
    - Use the smaller one between the current min value in the view and the min value calculated from the new delta table.
  - When tuples are deleted:
    - If the current min value equals to the min from the old delta table, it needs re-computation.
    - Otherwise, the current value remains.

Access to Materialized Views

- When SELECT is issued for materialized views with IVM:
  - case 1: Defined with DISTINCT:
    - All columns (except to __ivm_*) 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.

```
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;
 i
---
 3
 2
 1
(3 rows)

postgres=# INSERT INTO t0 VALUES (4);
INSERT 0 1
postgres=# SELECt * FROM m;
 i
---
 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

```
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 ORDER BY t;
 t
---
 A
 A
 B
 C
(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 ORDER BY t;
 t
---
 A
 B
 B
(3 rows)
```

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

The view with tuple duplicates is correctly updated.

```
Before:

 t
---
 A
 A
 B
 C
(4 rows)
```

Example 3

```
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.

```
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:

Scale factor of pgbench: 100

```
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;
```

SRA OSS, INC.

Simple Performance Evaluation (2)

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

The standard REFRESH of mv_normal took more than 10 seconds.

```
test=# CREATE INDEX on mv_ivm (aid,bid);
CREATE INDEX
test=# SELECT * FROM mv_ivm WHERE aid = 1;
 aid | bid | abalance | bbalance
-----+-----+----------+----------
   1 |   1 |       10 |       10
(1 row)

Time: 2.498 ms
```

Creating an index on mv_ivm

```
test=# UPDATE pgbench_accounts SET abalance = 1000 WHERE aid = 1;
UPDATE 1
Time: 18.634 ms
```

Updating a tuple in pgbench_accounts took 18ms.

```
test=# SELECT * FROM mv_ivm WHERE aid = 1;
 aid | bid | abalance | bbalance
-----+-----+----------+----------
   1 |   1 |     1000 |       10
(1 row)
```

mv_ivm was updated automatically and correctly.

SRA OSS, INC.

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.

Simple Performance Evaluation (4)

- Materialized views of aggregates on pgbench_accounts

Scale factor of pgbench: 1000

```
CREATE MATERIALIZED VIEW mv_normal2 AS
        SELECT bid, count(abalance), sum(abalance), avg(abalance)
        FROM pgbench_accounts GROUP BY bid;
```

```
CREATE INCREMENTAL MATERIALIZED VIEW mv_ivm2 AS
        SELECT bid, count(abalance), sum(abalance), avg(abalance)
        FROM pgbench_accounts GROUP BY bid;
```

Simple Performance Evaluation (5)

```
test=# REFRESH MATERIALIZED VIEW mv_normal2 ;
REFRESH MATERIALIZED VIEW
Time: 30494.729 ms (00:30.495)
```

The standard REFRESH of mv_normal2 took 30 seconds.

```
test=# SELECT * FROM mv_ivm2 WHERE bid = 1;
 bid | count  |  sum  |          avg
-----+--------+-------+------------------------
   1 | 100000 | -1855 | -0.01855000000000000000
(1 row)

test=# UPDATE pgbench_accounts SET abalance = abalance + 1000 WHERE aid = 1;
UPDATE 1
Time: 30.215 ms
```

Updating a tuple in pgbench_accounts took 30 ms.

x 1000 faster!

```
test=# SELECT * FROM mv_ivm2 WHERE bid = 1;
 bid | count  | sum  |          avg
-----+--------+------+------------------------
   1 | 100000 | -855 | -0.00855000000000000000
(1 row)
```

mv_ivm2 was updated automatically and correctly.

Current Restrictions

- Supported:
  - selection, projection, inner join, DISTINCT
  - Some aggregate functions and GROUP BY
    - count, sum, avg, min/max
- Not supported:
  - Other aggregates, HAVING
  - Self-join, sub-query
  - outer join
  - CTE, window functions
  - Set operations (UNION, EXCEPT, INTERSECT)

- We are now working on self-join, outer-join, and sub-query.

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.
  - There are additional __ivm_* columns for aggregate views.
  - Users can not use these names in materialized views supporting IVM.
  - This restriction is not applied to tables, views, or normal materialized views.

- 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.

Concurrent Transactions

- When concurrent transactions modify different base tables under a materialized view, we need to prevent update anomalies on the materialized view.

- In READ COMMITTED
  - Lock the materialized view to wait for concurrent transactions to finish.
  - Update the view by referring table changes which occurs in other transactions during lock waiting.
- In REPEATABLE READ or SERIALIZABLE
  - Table changes occurred in other transactions must not be visible, and views can not be maintained correctly in AFTER triggers.
  - When competing transactions are detected, raise an error and abort immediately.

Other Issues

- Performance improvements
  - Reducing CREATE/DROP of temporary tables
    - Using tuplestore instead
  - Query execution for applying delta to views
    - Using plan cache, converting to C rather than issuing SQL, etc.

- Optimizations
  - Detecting “Irrelevant Update”
    - Table changes which leave the materialized view unchanged
  - “counting” is unnecessary if a view doesn’t have DISTINCT or duplicates.
  - When the overhead of IVM is higher than normal REFRESH, it would be better to use the latter.
    - Cost estimation of optimizer may be usable.

Summary

- Our implementation of IVM on PostgreSQL
  - Immediate View Maintenance using AFTER trigger
  - Views with tuple duplicates or DISTINCT
    - counting algorithm
  - Some aggregates and GROUP BY
- Future works:
  - Support self-join and sub-queries (in progress)
  - Deferred Maintenance using table change logs
  - Performance improvement and optimizations

- Working-in-Progress patch has been submitted to pgsql-hackers
  - Subject: Implementing Incremental View Maintenance
  - Github: https://github.com/sraoss/pgsql-ivm/

Thank you