Implementing Incremental View Maintenance on PostgreSQL

Yugo Nagata

PGConf.eu 2018
2018.10.26

Who am I?

- Yugo Nagata
  - Chief engineer @ SRA OSS, Inc. Japan
  - PostgreSQL
    - Technical support
    - Consulting
    - Education
    - R&D

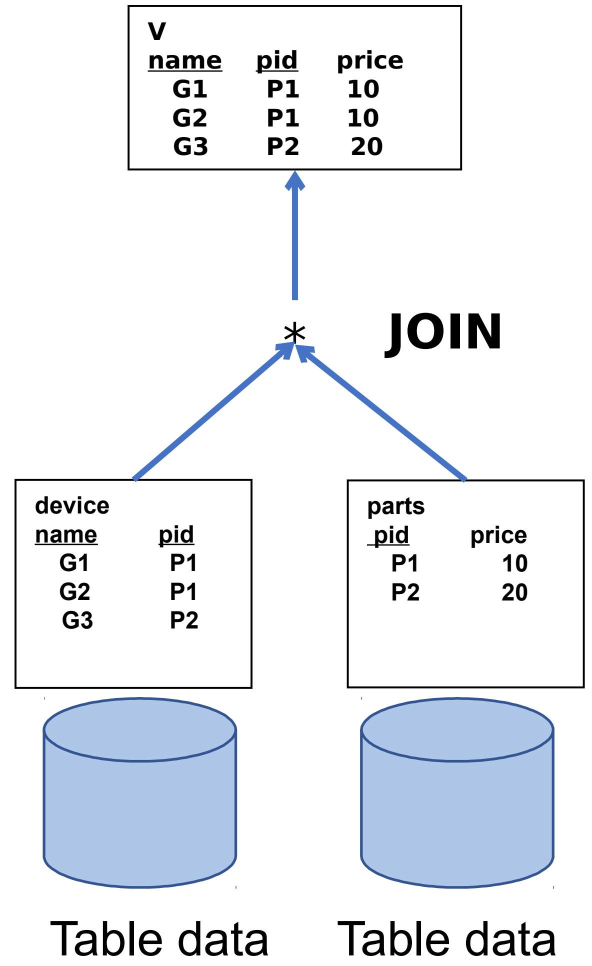
Outline

- Introduction
  - Views
  - Materialize views
- Incremental View maintenance (IVM)
  - Some approaches and our idea
- PoC Implementation of IVM
  - Overview and details
  - Examples
  - Performance Evaluation

Views

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

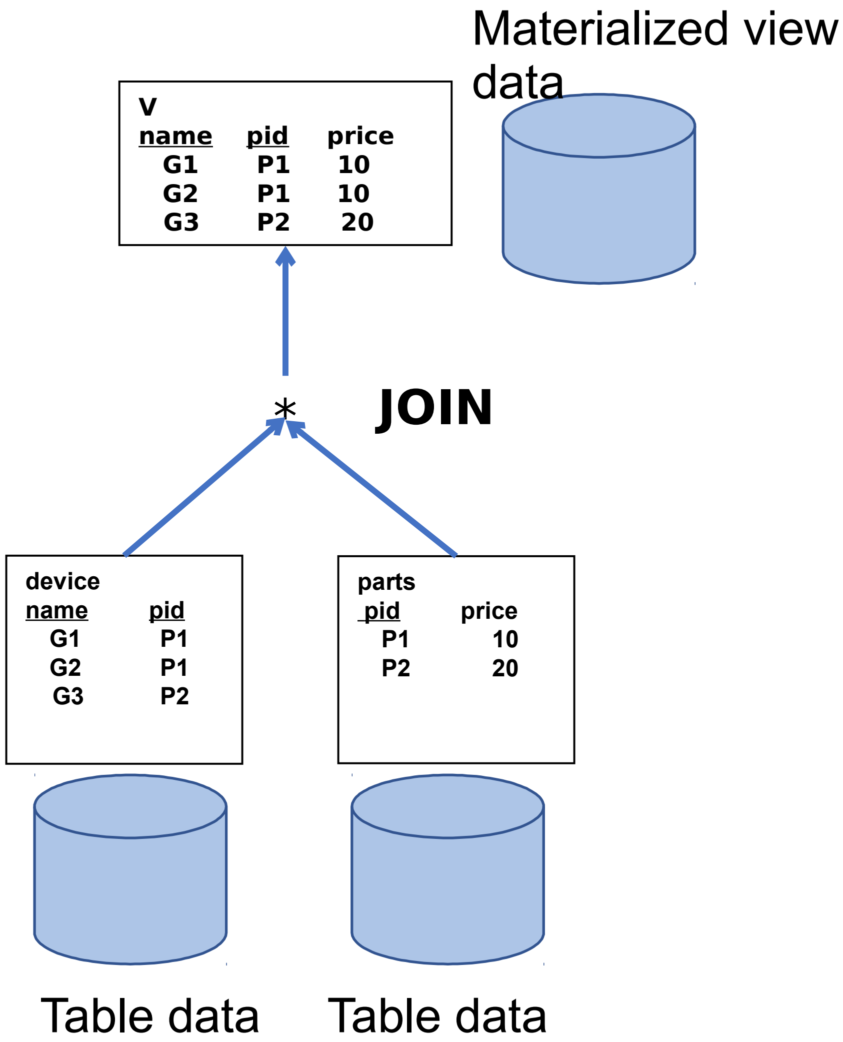
- A view is defined by a query on base tables.
  - Only the definition query is stored instead of contents of results.
- The result is computed when a query is issued to a view.



Materialized Views

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

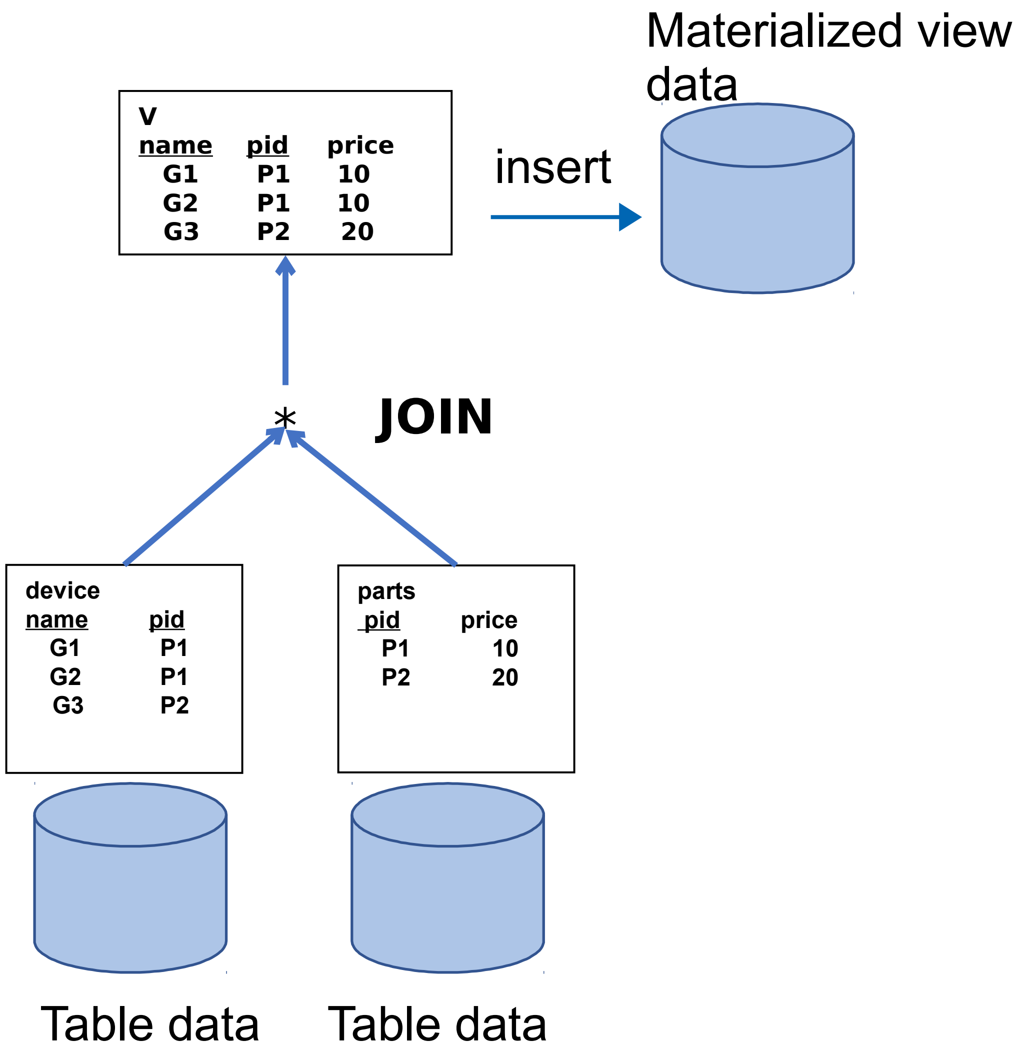
- Materialized views persist the results in a table-like form.
- No need to compute the result when a query is issued.
  - Enables faster access to data.
- The data is not always up to date.
  - Need maintenance.



Creating Materialized Views

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

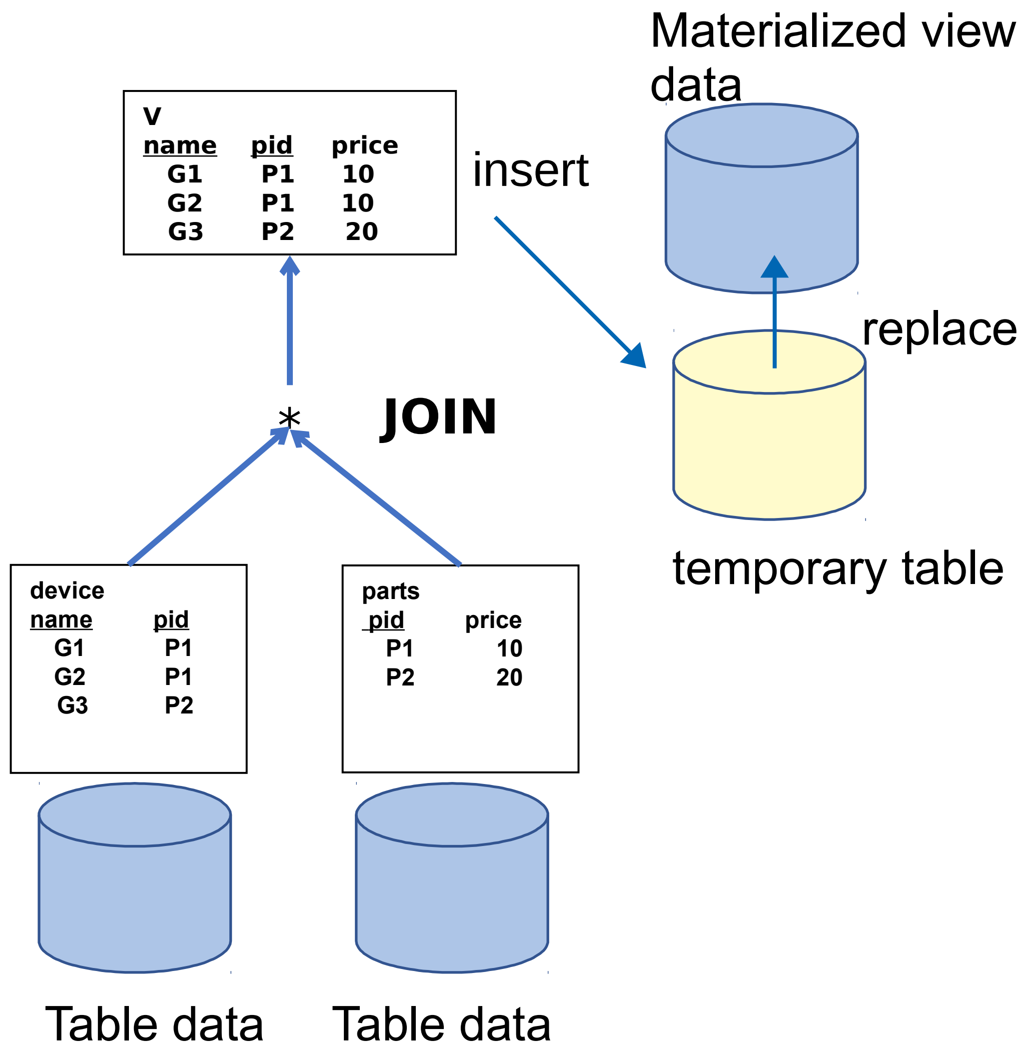
- The data of a materialized view is computed at definition time.
  - This is similar to "CREATE TABLE AS" statement.
  - The result of the definition query is inserted into the materialized view.
- Need maintenance to keep consistency between the materialized data and base tables.



Refreshing Materialized Views

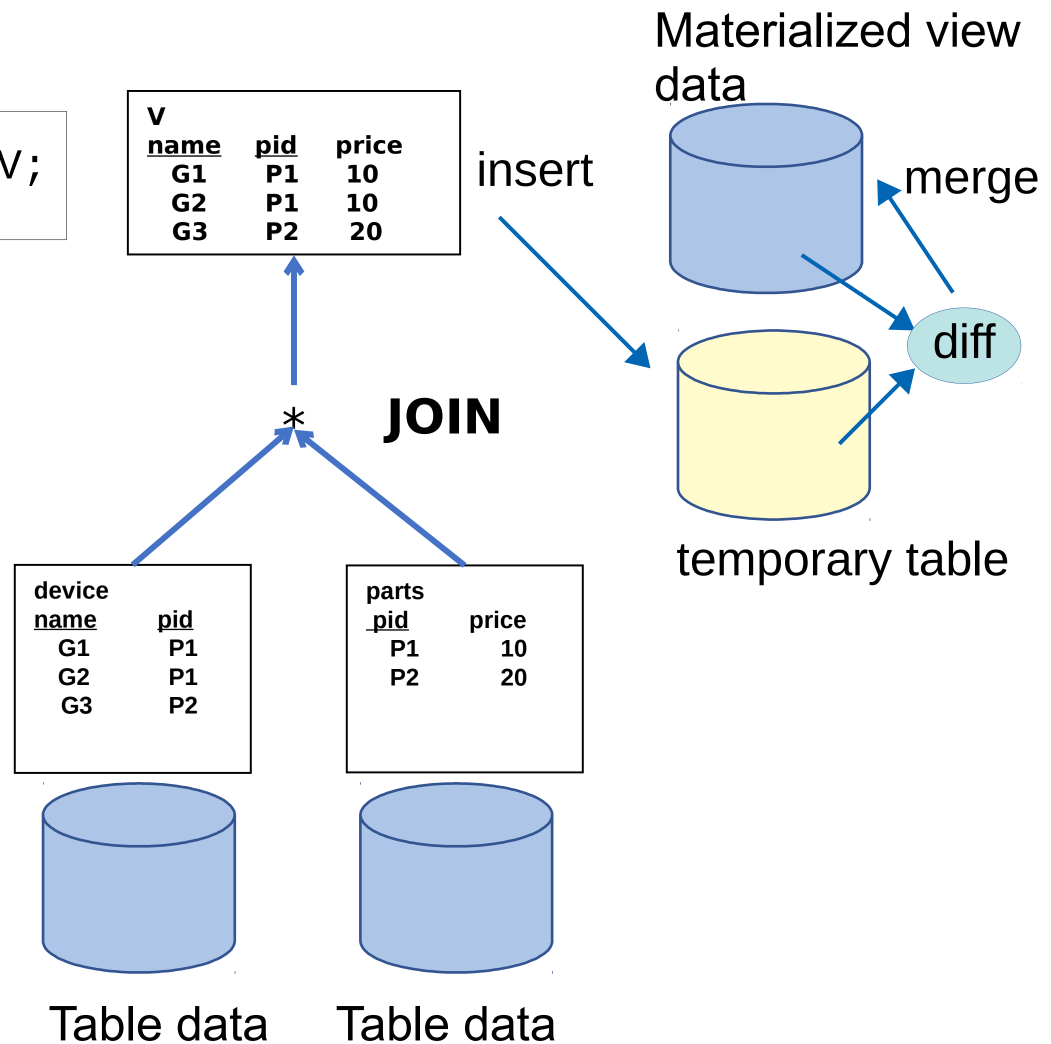
```
REFRESH MATERIALIZED VIEW V;
```

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



Refreshing Materialized Views

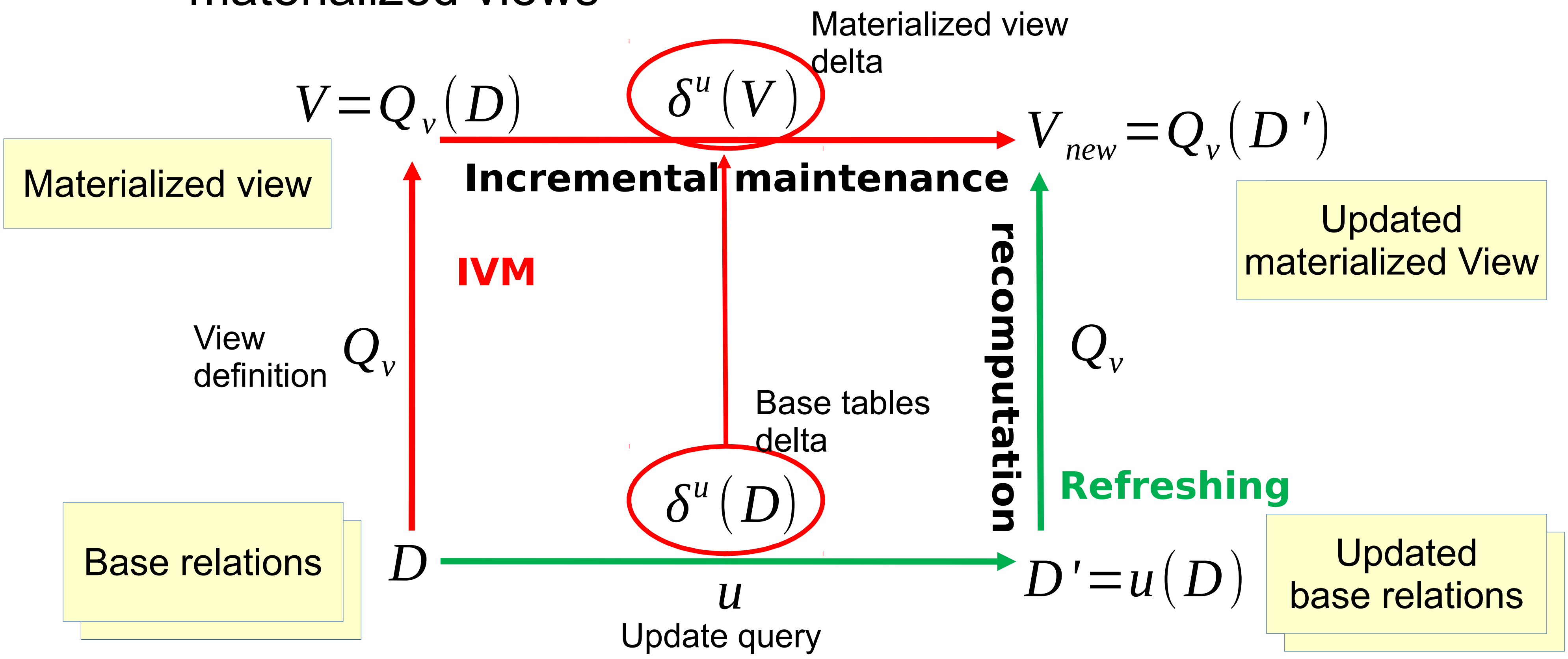
```
REFRESH MATERIALIZED VIEW CONCURRENTLY V;
```



- With CONCURRENTLY option, the materialized view is refreshed without locking out concurrent selects on the view.
  - Require at least one UNIQUE index on the materialized 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



Approaches to IVM

- Tuple-based
  - Specify the view tuples that need to be modified by computing diff sets.
- ID-based (Katsis et al., 2015)
  - Assume the base tables have primary keys and these are propagated to the materialized view.
  - Identify the to-be-modified tuples in the view through their IDs.
    - More efficient than Tuple-based
- OID-based (Masunaga et al., 2018)
  - Use OIDs to identify tuples rather than primary keys.
    - Allow to handle bag (multi-set) semantics.
  - OID is a system column of a tuple in PostgerSQL
    - Users do not need to concern about this.

Proof of Concept (PoC) implementation of IVM using OIDs on PostgreSQL

Basic idea

V

OID	device_name	pid	price
301	device1	P1	10
302	device2	P2	20
303	device3	P2	20

JOIN

devices

OID	device_name	pid
101	device1	P1
102	device2	P2
103	device3	P2

parts

OID	pid	parts_name	price
201	P1	part1	10
202	P2	part2	20

oid map

OID in matview	OIDs in base tables
301	101, 201
302	102, 202
303	103, 202

Basic idea

V

OID	device_name	pid	price
301	**device1**	**P1**	**15**
302	device2	P2	20
303	device3	P2	20

oid map

OID in matview	OIDs in base tables
301	101, **201**
302	102, 202
303	103, 202

JOIN

devices

OID	device_name	pid
101	device1	P1
102	device2	P2
103	device3	P2

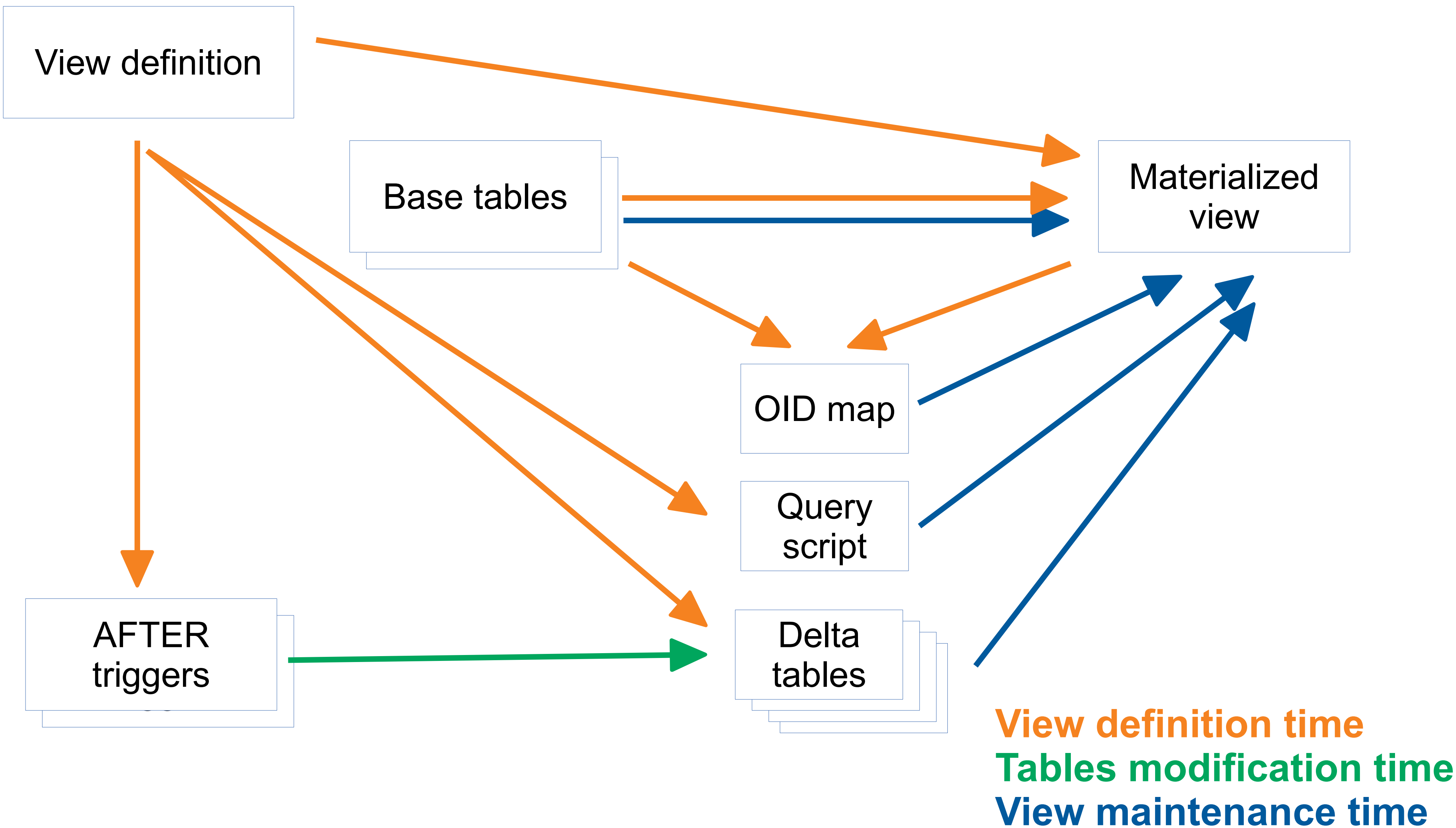
parts

OID	pid	parts_name	price
201	**P1**	**part1**	**15**
202	P2	part2	20

If a tuple in parts table with OID=201 is updated,
only a tuple in the materialized view with OID=301 is affected.

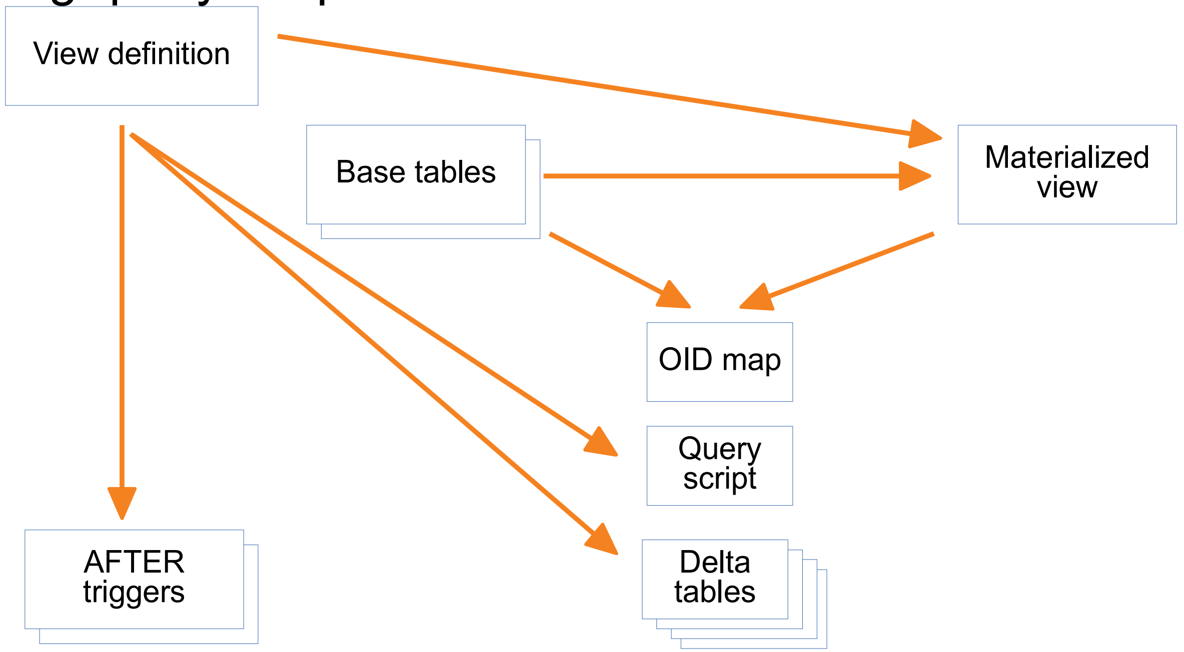
PoC implementation of Incremental View Maintenance using OIDs

Overview


View definition
Base tables
Materialized view
OID map
Query script
AFTER triggers
Delta tables
View definition time
Tables modification time
View maintenance time

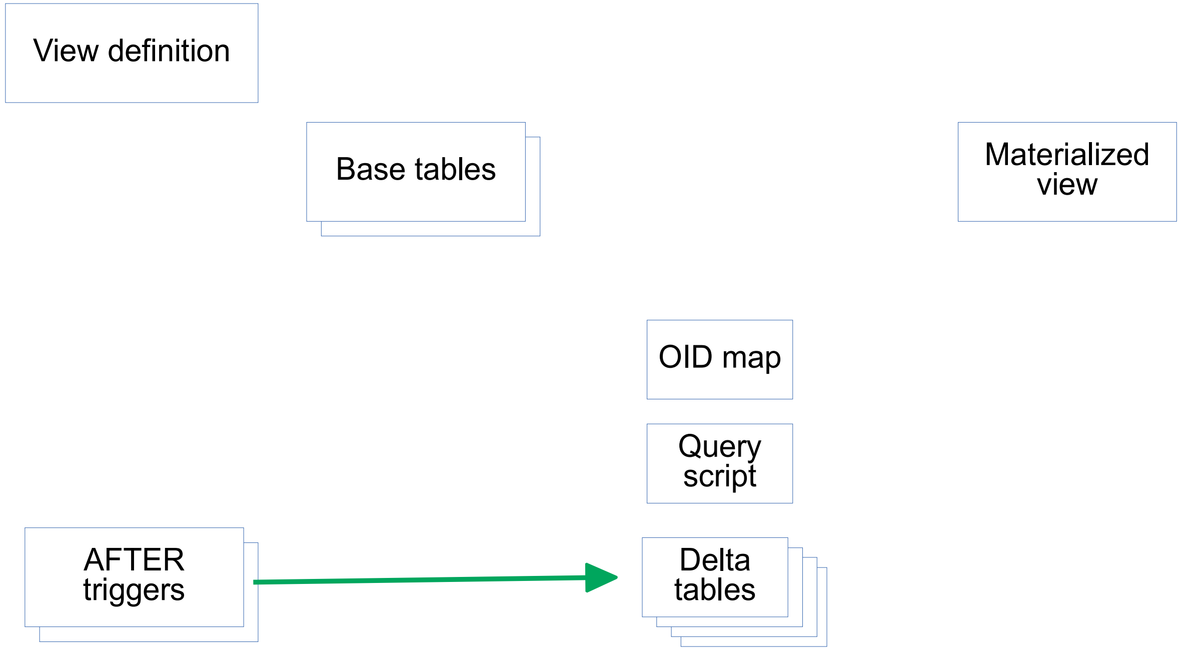
Overview

1. View definition time
   - Computing the materialized view data.
   - Creating OID map between the base tables and the materialized view.
   - Creating delta tables for the base tables.
   - Creating AFTER triggers on the base tables.
   - Generating query scripts to be run at view maintenance time.



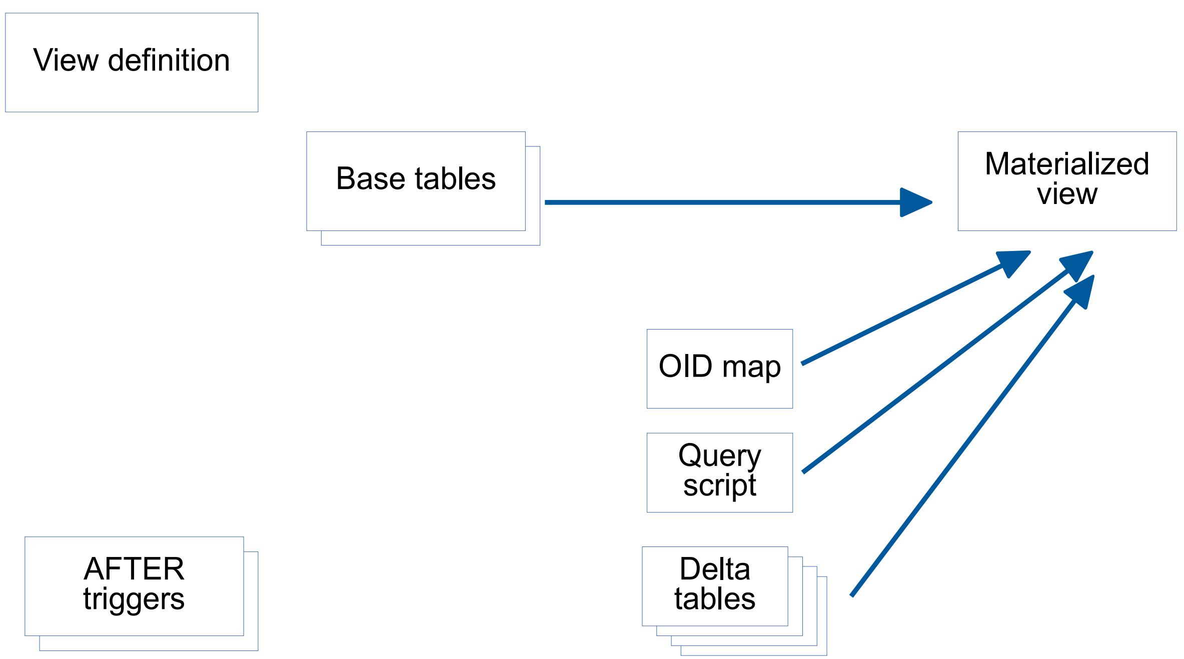
Overview

2. Table modification time
   - Logging changes on the base tables into the delta tables.

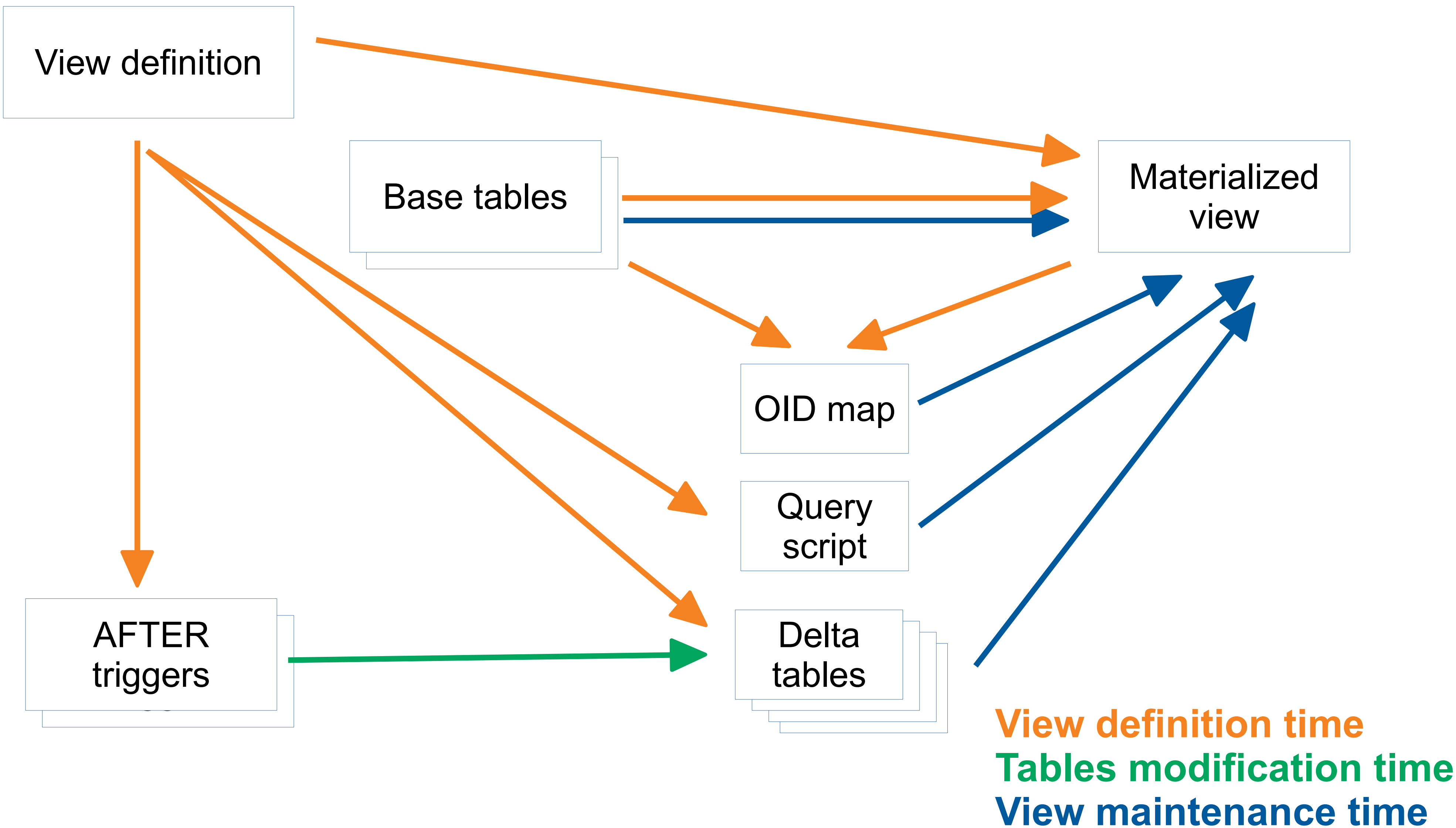


Overview

3. View maintenance time
   – Executing the query scripts to perform incremental maintenance of the materialized view.



Overview


View definition
Base tables
Materialized view
OID map
Query script
Delta tables
AFTER triggers
View definition time
Tables modification time
View maintenance time

Before creating materialized views...

- Base tables have to have OIDs.
  - Create tables with oids option

```
CREATE TABLE mytable (i int) WITH OIDS;
```

  - or use ALTER TABLE

```
ALTER TABLE mytable SET WITH OIDS;
```

- System tables for storing IVM metadata
  - pg_ivm_oidmap
    - Mapping row OIDs in materialized view and base relations
  - pg_ivm_script
    - Storing query scripts to be executed in view maintenance time
  - pg_ivm_deltamap
    - Mapping table OIDs of base relations and their delta tables

Materialized Views with OIDs

- Materialized views also have to be defined with OIDs.
  - The current PostgreSQL implementation doesn't support materialized views with OIDs.
  - Our PoC implementation allows materialized views to have OIDs.

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

View Definition Time (1) Creating OID map

- Row OIDs are collected during executing the SELECT query.

V (relation OID: 3333)

OID	device_name	pid	price
301	device1	P1	10
302	device2	P2	20
303	device3	P2	20

insert

JOIN

devices (relation OID: 1111)

OID	device_name	pid
101	device1	P1
102	device2	P2
103	device3	P2

parts (relation OID: 2222)

OID	pid	parts_name	price
201	P1	part1	10
202	P2	part2	20

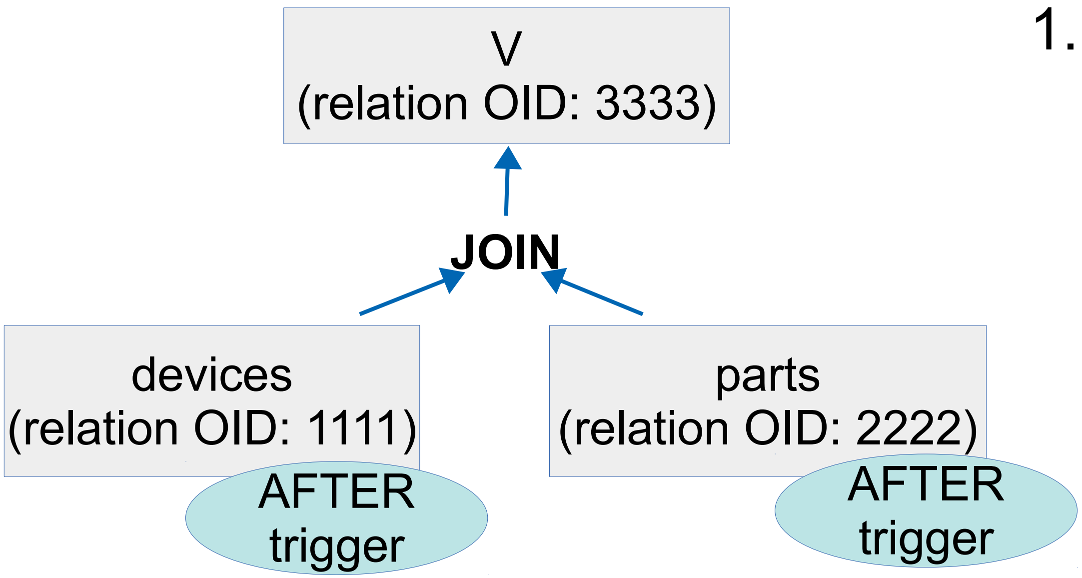
insert

pg_ivm_oidmap

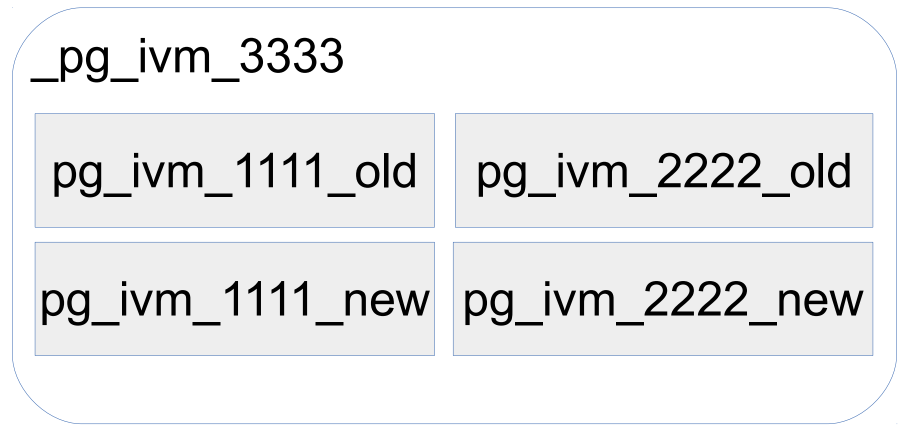
viewrelid	viewoid	baserelid	baseoid
3333	301	1111	101
3333	301	2222	201
3333	302	1111	102
3333	302	2222	202
3333	303	1111	103
3333	303	2222	202

View Definition Time (2) Query tree analysis

- Extract base tables from the view definition query.



1. Create a schema and delta tables



2. Create AFTER triggers on base tables.

3. Generate query scripts to be run at view maintenance time.

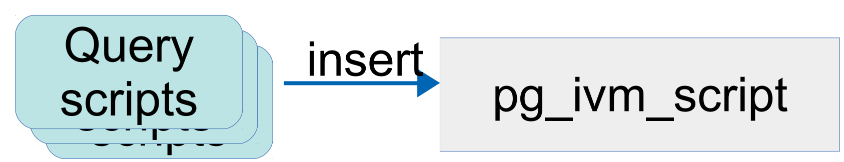


Table Modification time (1)

- AFTER trigger on the base table is executed.
  - OLD delta tuples are inserted into pg_ivm_xxx_old.
    - Deleted tuples
    - Old contents of updated tuples
  - NEW delta tuples are inserted into pg_ivm_xxx_new.
    - Inserted tuples
    - New contents of updated tuples

parts (relation OID: 2222)

OID	pid	parts_name	price
201	P1	part1	**10 → 15**
202	P2	part2	20

pg_ivm_2222_old

OID	pid	parts_name	price
201	P1	part1	**10**

pg_ivm_2222_new

OID	pid	parts_name	price
201	P1	part1	**15**

Table Modification time (2)

- AFTER trigger on the base table is executed.
  - When tuples are deleted or updated, old contents in the new delta table are dropped.
    - This allows a table to be modified multiple times.

parts (relation OID: 2222)

OID	pid	parts_name	price
201	P1	part1	**15 → 25**
202	P2	part2	20

pg_ivm_2222_old

OID	pid	parts_name	price
201	P1	part1	**10**
201	P1	part1	**15**

pg_ivm_2222_new

OID	pid	parts_name	price
~~201~~	~~P1~~	~~part1~~	~~**15**~~
201	P1	part1	**25**

deleted

View maintenance time (1)

- Syntax for Incremental View Maintenance (provisional)

```
REFRESH MATERIALIZED VIEW INCREMENTAL V;
```

- Execute query scripts in pg_ivm_query.

1. Delete old tuples from the materialized view

pg_ivm_2222_old

OID	pid	parts_name	price
201	P1	part1	10

pg_ivm_oidmap

deleted

viewrelid	viewoid	baserelid	baseoid
3333	301	1111	101
~~3333~~	~~301~~	~~2222~~	~~201~~
3333	302	1111	102
3333	302	2222	202
3333	303	1111	103
3333	303	2222	202

V (relation OID: 3333)

OID	device_nam e	pid	price
~~301~~	~~device1~~	~~P1~~	~~10~~
302	device2	P2	20
303	device3	P2	20

deleted

View maintenance time (2)

2. Insert new tuples into the materialized view
   - JOIN of a base table and a NEW delta table results in new tuples to-be-inserted into the materialized view.

OID	device_name	pid	price
304	device1	P1	**15**

insert

V (relation OID: 3333)

OID	device_name	pid	price
302	device2	P2	20
303	device3	P2	20

(filtering if WHERE clause exits)

JOIN

devices (relation OID: 1111)

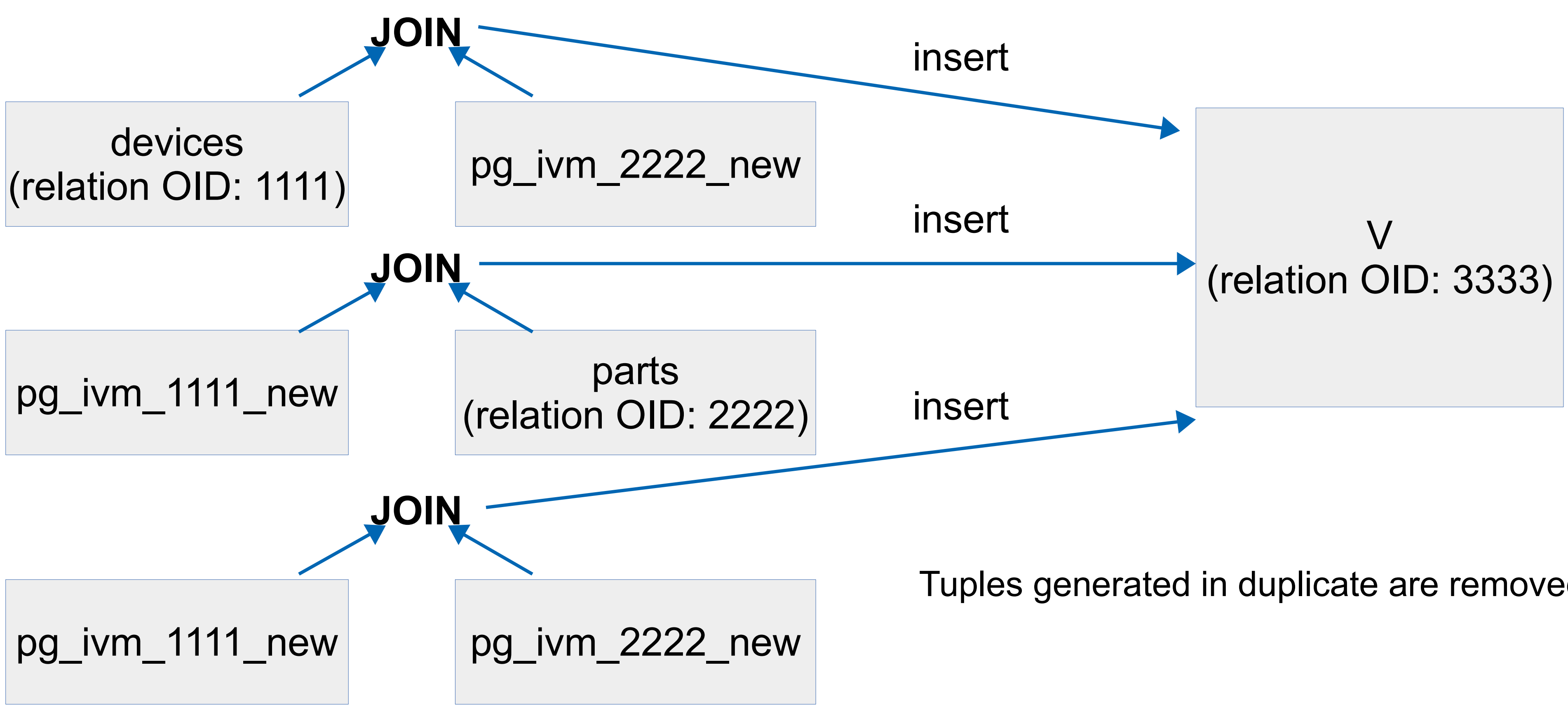
OID	device_name	pid
101	device1	P1
102	device2	P2
103	device3	P2

pg_ivm_2222_new

OID	pid	parts_name	price
201	P1	part1	**15**

View maintenance time (3)

- When both tables in JOIN are modified :
  - Three JOIN results are inserted into the materialized view.



Tuples generated in duplicate are removed.

Examples (1)

- Materialized views of a simple join using pgbench tables:

Standard materialized view:

```
CREATE MATERIALIZED VIEW mv_normal AS
  SELECT a.aid, a.abalance, t.tbalance
    FROM pgbench_accounts a
      JOIN pgbench_tellers t ON a.bid = t.bid
    WHERE t.tid in (1,2,3) ;
```

IVM materialized view:

```
CREATE MATERIALIZED VIEW mv_ivm WITH OIDS AS
  SELECT a.aid, a.abalance, t.tbalance
    FROM pgbench_accounts a
      JOIN pgbench_tellers t ON a.bid = t.bid
    WHERE t.tid in (1,2,3) ;
```

Scale factor of pgbench: 500

- pgbench_accounts: 50,000,000 rows
- pgbench_tellers: 5,000 rows
- Materialized view: 300,000 rows

Examples (2)

- Updating pgbench_accounts:

```
ivm_demo=# UPDATE pgbench_accounts SET abalance = abalance + 1 WHERE aid = 1;
UPDATE 1
Time: 9.749 ms

ivm_demo=# REFRESH MATERIALIZED VIEW mv_normal;
REFRESH MATERIALIZED VIEW
Time: 39979.546 ms (00:39.980)

ivm_demo=# REFRESH MATERIALIZED VIEW INCREMANTALLY mv_ivm;

REFRESH MATERIALIZED VIEW
Time: 537.591 ms

ivm_demo=# SELECT count(1) FROM (
            (SELECT * FROM mv_normal EXCEPT SELECT * FROM mv_ivm)
             UNION ALL
            (SELECT * FROM mv_ivm EXCEPT SELECT * FROM mv_normal)) q;
 count
-------
     0
(1 row)
```

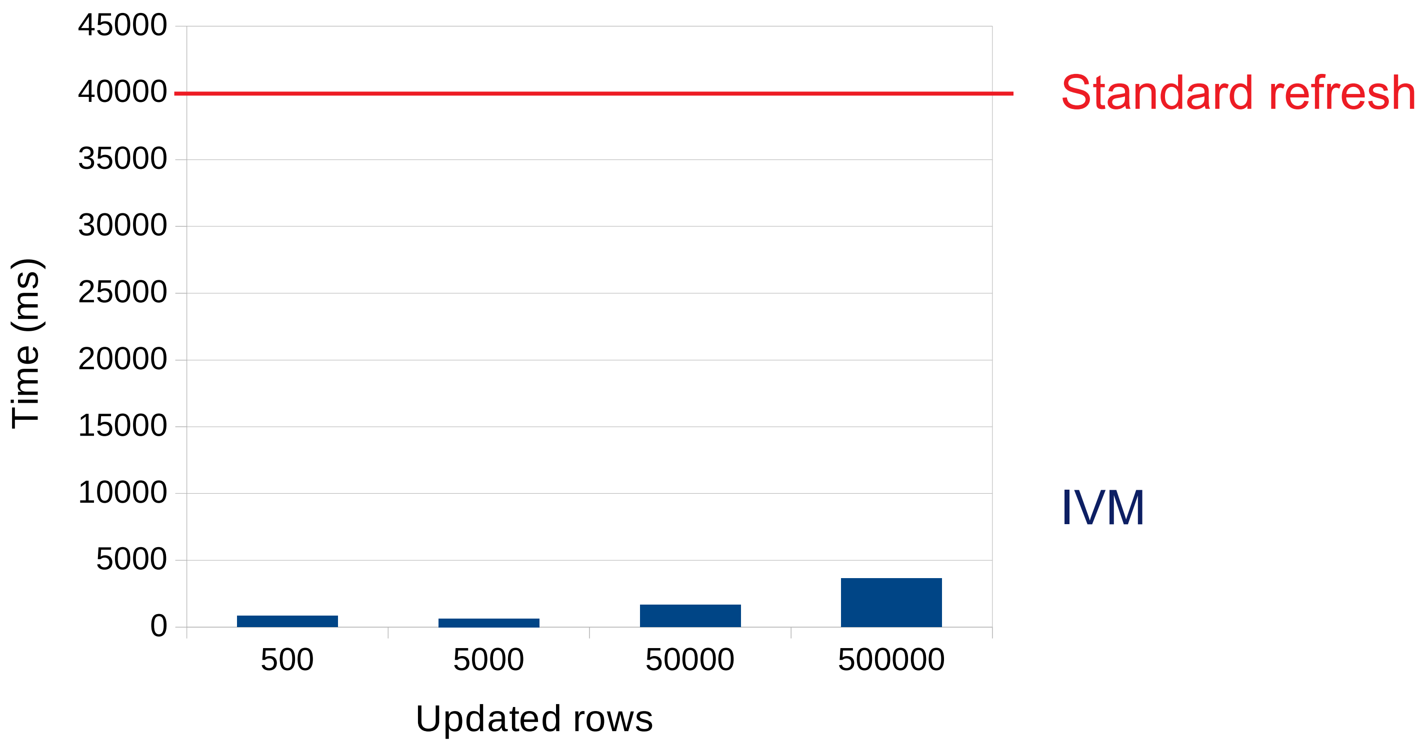
Updating a row in pgbech_accounts

IVM is (x 74) faster

Confirming the two results are same

Examples (3)

- Updating pgbench_accounts:
  - IVM is faster than the standard refresh.
  - The execution time increases as the number of updated rows increases.



Examples (4)

- Updating pgbench_tellers:

```
CREATE MATERIALIZED VIEW mv_ivm WITH OIDS AS
  SELECT a.aid, a.abalance, t.tbalance
    FROM pgbench_accounts a
      JOIN pgbench_tellers t ON a.bid = t.bid
    WHERE t.tid in (1,2,3) ;
```

```
ivm_demo=# UPDATE pgbench_tellerss SET tbalance = tbalance + 1 WHERE tid = 5;
UPDATE 1
Time: 10.007 ms
```

Updating a row in pgbech_tellers which is unrelated to the view.

```
ivm_demo=# REFRESH MATERIALIZED VIEW INCREMENTALLY mv_ivm;
REFRESH MATERIALIZED VIEW
Time: 512.998 ms
```

IVM is fast because of nothing to do

```
ivm_demo=# UPDATE pgbench_tellers SET tbalance = tbalance + 1 WHERE tid = 1;
UPDATE 1
Time: 9.201 ms
```

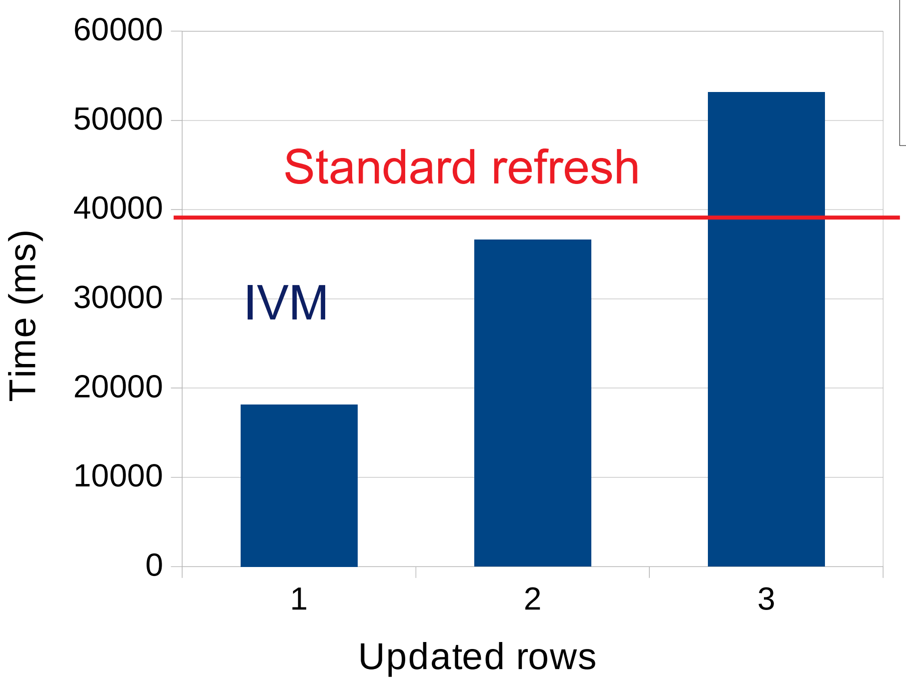
Updating a row in pgbech_tellers which is related to the view.

```
ivm_demo=# REFRESH MATERIALIZED VIEW INCREMENTALLY mv_ivm;
REFRESH MATERIALIZED VIEW
Time: 19555.446 ms (00:19.555)
```

Examples (5)

- Updating pgbench_tellers:
  - IVM is not so better than the standard refresh.
    - A row update in pgbench_tellers is relating to all rows of pgbench_accounts.
    - 3 rows update causes update of ALL rows in the view.
  - Overhead of the oidmap maintenance

```
CREATE MATERIALIZED VIEW mv_ivm WITH OIDS AS
  SELECT a.aid, a.abalance, t.tbalance
    FROM pgbench_accounts a
      JOIN pgbench_tellers t ON a.bid = t.bid
    WHERE t.tid in (1,2,3) ;
```



Current Restrictions

- The current PoC implementation of IVM is very simplified.
- A lot of restrictions
  - Only simple join view is supported.
    - Join of two base tables, selection, and projection
    - Not supporting:
      - Aggregation, multiple joins, sub queries, ...etc.
  - Plans used for creating and refreshing the view is limited.
    - Nested loop join, merge join, sort, seqscan
    - Not supporting:
      - Hash-join, bitmap scan, parallel scan, …etc.

About using OIDs

- OIDs in our implementation:
  - Identify tuples in base relations and materialized views.
  - Provide mapping between these tuples.

- Problems:
  - 32-bit integer: It is not large enough to provide uniqueness in large tables.
  - Using a user-created table's OID column as a primary key is discouraged.
  - It is hard to handle in the implementation …

→ Other better way might need to be investigated.

Summary

- PoC implementation of IVM on PostgreSQL using OIDs
  - Fast refresh of a materialized view
  - It would be efficient when small fraction of a large base table is update.

- Future plans:
  - Eliminate performance issues:
    - Overhead of the oid mapping maintenance.
    - Direct update of the materialized view instead of delete & insert.
  - Support more generally defined view and plans.
  - Avoid to rely on OIDs:
    - Using unique index on base tables?

Thank you

SRA OSS, INC.