Design Challenges for Incremental View Maintenance in PostgreSQL

Yugo Nagata
nagata@sraoss.co.jp

PGConf.dev 2026

Materialized View:

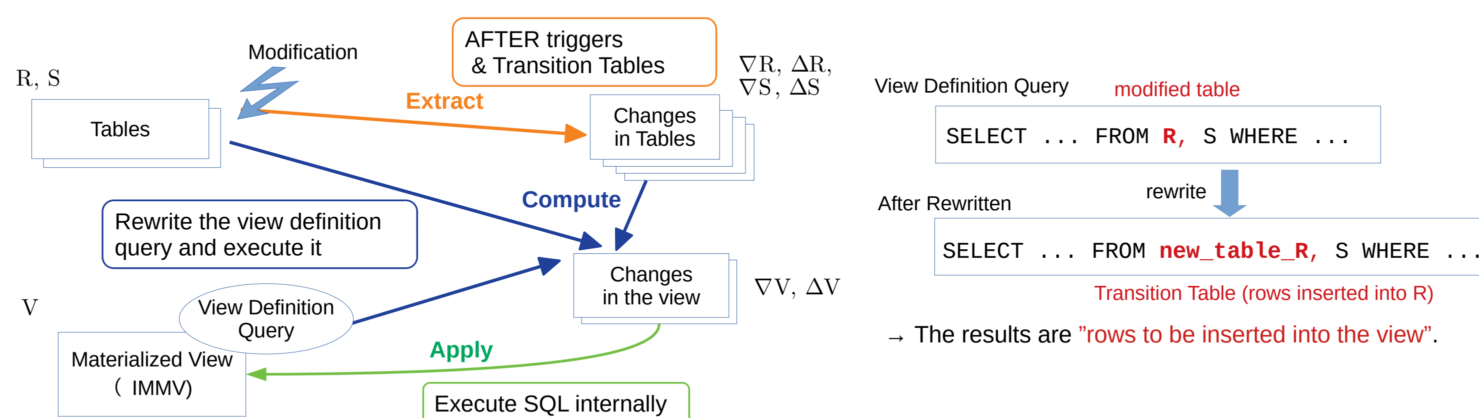
- Stores the result of a view definition query.
- Requires refresh when base tables are modified.
- REFRESH MATERIALIZED VIEW recomputes the entire view.

Incremental View Maintenance (IVM):

- Computes and applies only the incremental changes to the view.

Design of the Current Patch:

- The view is automatically maintained when base tables change.
- AFTER triggers are created on base tables.
- Transition Tables are used to capture changes.
- View definition queries are rewritten and executed to compute incremental changes.



Design Considerations (seeking feedback):

1. Refresh timing
   - Immediate
   - Deferred
2. Maintenance placement
   - Trigger-based
   - In the executor
3. Catalog design
   - Reusing existing catalogs (pg_class, pg_trigger, etc.)
   - Adding new catalog objects
4. Capturing pre-update state (before changes)
   - Snapshot-based
   - Transition-table reconstruction
   - Hybrid
5. Initial feature scope
   - Selection / Projection / Join
   - DISTINCT and duplicate handling
   - Aggregation

Refresh Timing

- Immediate (current patch)
  - View is updated immediately after each base table modification
  - No change tracking is required
  - Impacts update performance
- Deferred
  - View is updated later (asynchronously or on demand)
  - Requires a mechanism to track changes on base tables
  - Better write performance, but added complexity

Maintenance Placement

- Trigger-based (current patch)
  - View–table relationship is managed via triggers (pg_trigger)
  - Transition tables are used to capture changes
  - Reuses existing infrastructure; no major changes to the executor
- In the executor
  - Avoids reliance on triggers (similar to declarative partitioning)
  - May reuse parts of trigger functionality (e.g., transition tables)

Catalog Design

- Reusing existing catalogs (current patch)
  - Add a flag to pg_class to indicate IVM-enabled views (possibly stored in reloptions)
  - View definition stored in pg_rewrite
  - View–table relationships managed via triggers (pg_trigger)
- Adding new catalog objects
  - Manage view–table relationships in a new catalog (e.g., pg_matview_tables)
  - Store materialized view status in a dedicated catalog or stats view (e.g., last refresh time, staleness)

Commitfest item:

Capturing Pre-update State

- Pre-update state is required when multiple base tables are modified in a query

Example:	
View definition	$V \stackrel{\text{def}}{=} R \bowtie S$
Tables modifications	R _new = R_old ∪ ΔR, S_new = S_old ∪ ΔS
Incremental change	ΔV = (ΔR ⋈ S_old) ∪ (R_new ⋈ ΔS)

- Snapshot-based
  - Uses a snapshot to obtain the pre-update state
  - Requires infrastructure to apply a snapshot selectively to a table (similar to Oracle's AS OF)
- Transition-table reconstruction
  - Reconstructs the pre-update state from OLD/NEW tables
  - Requires set operations (EXCEPT ALL / UNION ALL)
  - Performance considerations; also needed for deferred maintenance
- Hybrid (current patch)
  - Filters inserts via snapshot and restores deletes from OLD

```
SELECT… FROM tbl
 WHERE ivm_visible_in_prestate(t.tableoid, t.ctid, matview_oid)
UNION ALL SELECT … FROM deleted_tuples_from_tbl;
```

- ivm_visible_in_prestate:
  Returns true if a row is visible in the pre-update snapshot (captured before modification, e.g., in a BEFORE trigger)

Initial Feature Scope

- The current patch supports:
  - Selection / Projection / Join
  - DISTINCT and duplicate handling
  - Aggregation
- Design considerations:
  - Should the initial scope be simplified?
    (e.g., start with SPJ and add features incrementally)
  - Additional complexity:
    - Duplicate handling (tuple counting)
    - Aggregation (complex delta computation per aggregate)