PGC POSTGRESCONF CN 2021 | PGConf.Asia 12.14-17

PostgreSQL Internals and Incremental View Maintenance Implementation

Yugo Nagata (SRA OSS, Inc. Japan)

https://2021.postgresconf.cn

About Me

- Yugo Nagata
  - Software Engineer at SRA OSS, Inc. Japan
  - Research and Development on PostgreSQL
    - Incremental View Maintenance (IVM)
  - Lecture on PostgreSQL Internal

This Talk

- PostgreSQL internals related to Incremental View Maintenance (IVM) implementation
  - Query and utility command processing related to materialized Views
  - Some components used in IVM implementation

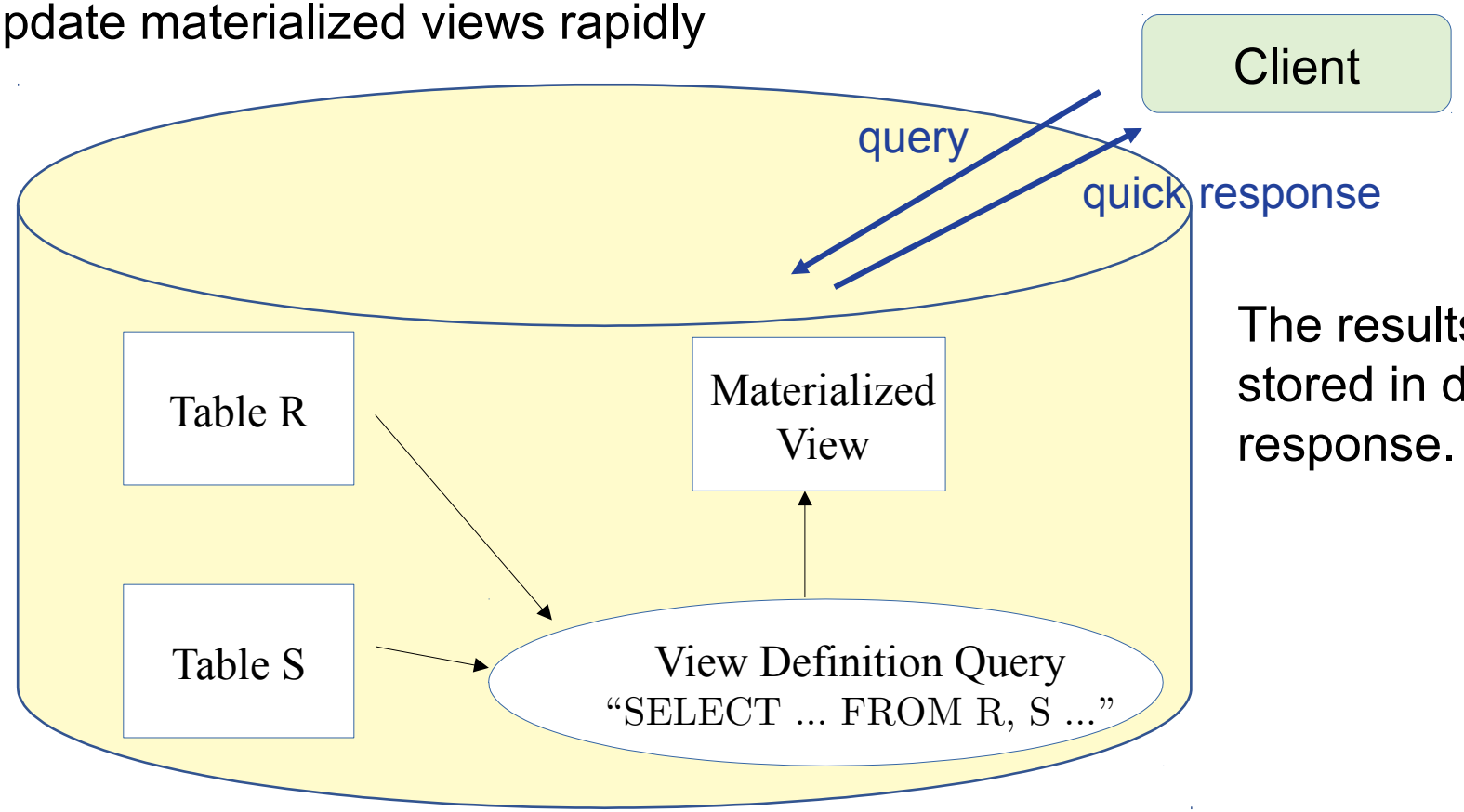
- For those who are interested in PostgreSQL internals

CONTENT

What is Incremental View Maintenance?

What is Incremental View Maintenance? (1)

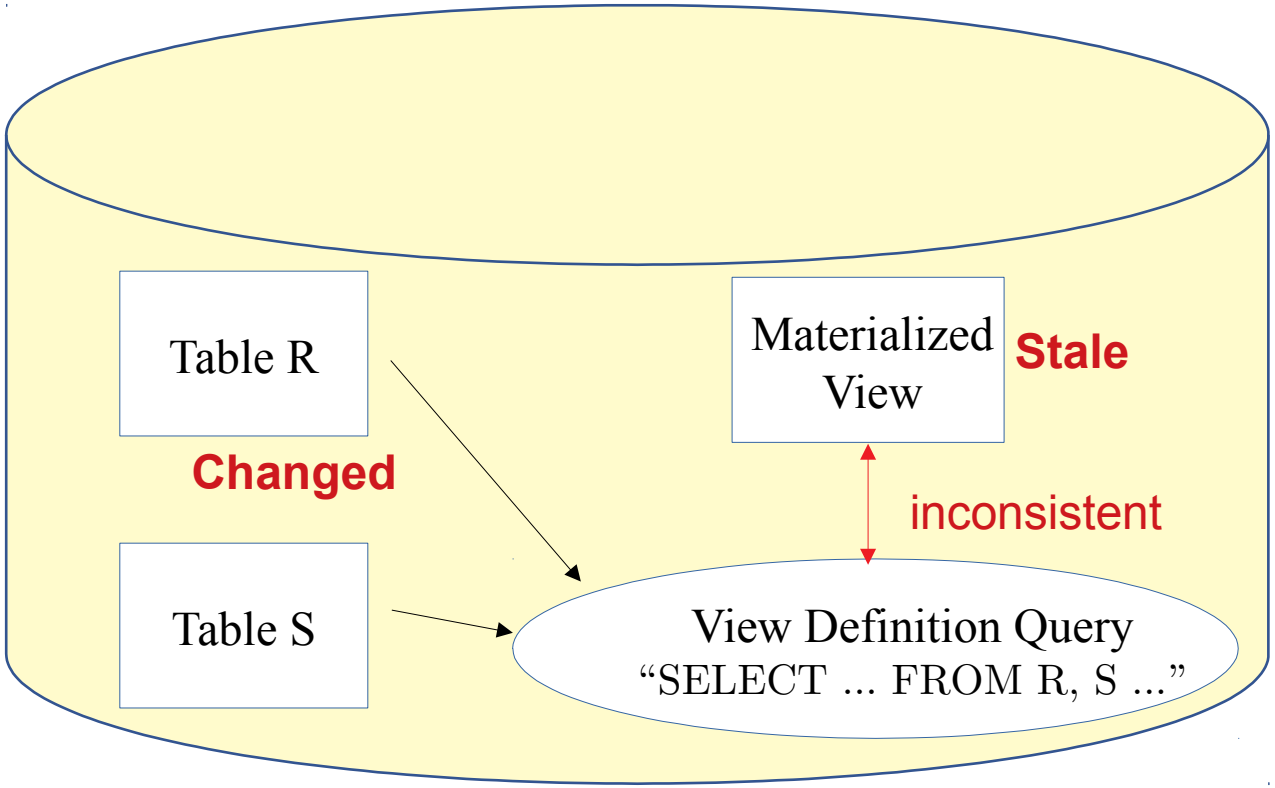
A way to update materialized views rapidly



The results of the query are stored in database for quick response.

What is Incremental View Maintenance? (2)

A way to update materialized views rapidly

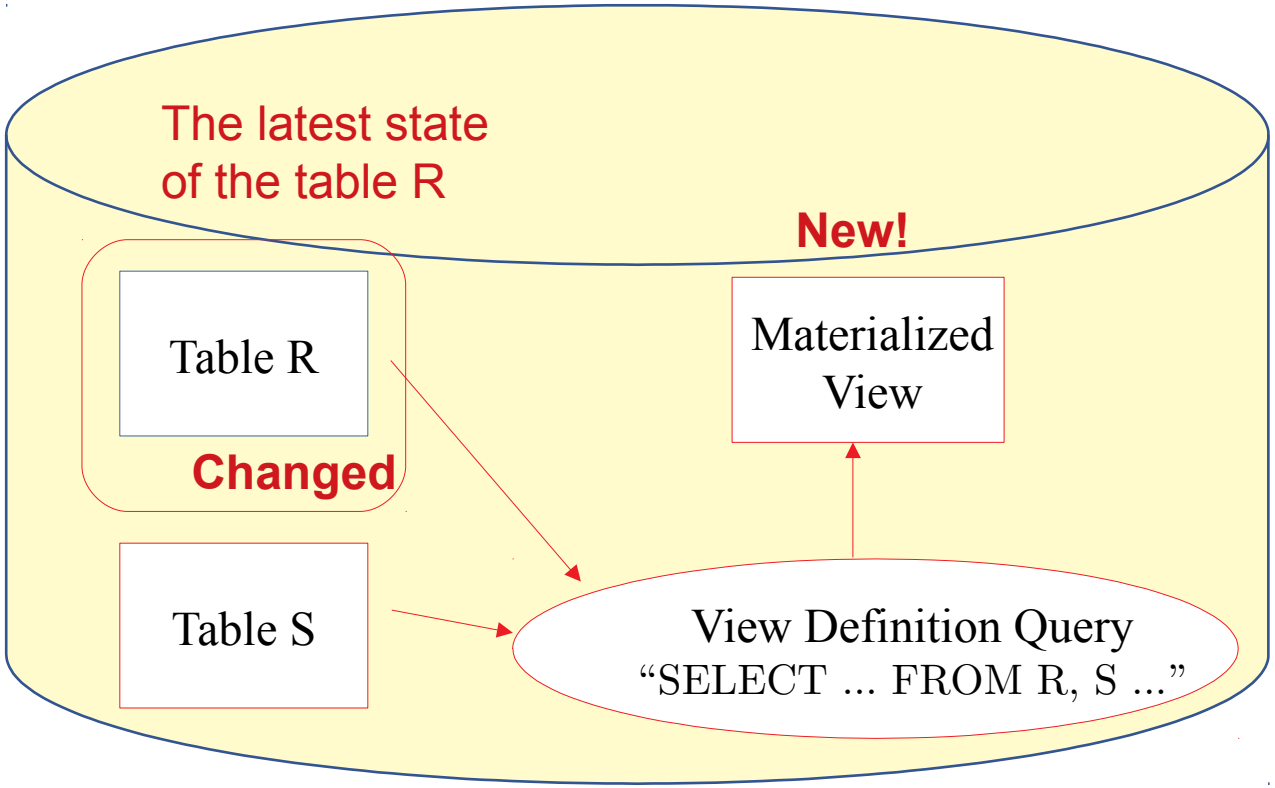


The results of the query are stored in database for quick response.

Needs to be maintained after a base table is modified

What is Incremental View Maintenance? (3)

A way to update materialized views rapidly



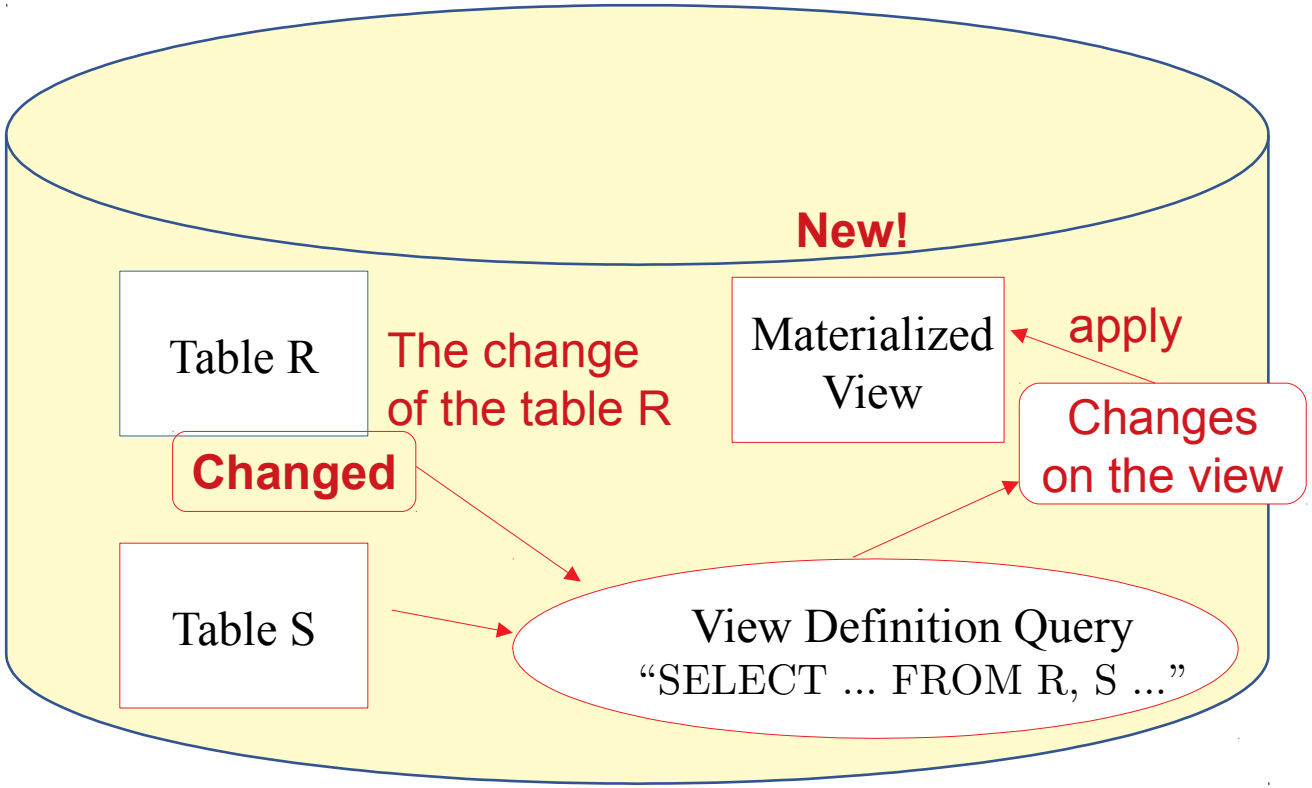
REFRESH MATERIALIZED VIEW

II

1. Re-compute the contents using the latest state of tables.

What is Incremental View Maintenance? (4)

A way to update materialized views rapidly



REFRESH MATERIALIZED VIEW

II

1. Re-compute the contents using the latest state of tables.

2. Compute and apply only the incremental change.

II

Incremental View Maintenance

Incremental View Maintenance (IVM) on PostgreSQL

- Not supported on PostgreSQL yet.
- We have proposed to implement IVM on PostgreSQL.
  - Materialized views can be updated automatically and incrementally when a base table is updated.

- Details will be explained in the last part.

PostgreSQL Query Processing

PostgreSQL Internals and Materialized View

- Materialized Views needs to process SELECT query.
  - A view is created from the results of query processing.

```
CREATE MATERIALIZED VIEW mv AS
    SELECT x, y FROM R, S WHERE R.i = S.i;
```

  - How a query is processed in PostgreSQL ?

Query Processing (1)

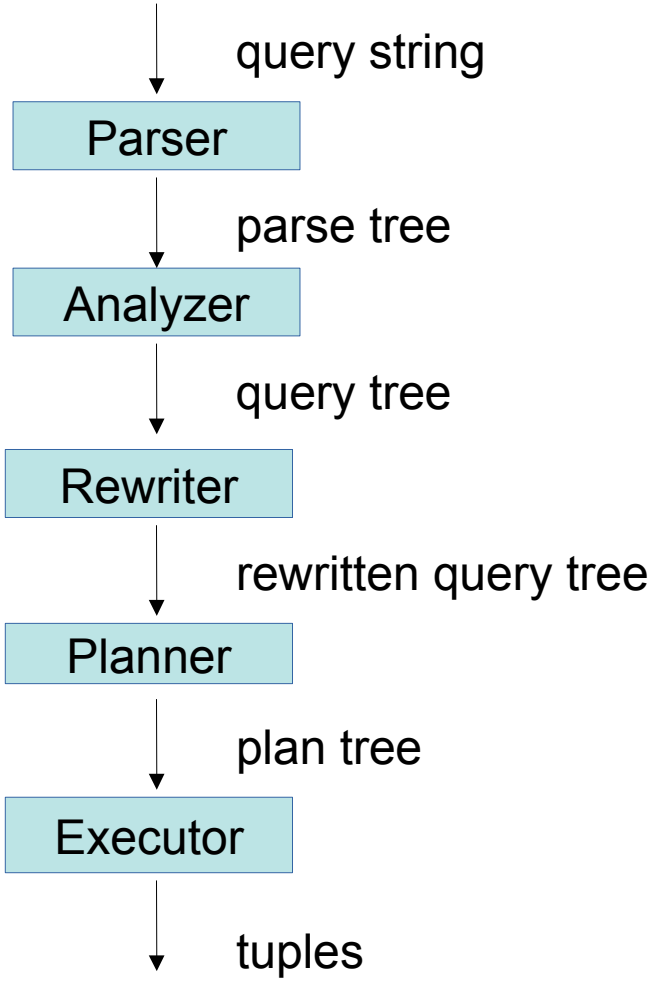
- Parser
  - Check the syntax of the query
  - Build a parse tree

src/backend/parser/gram.y

```
SELECT opt_all_clause opt_target_list
 into_clause from_clause where_clause
 group_clause having_clause window_clause
     {
         SelectStmt *n = makeNode(SelectStmt);
         n->targetList = $3;
         n->intoClause = $4;
         n->fromClause = $5;
         n->whereClause = $6;
         n->groupClause = ($7)->list;
         n->groupDistinct = ($7)->distinct;
         n->havingClause = $8;
         n->windowClause = $9;
         $$ = (Node *)n;
     }
```

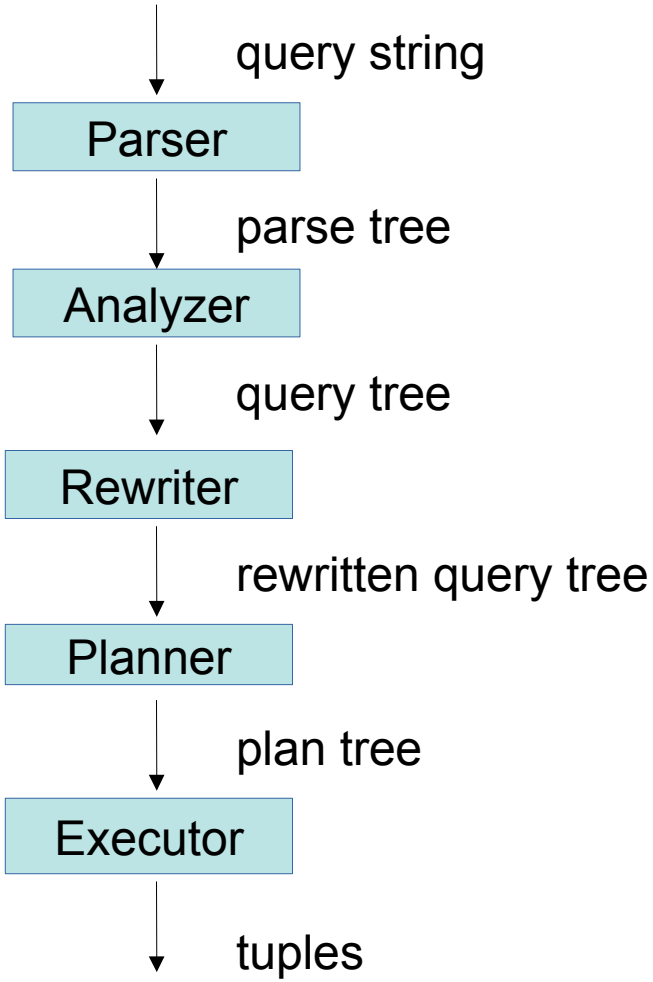
syntax rule

Building a query tree node



Query Processing (2)

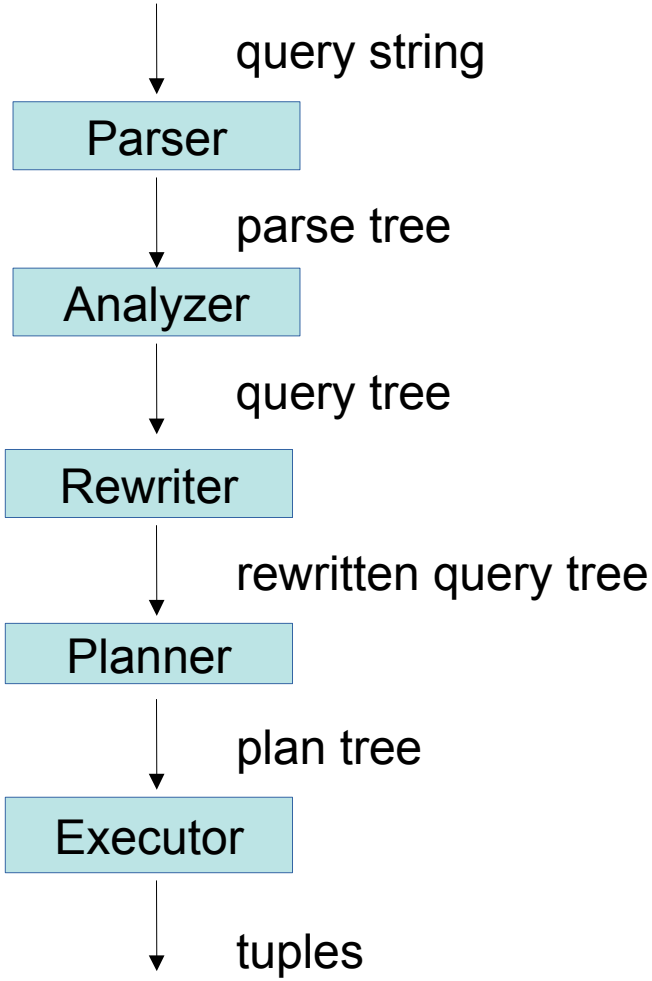
- Analyzer
  - Perform semantic analysis
    - system catalog look-up
  - Examples:
    - table name → table OID
    - Expand “ * “ to column names
    - Identify OIDs of types, operators, etc.
  - Transform a parse tree to a query tree



Query Processing (3)

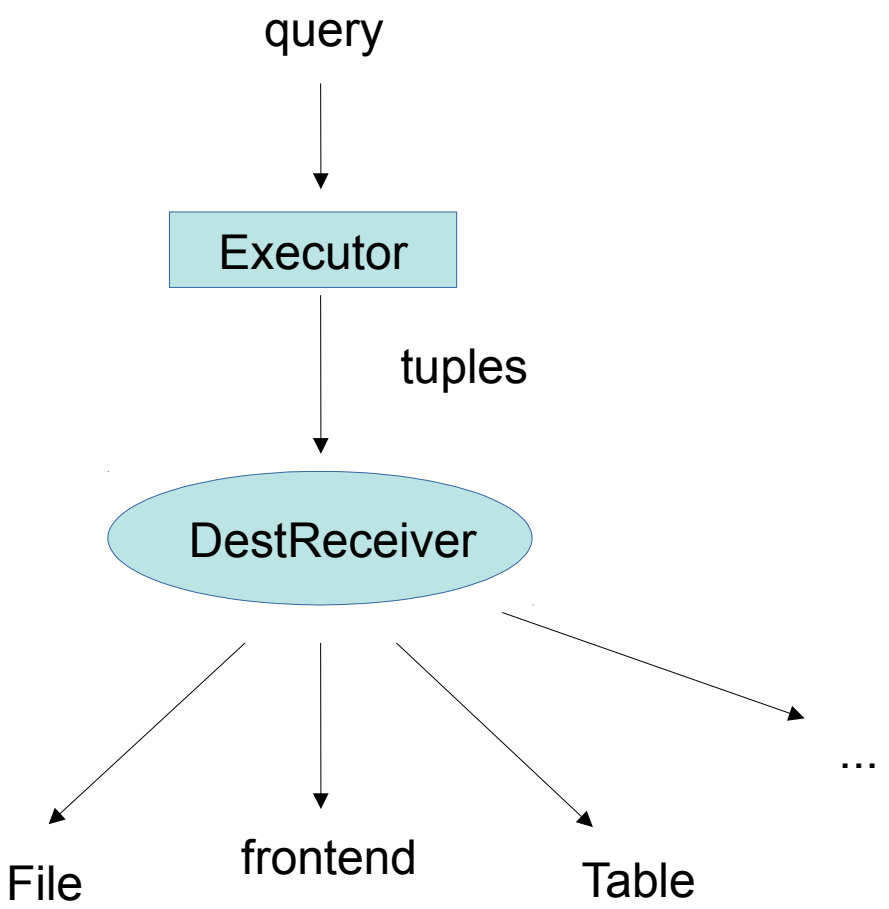
- Rewriter
  - Apply RULE to query tree
  - VIEW support
- Planner
  - Create an optimal execution plan
  - Build a plan tree
- Executor
  - Retrieve tuples in the way given by the plan tree

  → Where will the tuples go?



DestReceiver

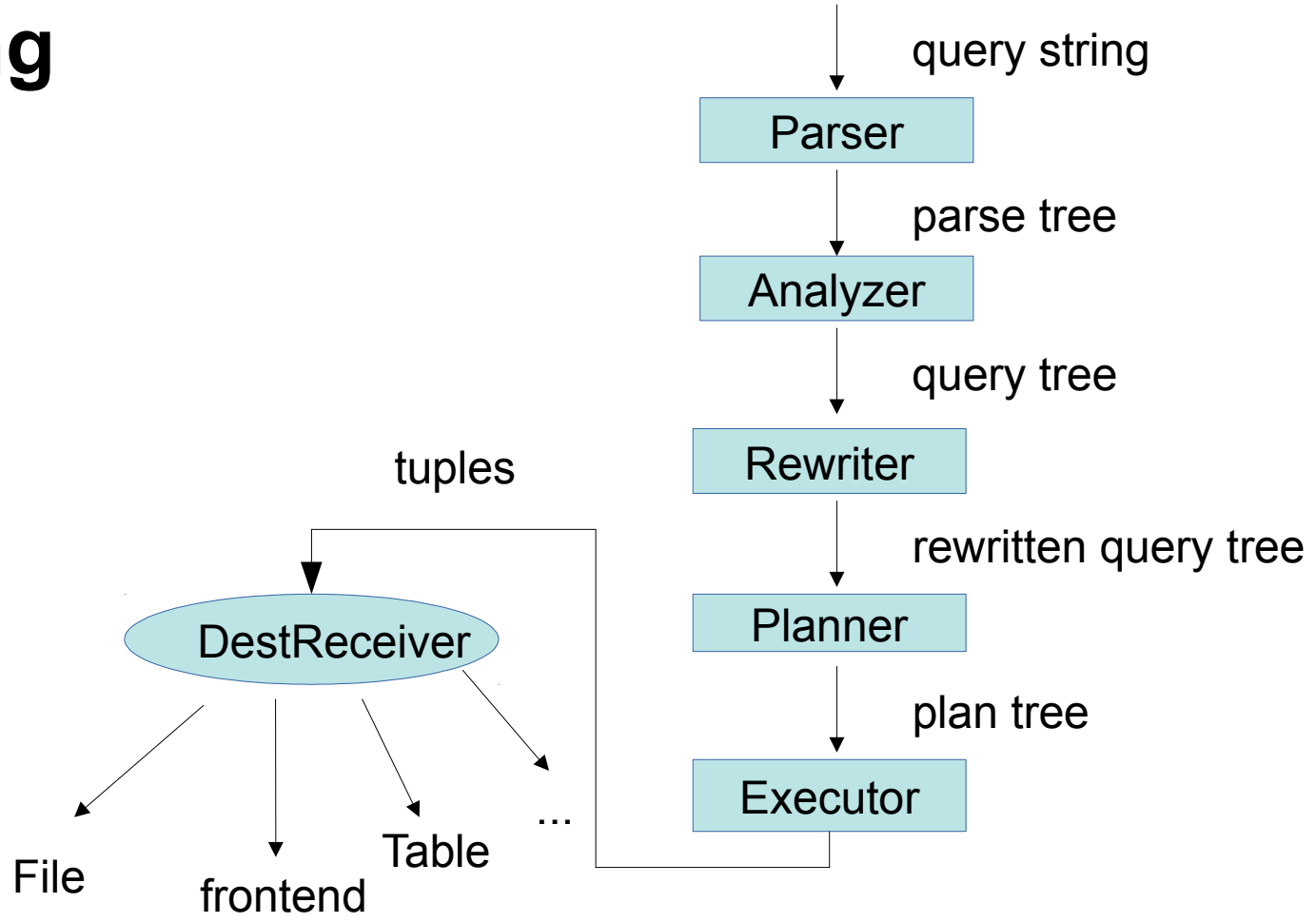
- Object to manage tuple destination
- In most queries, DestRemote is used.
  - Results are sent to frontend process.
- Other cases:
  - Sent to a file (COPY TO)
  - Sent to a new table (SELECT INTO, CREATE AS)
  - etc.



Recap: Query Processing

- Parser
- Analyzer
- Rewriter
- Planner
- Executor

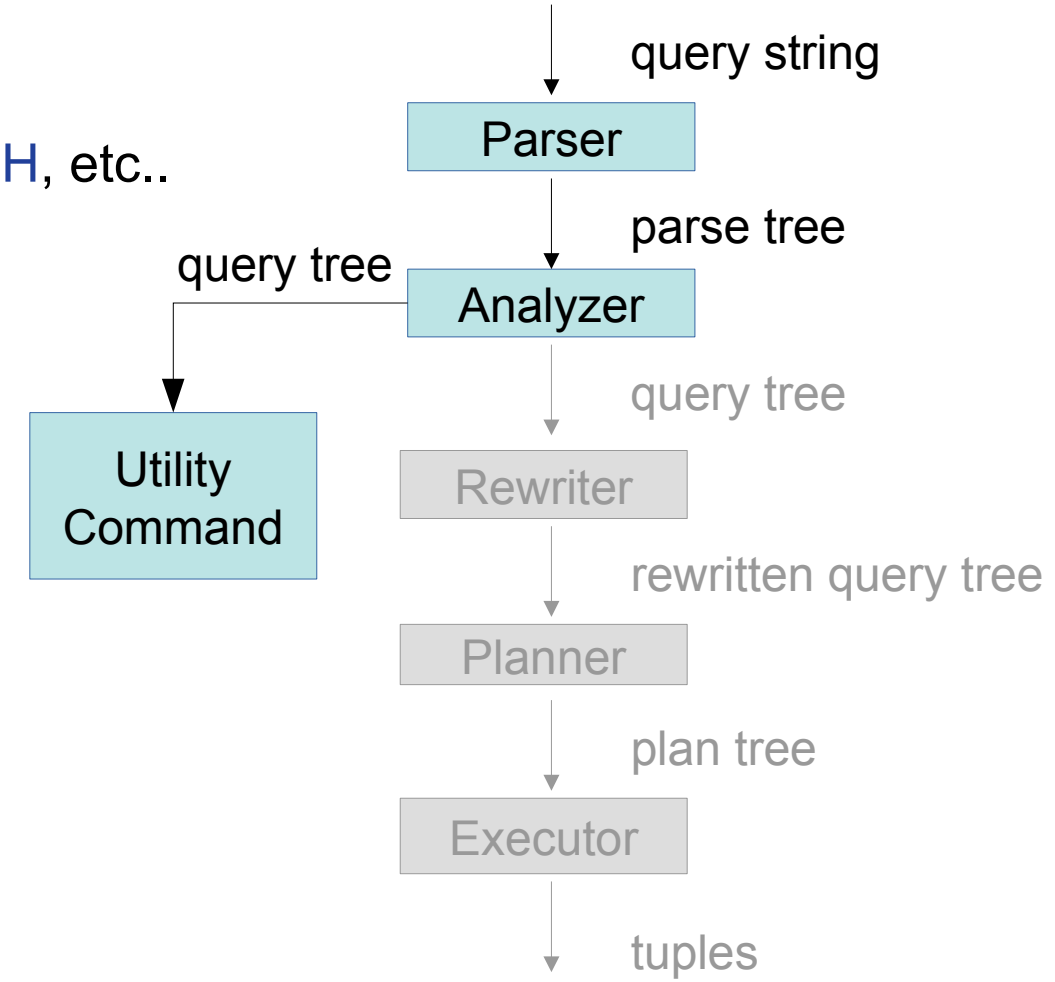
- The destination of tuples are managed by DestReceiver



Utility Command and Materialized View

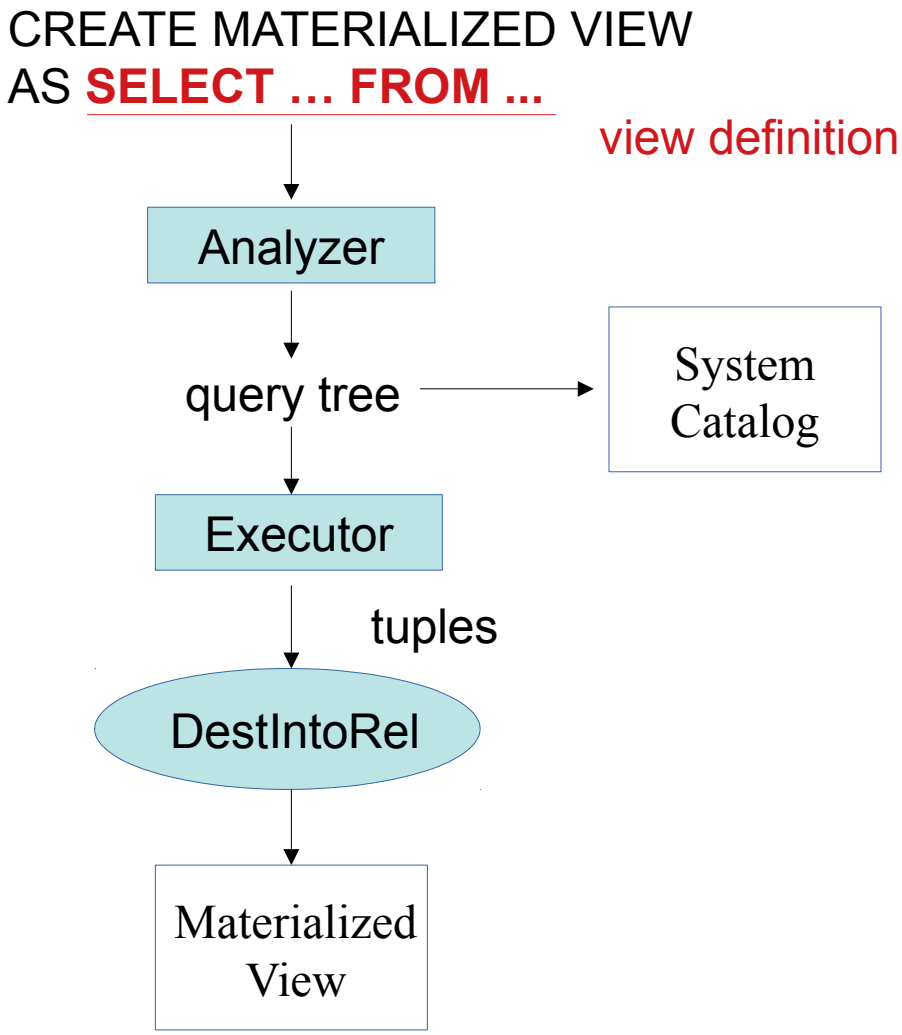
Utility Command

- CREATE, DROP, ALTER, COPY, VACUUM, REFRESH, etc..
- Commands have to be parsed first.
- Some commands including SELECT need Analyzer.
  - SELECT INTO
  - CREATE TABLE AS
  - CREATE MATERIALIZED VIEW
  - etc...
- Rewriter, Planner, and Executor are not required.



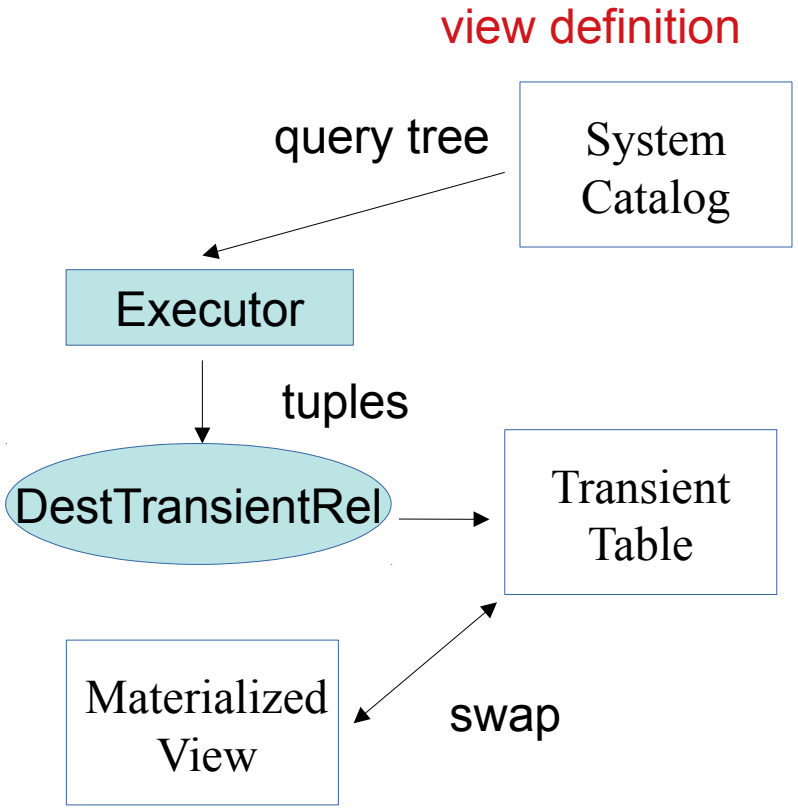
CREATE MATERIALIZED VIEW

- The behavior is similar to CREATE AS / SELECT INTO
  - A relation is created from query results.
  - Implemented in the same function.
    (ExecCreateTableAs in backend/commands/createas.c)
- The view definition query are executed.
  - Rewriter, Planner, and Executor are called internally.
  - The results of the query are stored into a new table
  - (= a materialized view).
- Also, the query is stored in system catalog as a query tree.



REFRESH MATERIALIZED VIEW

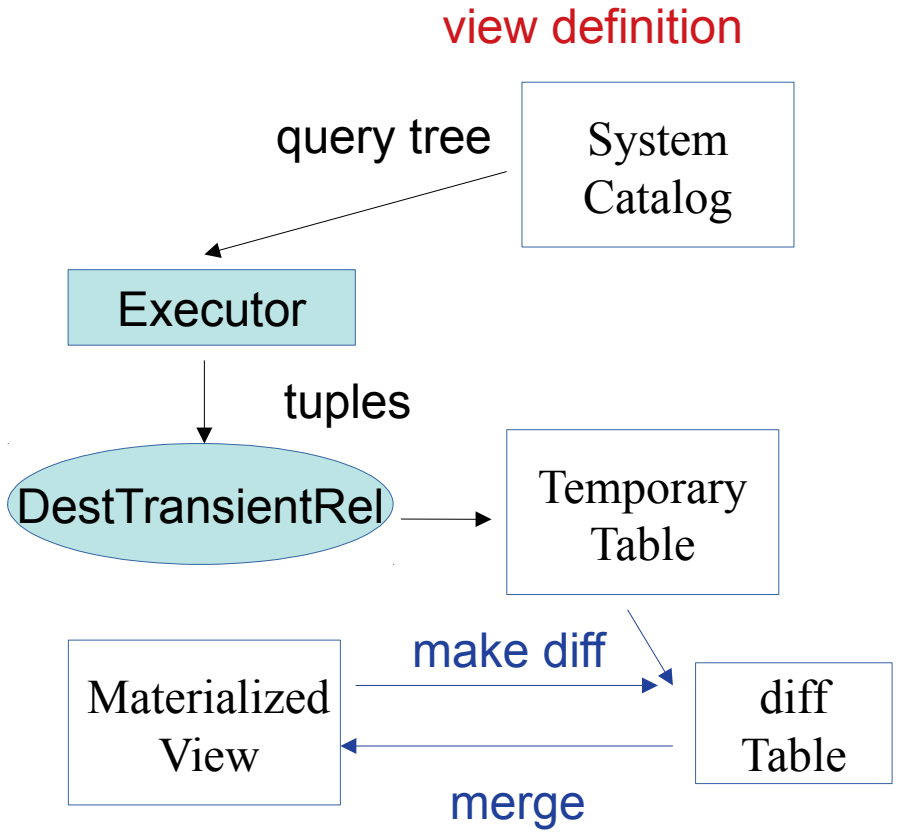
- Update a materialized view by re-execute the view definition query
- Basically, same as CREATE MATERIALIZED VIEW
  - The results of the view definition query are stored into a transient table.
  - The transient table and the materialized view are swapped.
  - Require a strong lock.



REFRESH MATERIALIZED VIEW CONCURRENTLY

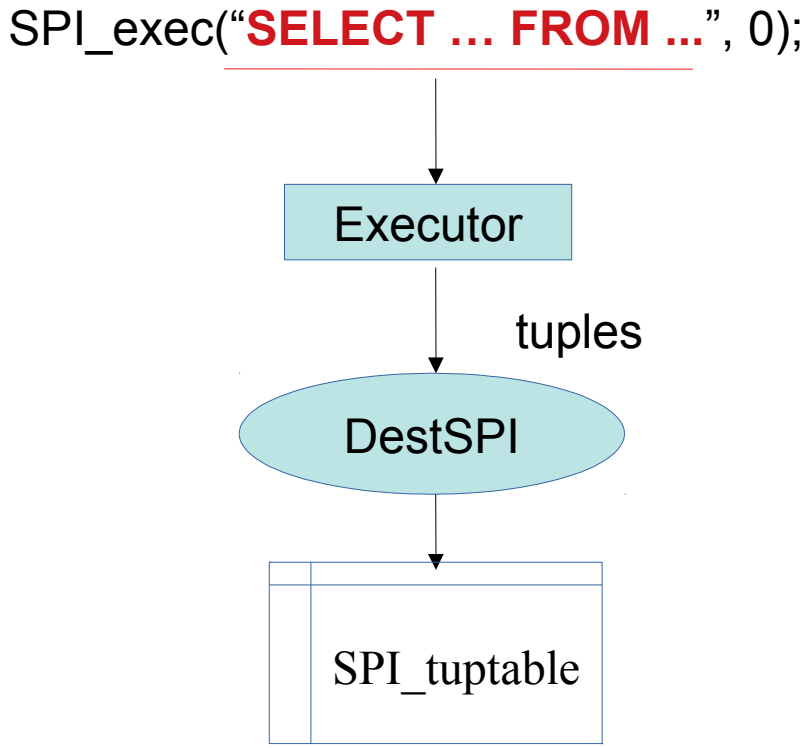
- Update a materialized view with a weaker lock
- The results of the view definition query are stored into a temporary table.
- Create the diff table by comparing the results and the materialized view (using LEFT JOIN).
- Finally, merge it to the materialized view (using DELETE and INSERT).

SQL queries are executed via SPI.



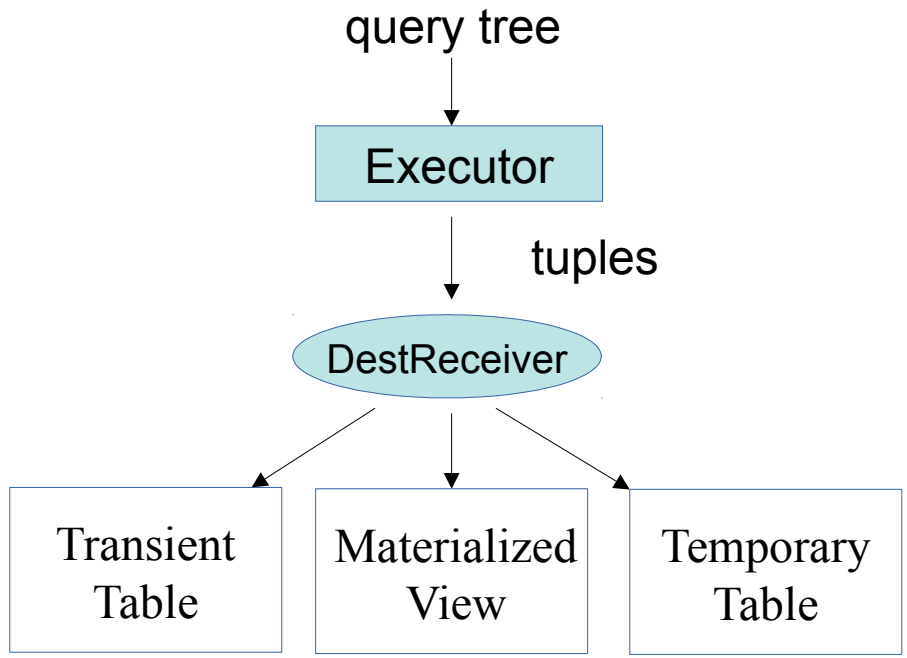
SPI: Server Programming Interface

- A set of interface functions for running SQL commands in C functions.
- Example:
  - `SPI_exec("INSERT INTO tbl ...", 0);`
  - `SPI_exec("SELECT ... FROM tbl WHERE ...", 0);`
- The results are stored in memory.
  - Can be accessed via a global variable SPI_tuptable.



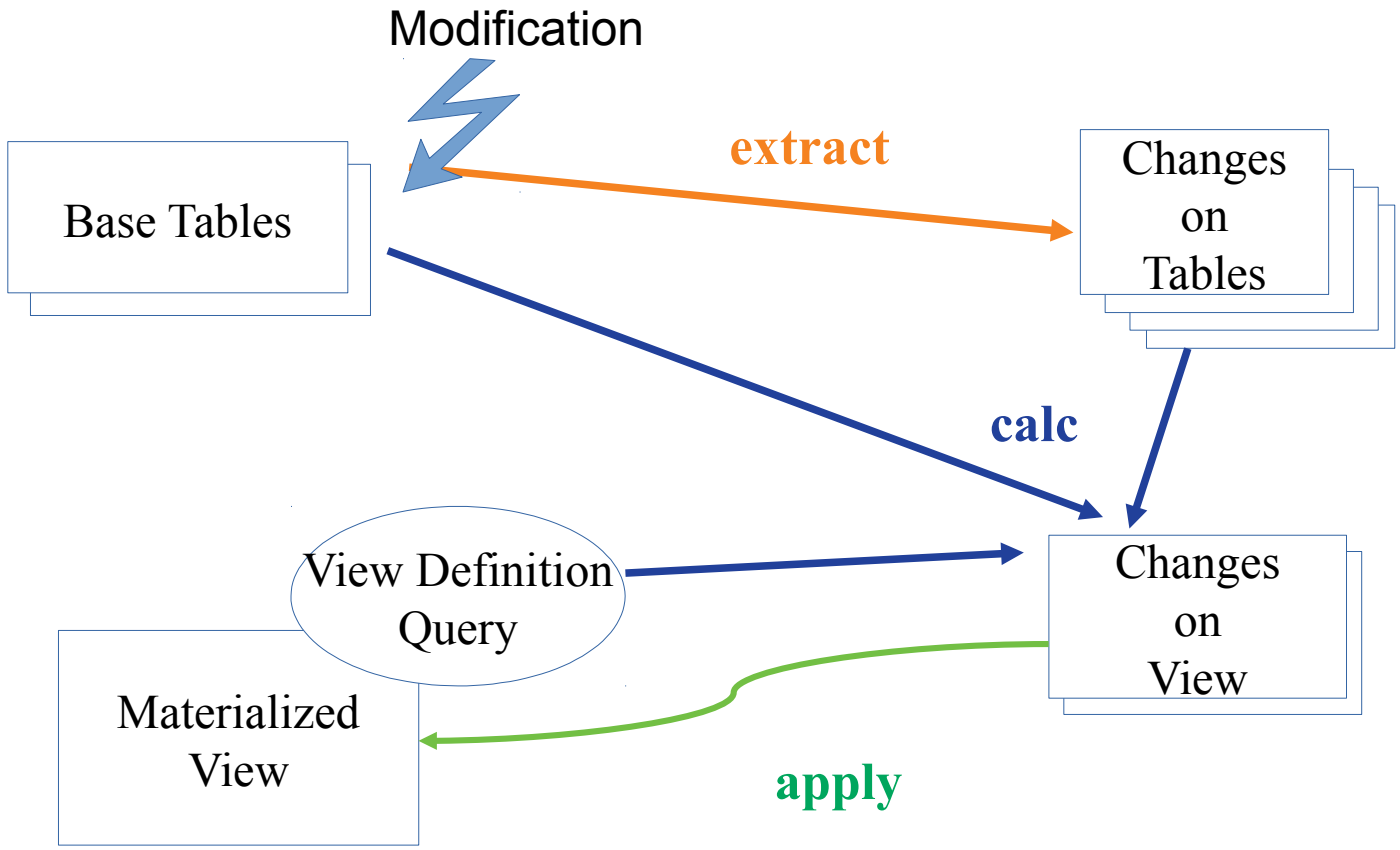
Recap: Utility Command of Materialized View

- The view definition query are executed in commands.
- The results are stored into various types of destinations depending on the situation.
  - Controlled by DestReceiver.
- SQL queries are executed by using SPI.
  (in REFRESH MATERIALIZED VIEW CONCURRENTLY command)

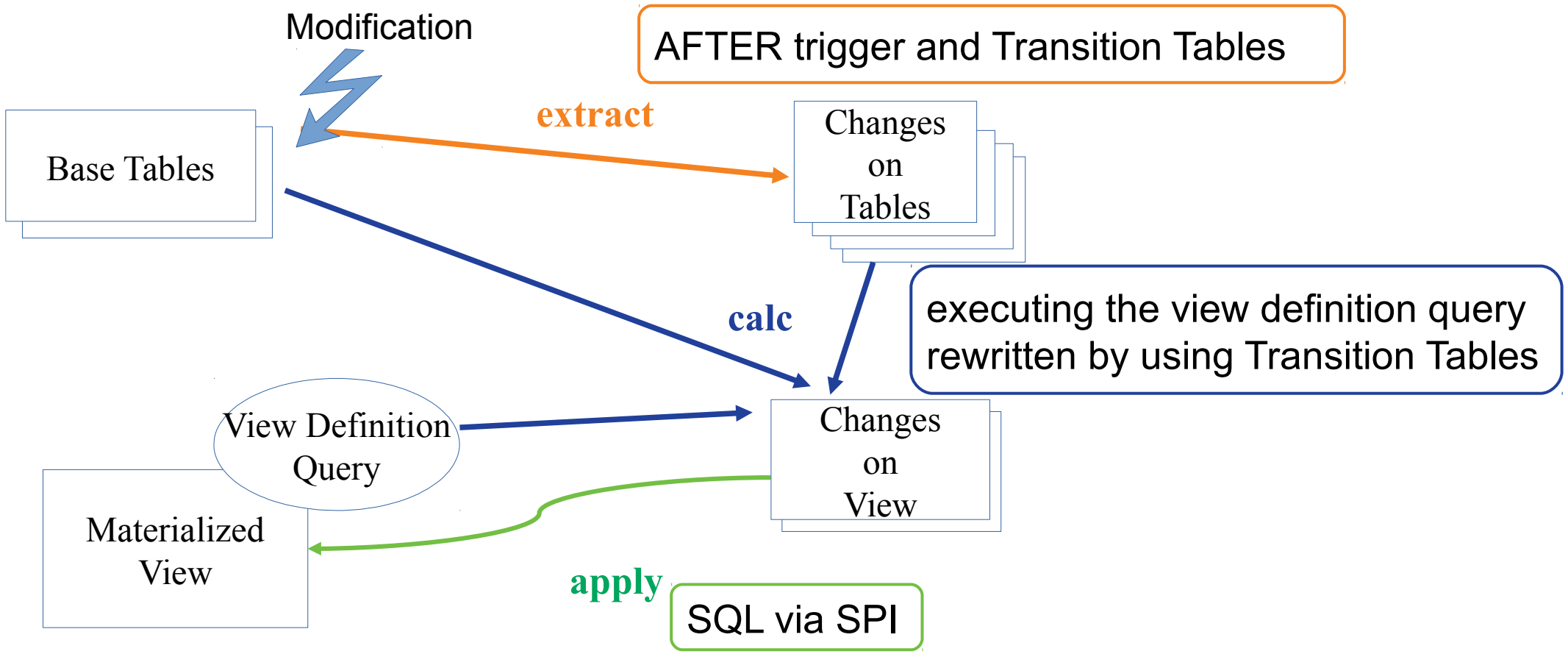


PostgreSQL Internal and Incremental View Maintenance Implementation

Incremental View Maintenance Implementation



Incremental View Maintenance Implementation



Creating Incrementally Maintainable Materialized Views (1)

Syntax:

```
CREATE INCREMENTAL MATERIALIZED VIEW mv AS
    SELECT x, y FROM R, S WHERE R.i = S.i;
```

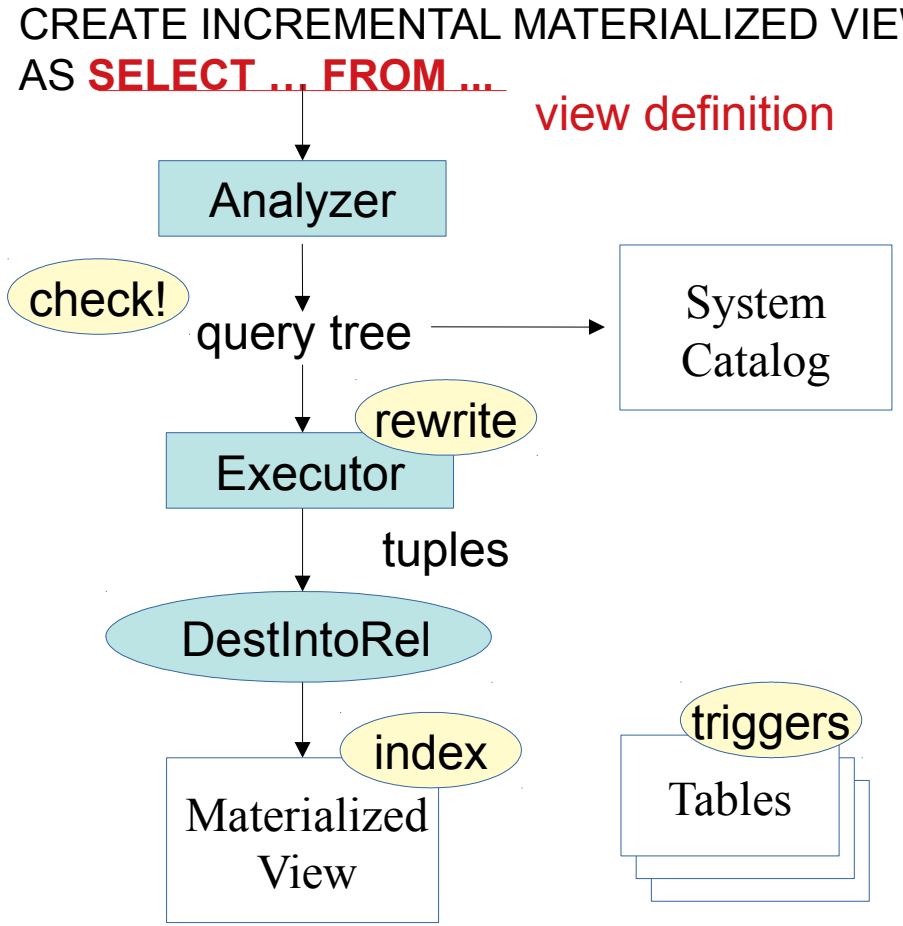
Parser needs changes:

src/backend/parser/gram.y

```
CreateMatViewStmt:
        CREATE OptNoLog incremental MATERIALIZED VIEW create_mv_target AS SelectStmt opt_with_data
                {
                    CreateTableAsStmt *ctas = makeNode(CreateTableAsStmt);
                    ctas->query = $8;
                    ctas->into = $6;
                    ctas->objtype = OBJECT_MATVIEW;
                    ...
                    $6->ivm = $3;
                    $$ = (Node *) ctas;
                }
...
incremental:    INCREMENTAL             { $$ = true; }
                | /*EMPTY*/             { $$ = false; }
```

Creating Incrementally Maintainable Materialized Views (2)

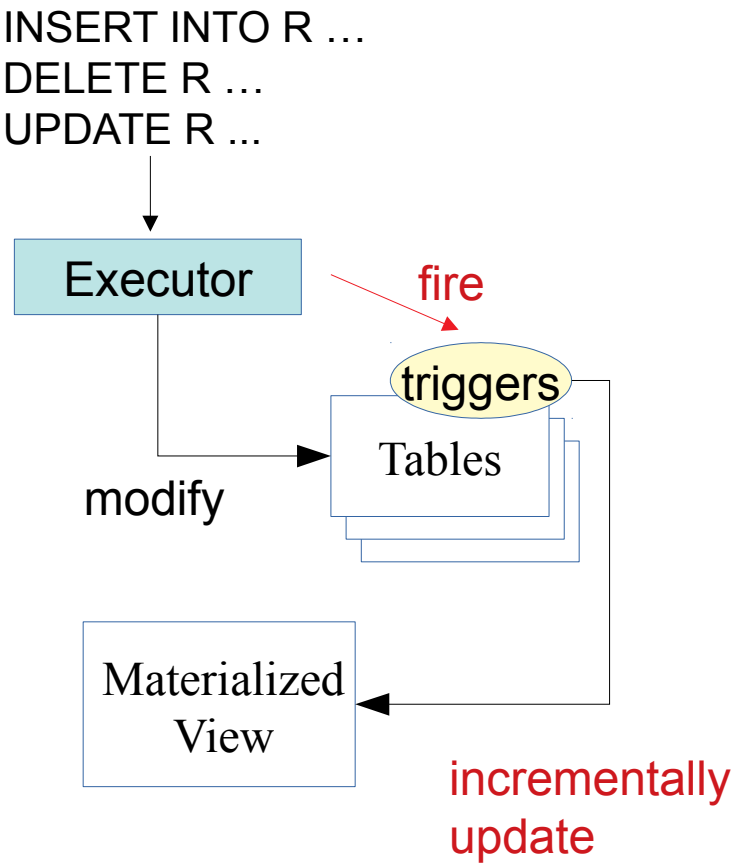
- Changes in ExecCreateTableAs() (backend/commands/createas.c)
  - Check the view definition to forbid unsupported query.
  - The query is rewritten before execution.
    - For aggregates and DISTINCT support
  - A unique index is created on the view if possible.
  - AFTER triggers are created on all base tables.



AFTER trigger

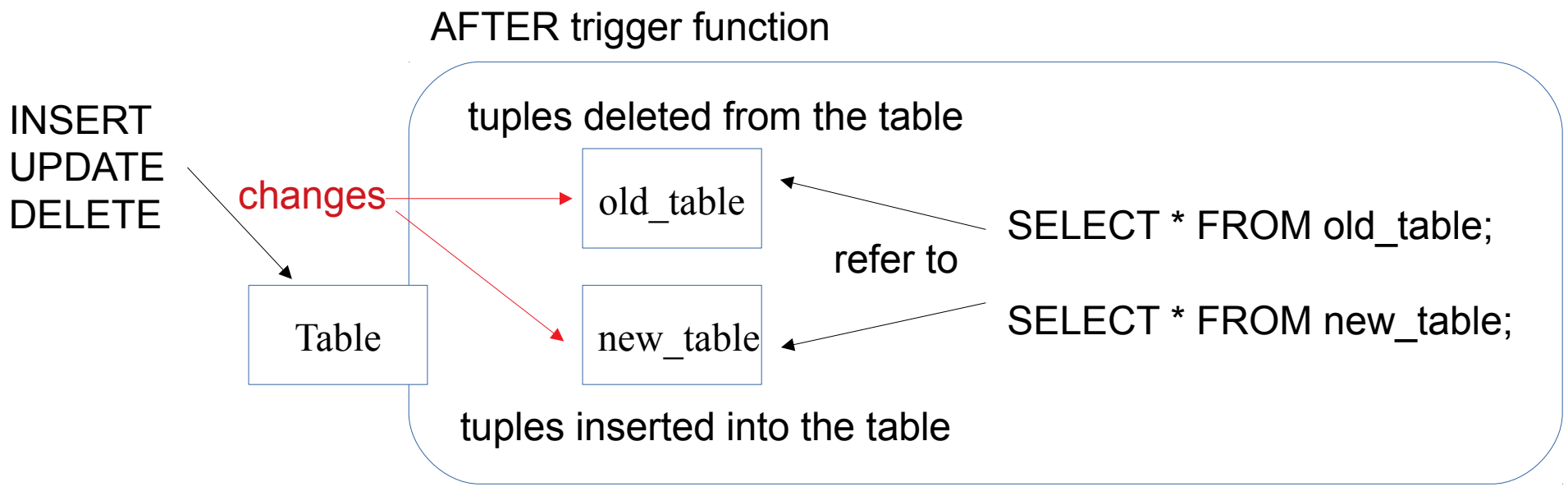
- Fired just after a base table is modified.
- In the IVM AFTER trigger function:
  - Changes on the table are extracted.
  - Changes on the view are calculated.
  - The calculated changes are applied to the view.

The changes on the table is extracted using ***Transition Table***.



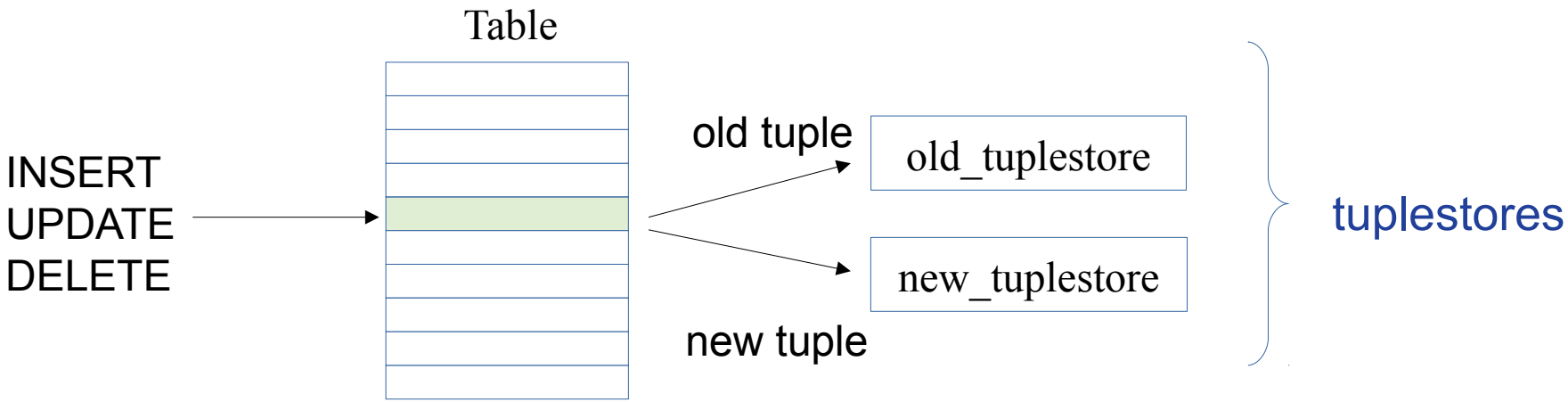
Transition Tables (1): What is Transition Table?

- Transition Tables:
  - A feature of AFTER trigger.
  - Changes on tables can be referred to in the trigger function like normal tables **by specified names.**



Transition Tables (2): How are the changes collected?

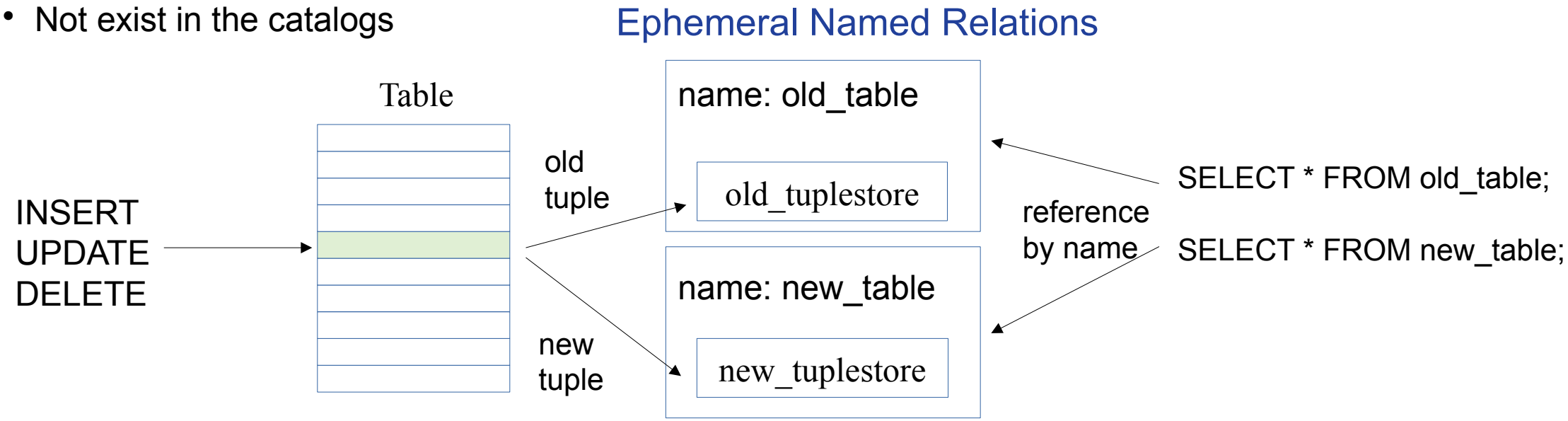
- During modifying a table, each old and/or new tuple is put into *tuplestores*.



- Tuplestore
  - Temporary storage of tuples (in memory or disk)
  - Used also for holdable cursor, CTE, set-returning functions, etc.

Transition Tables (3): How to access to Transition Tables?

- A transition table is accessed as an *Ephemeral Named Relation (ENR)*
  - Tuples are stored in a tuplestore instead of database.
  - Referenced by the names instead of OID
    - Not exist in the catalogs



Calculating Changes on View

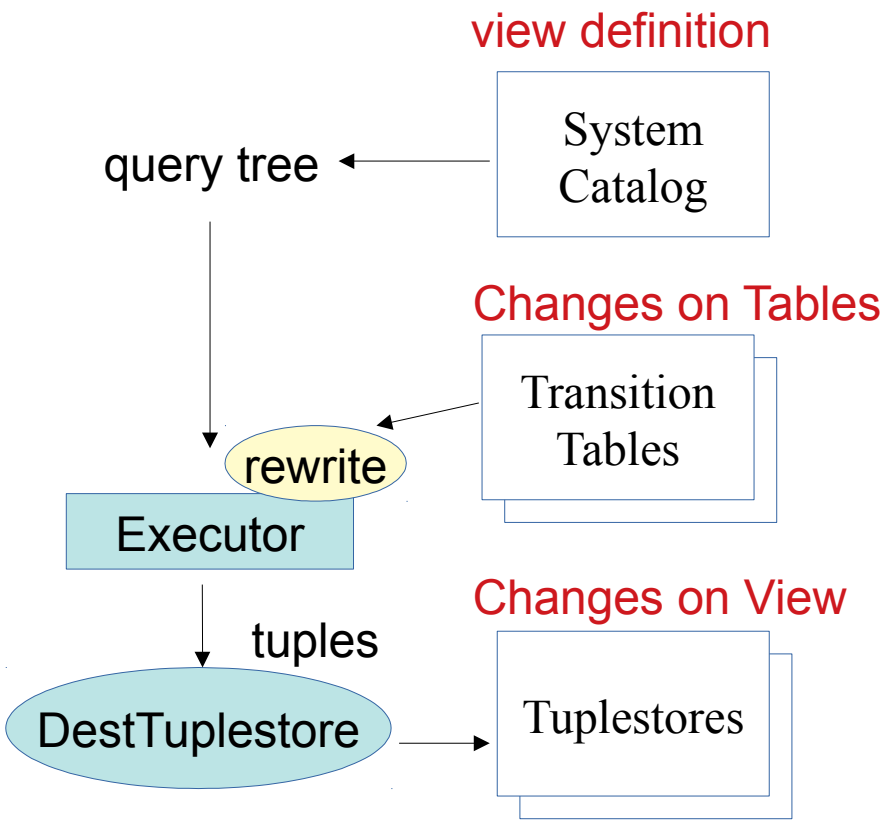
- Execute the view definition query with some rewrite:
  - Replacing the modified table with the transition table.

modified table

```
SELECT x, y FROM R, S WHERE ...
```

```
SELECT x, y FROM new_table_R, S WHERE ...
```

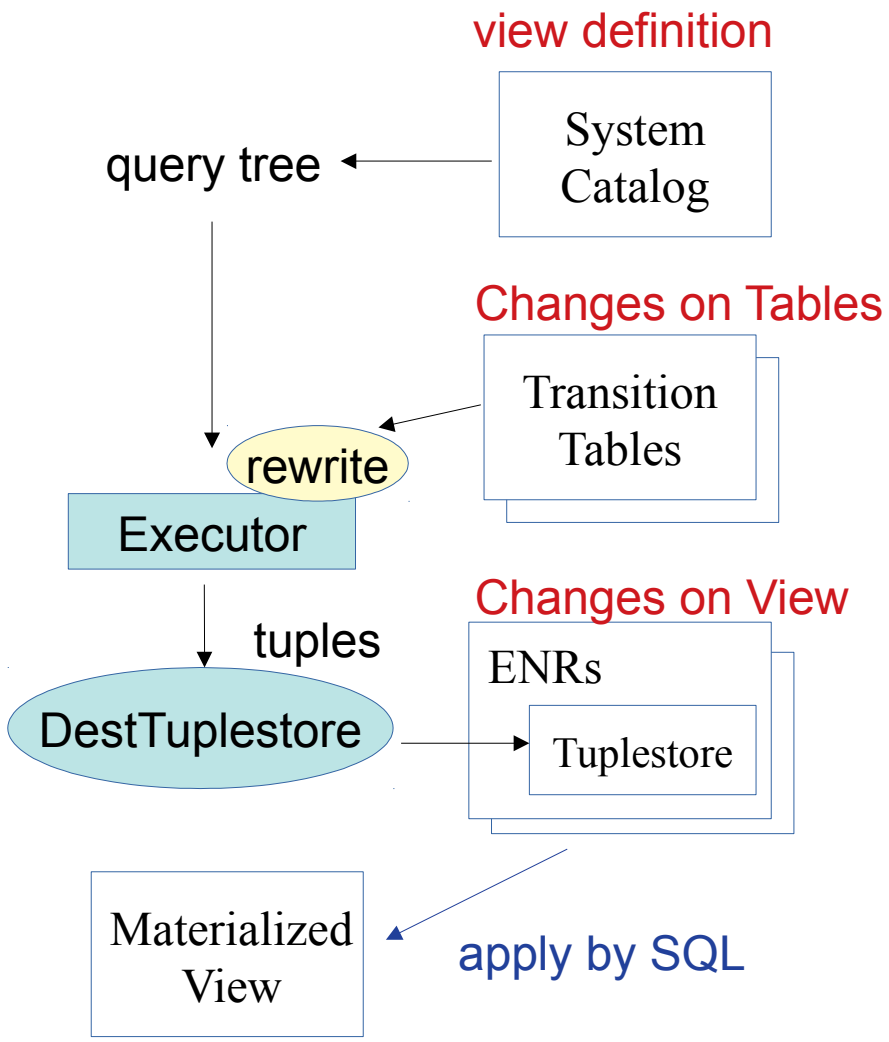
transition table



  - The results are stored into a tuplestores.

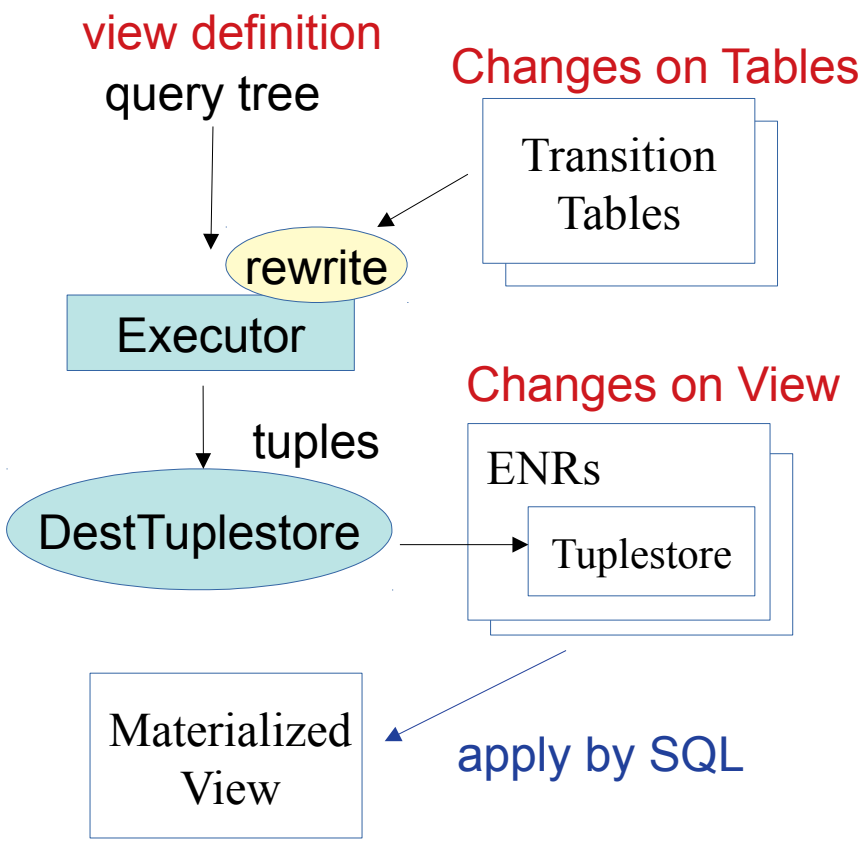
Applying the Change to View

- Create Ephemeral Named Relations (ENRs) from tuplestores that contains changes on the view.
- Apply these tuples in ENRs into the view.
  - Executing SQL by using SPI.
    - Use DELETE to remove tuples from the view.
    - Use INSERT to insert tuples into the view.
    - Use UPDATE to update aggregated values in the view.



Recap: Incremental View Maintenance Implementation

- CREATE INCREMENTAL MATERIALIZED VIEW
  - The parser and the function for this command are changed.
- Transition Tables is used to extract the table changes
  - Tuples are stored in tuplestore and accessed as an Ephemeral Named Relation (ENR).
- Tuplestore and ENR are also used for calculating and applying the view changes.
- The view changes are applied by using SQL via SPI.



Summary

- Overview of query processing and utility commands
  - The destination of results tuples are managed by DestReceiver
  - SPI can be used to run SQL queries.
- Materialized View uses these infrastructure.

- Incremental View Maintenance Implementation uses also:
  - AFTER trigger and transition tables
  - tuplestore and Ephemeral Named Relation

THANK YOU

Yugo Nagata

(nagata @ sraoss.co.jp)

SRA OSS, INC.

CHINA POSTGRESQL ASSOCIATION | PGCC POSTGRESCONF CN 2021 | PGConf.Asia 12.14-17