HypoPG2: Hypothetical Partitioning support for PostgreSQL

PgConf.Russia 2019

Feb. 6th

Julien Rouhaud

Who worked on HypoPG 2?

- Julien Rouhaud (From Paris, France)
  - **Working on HypoPG, POWA and some contribution to PostgreSQL**

- Yuzuko Hosoya (From Tokyo, Japan)
  - **NTT Open Source Software Center**
  - **Working on HypoPG**
  - **Interested in Partitioning/Planner**

Agenda

I. Introduction of HypoPG

II. Why Hypothetical Partitioning?

III. Usage & Architecture of Hypothetical Partitioning

IV. Demo

V. Future Works

VI. Summary

Introduction of HypoPG

Introduction of HypoPG

Latest version	HypoPG 2.0.0beta
Support	PostgreSQL 9.2 and above (pg10+ for hypothetical partitioning)
Github	https://github.com/HypoPG/hypopg
Documentation	https://hypopg.readthedocs.io

HypoPG 1.X

- Supports hypothetical Indexes
  - Allows users to define hypothetical indexes on real tables and data
  - If hypothetical indexes are defined on a table, planner considers them along with any real indexes
  - Outputs queries' plan/cost with EXPLAIN as if hypothetical indexes actually existed

- Helps index design tuning

HypoPG 2 (beta is available!)

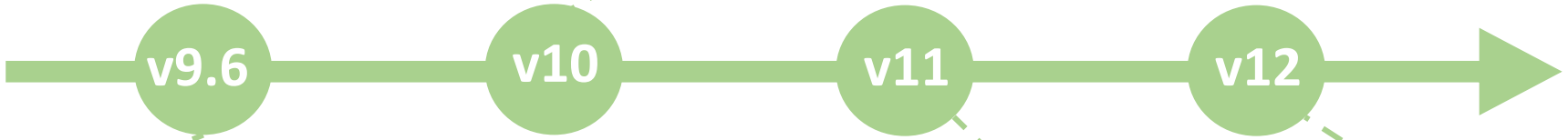
- Multiple months of work!
  - Original idea from Hosoya Yuzuko
- Supports hypothetical partitioning (PostgreSQL 10 and above)
  - Allows users to try/simulate different hypothetical partitioning schemes on real tables and data
  - Outputs queries' plan/cost with EXPLAIN using hypothetical partitioning schemes
- Helps partitioning design tuning

Why Hypothetical Partitioning?

PostgreSQL Partitioning

- Declarative Partitioning was introduced in PostgreSQL 10
- Improved in all new versions

Declarative Partitioning was introduced
- Partitioning enabled by CREATE TABLE statement
- RANGE/LIST Partitioning



Partitioning by combination of following features
- Table Inheritance
- Check Constraint
- Trigger

Many enhancements of partitioning
- HASH Partitioning
- Partitioned Index
- Default Partition
- Faster Partition Pruning
- Partitionwise Join/Aggregation
- Improvement planning for partitioning

and so on

Query Performance and Partitioning Schemes

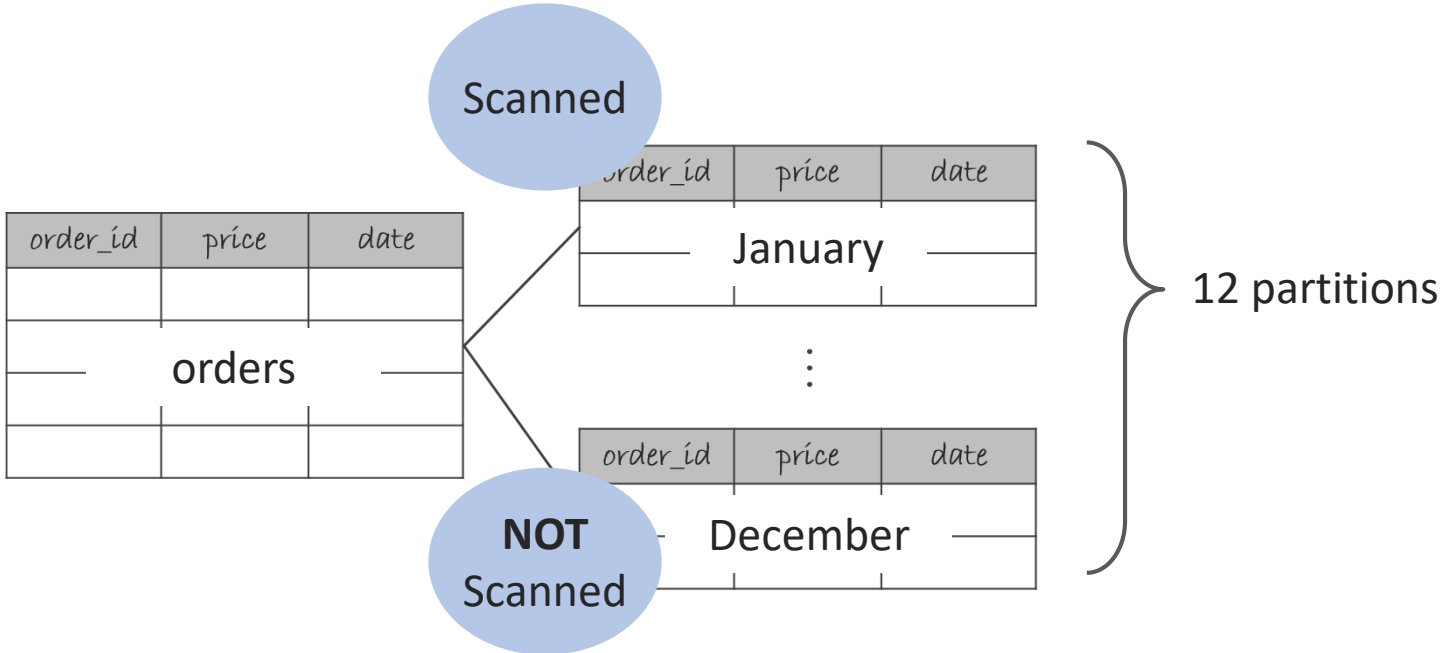
- The most IMPORTANT factor affecting query performance over a partitioned table is partitioning schemes

  - ✓ Which tables to be partitioned
  - ✓ Which strategy to be chosen
  - ✓ Which columns to be specified as a partition key
  - ✓ How many partitions to be created

Partition Pruning

- Do not scan unnecessary partitions to reduce data size to be read from the disk

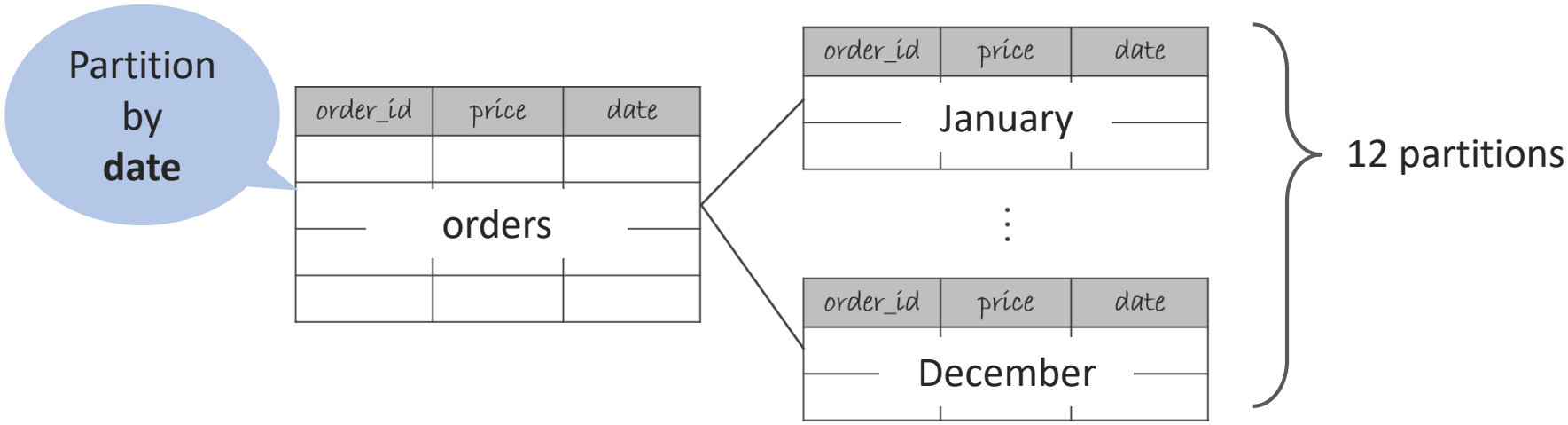
```
SELECT * FROM orders WHERE date='January';
```



Partition Pruning

- Do not scan unnecessary partitions to reduce data size to be read from the disk
- Optimal partitioning Scheme:
  Partition by the column specified in WHERE clause

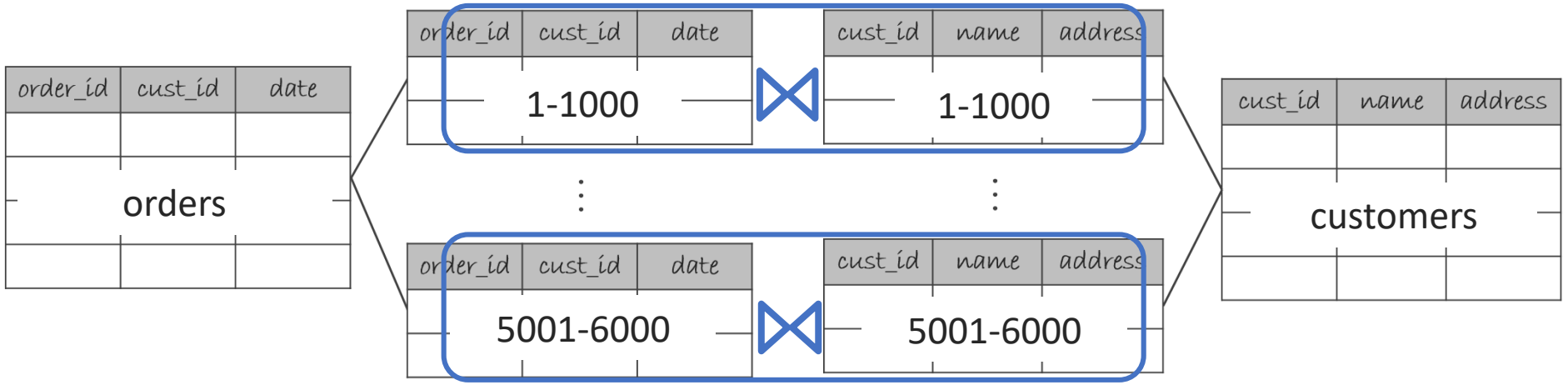
```
SELECT * FROM orders WHERE date='January';
```



Partitionwise Join

- Joining pairs of partitions might be faster than joining large tables

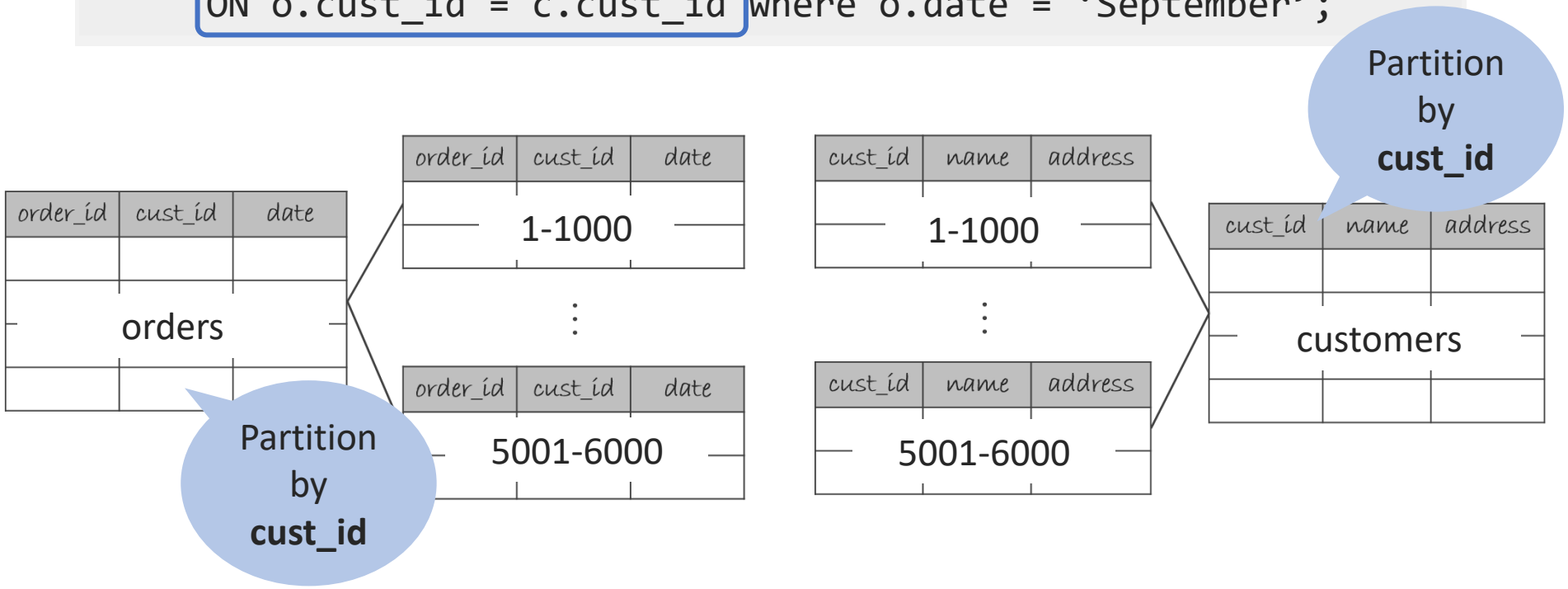
```
SELECT c.name FROM orders o LEFT JOIN customers c
ON o.cust_id = c.cust_id where o.date = ‘September’;
```



Partitionwise Join

- Joining pairs of partitions might be faster than joining large tables
- Optimal partitioning scheme:
  Partitioned by the column specified in JOIN clause

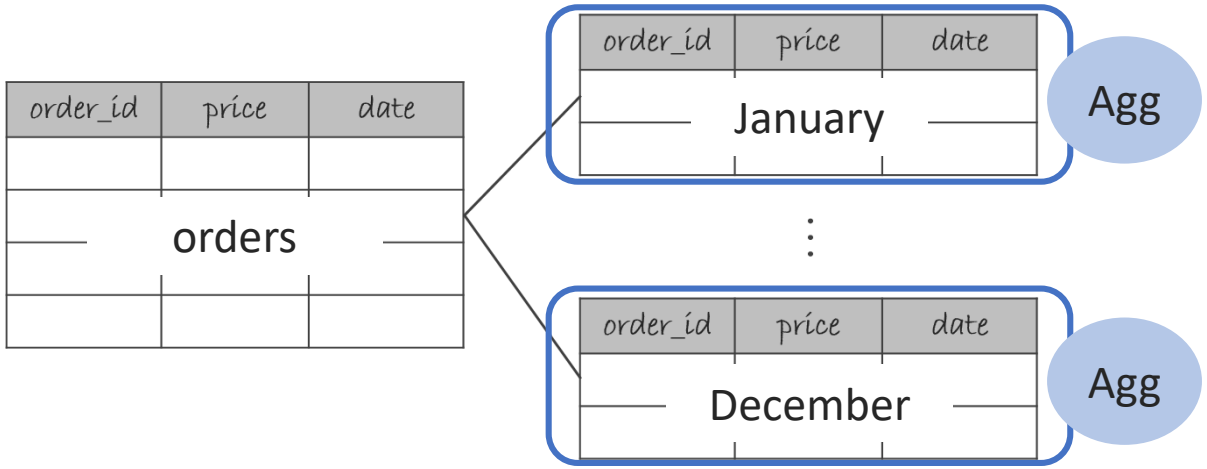
```
SELECT c.name FROM orders o LEFT JOIN customers c
ON o.cust_id = c.cust_id where o.date = 'September';
```



Partitionwise Aggregation

- Performing aggregation per partition and combining them might be faster than performing aggregation on large table

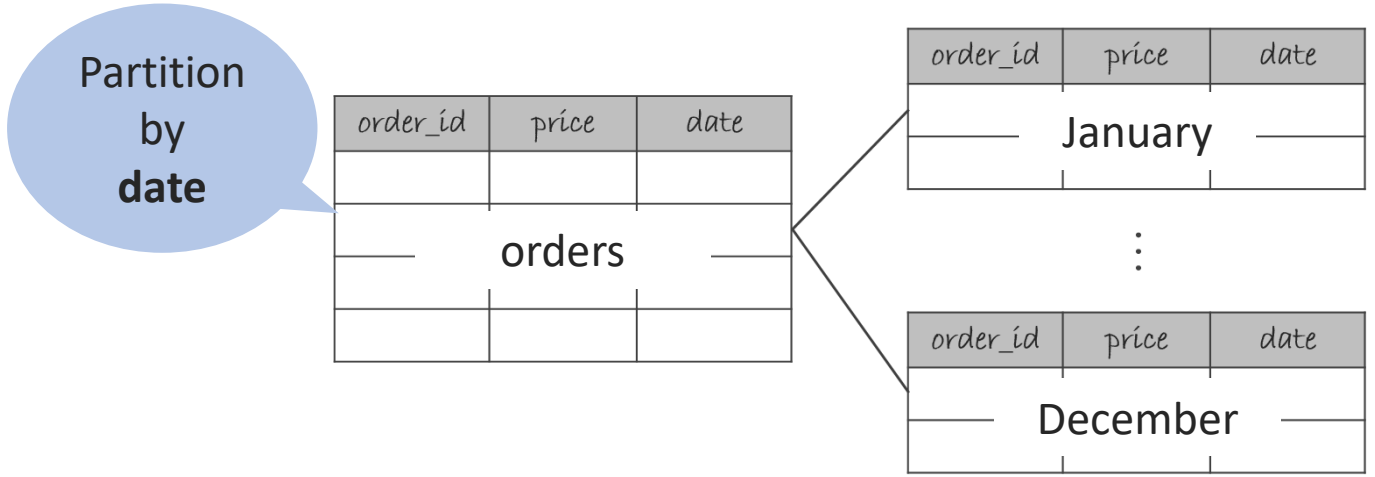
```
SELECT date, sum(price) FROM orders GROUP BY date;
```



Partitionwise Aggregation

- Performing aggregation per partition and combining them might be faster than performing aggregation on large table
- Optimal partitioning scheme:
  Partitioned by the column specified in GROUP BY clause

```
SELECT date, sum(price) FROM orders GROUP BY date;
```

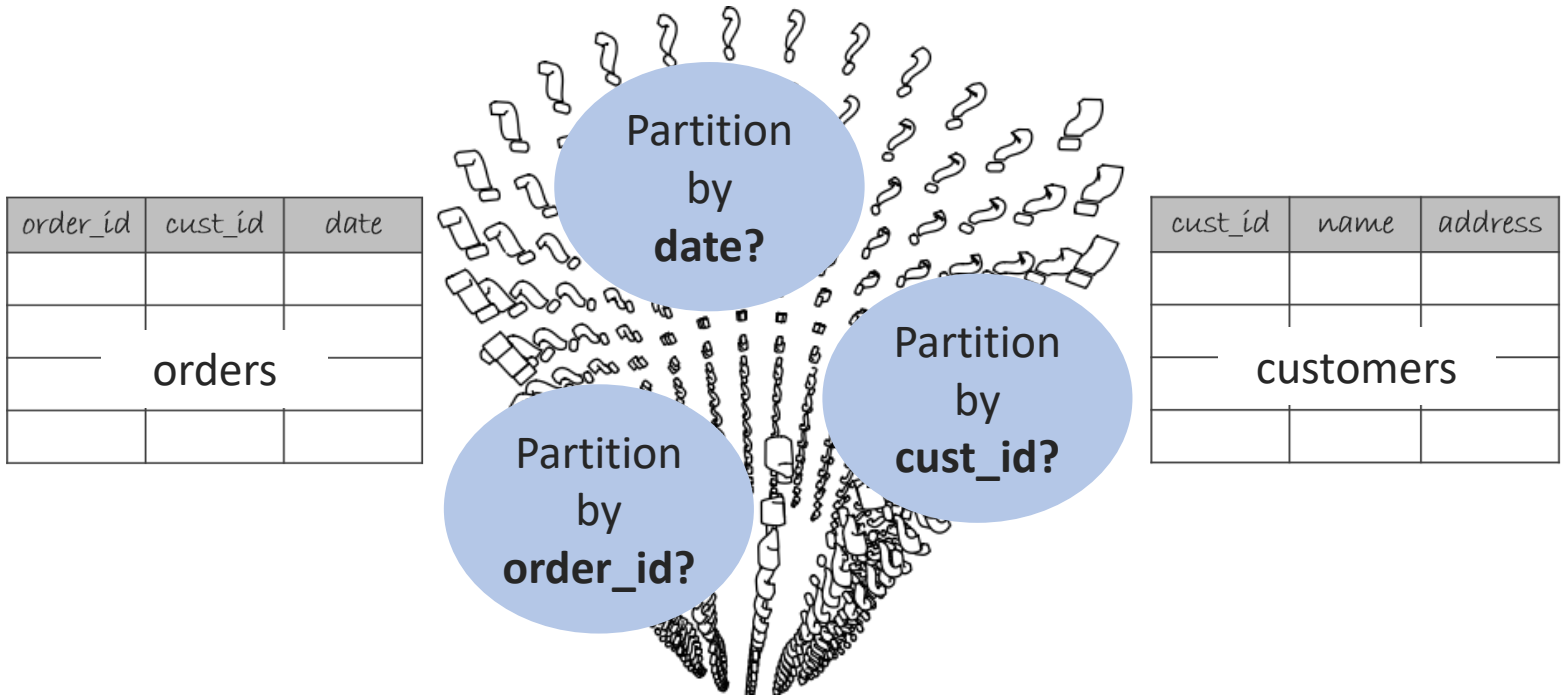


Question:
How do you partition these tables?

- Assume this situation

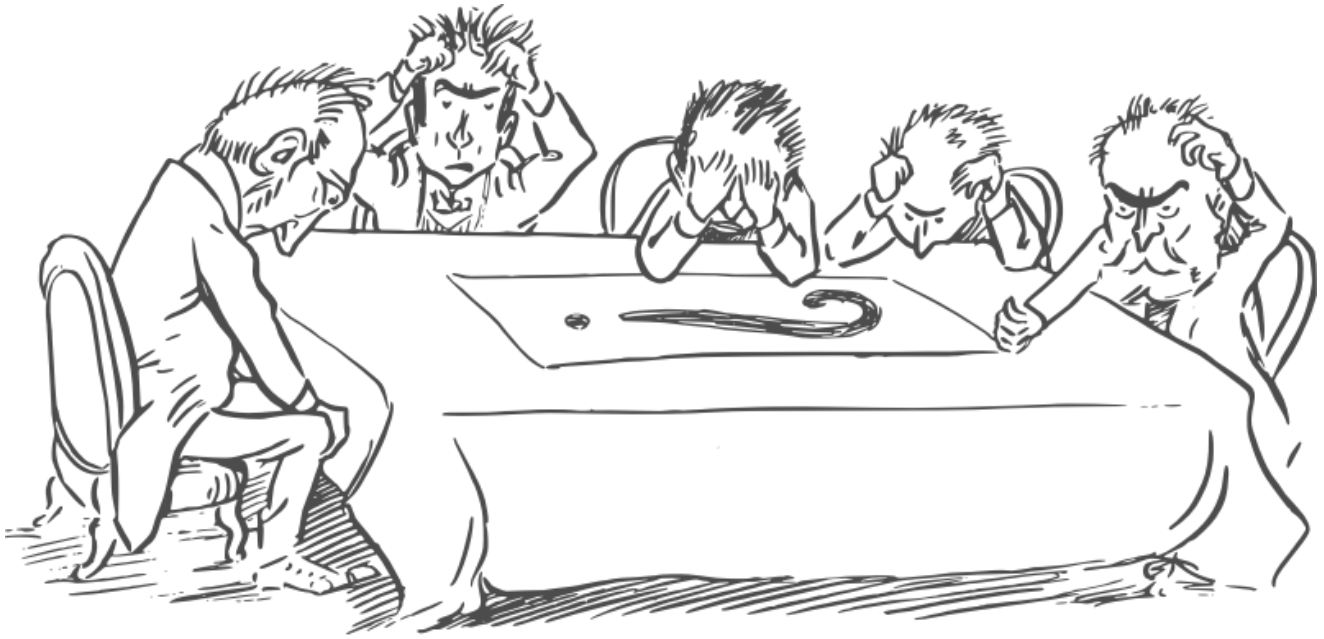
```
SELECT * FROM orders WHERE date='January';
```

```
SELECT c.name FROM orders o LEFT JOIN customers c
ON o.cust_id = c.cust_id where o.date = 'September';
```



It’s VERY HARD …

- There are often multiple choices on partitioning schemes
- Queries’ plan/cost are largely affected by data size and parameter settings



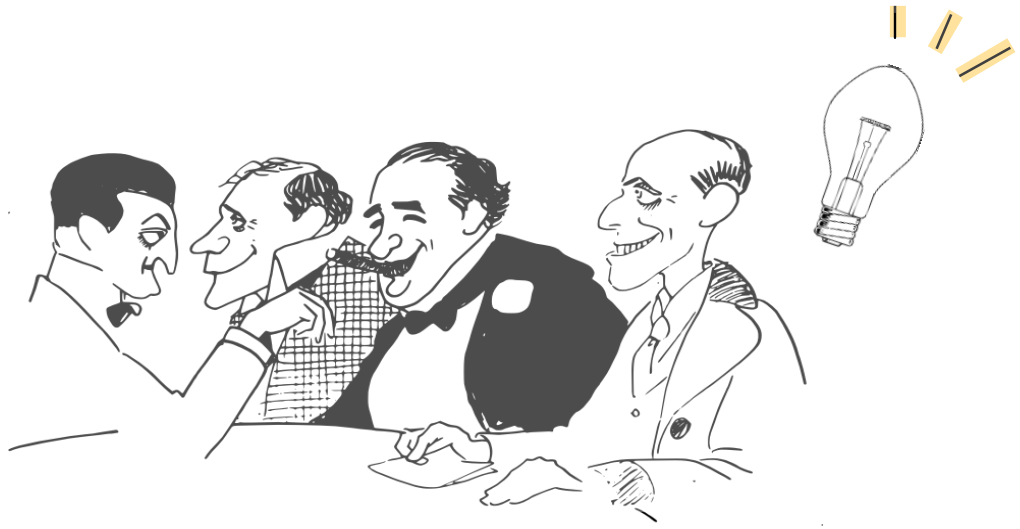
It’s VERY HARD …

- There are often multiple choices on partitioning schemes
- Query plan/cost are largely affected by data size and parameter settings

Do you want to know how your queries would behave before actually partitioning tables?

Hypothetical Partitioning is Helpful!

- You can quickly check how your queries would behave if certain tables were partitioned without actually partitioning any tables and wasting any resources or locking anything
- You can try different partitioning schemes in a short time



Usage & Architecture of Hypothetical Partitioning

How to Create Hypothetical Partitioning Schemes

- Need to have an actual plain table
- Create hypothetical partitioning schemes using following function:

For hypothetical partitioned table

```
hypopg_partition_table
('table_name', 'PARTITION BY strategy(column)')
```

For hypothetical partitions
NOTE: the 3rd argument is only for subpartitioning

```
hypopg_add_partition
('table_name',
 'PARTITION OF parent FOR VALUES partition_bound',
 'PARTITION BY strategy(column)')
```

How to Estimate Cost/Rows

- HASH partitioning (simple example, no subpartition)

  For each partition
  - get its modulus
  - compute the total fraction of the root table that should be in the given partition

- Example
  - CREATE TABLE t1 … FOR VALUES WITH (MODULUS 10, REMAINDER 1)
  - CREATE TABLE t2 … FOR VALUES WITH (MODULUS 5, REMINDER 2)
  - Partition t1 would have 10% of all rows
  - Partition t2 would have 20% of all rows

How to Estimate Cost/Rows

- RANGE/LIST partitioning (simple example, no subpartition)

  For each partition

  - get the partition's constraint
  - compute the selectivity for those constraints

- Example
  - CREATE TABLE t1 … FOR VALUES FROM (1) TO (10)
  - the constraints are

    id >= 1 AND id < 10
  - estimate this predicate using standard PostgreSQL selectivity functions

But this has defaults!

- This approach can lead to very bad estimates in some cases
  (list partitioning, subpartitioning, complex partitioning expressions…)
- We added a function to compute accurate statistics according to the defined partitioning scheme:
  hypopg_analyze()
- We also store the number of estimated rows in each partition during the process
- Be careful, this has to be explicitly called!
  (there’s no autovacuum for hypopg_analyze)

How to Create Hypothetical Statistics

- Create hypothetical statistics for hypothetical partitions using following function:

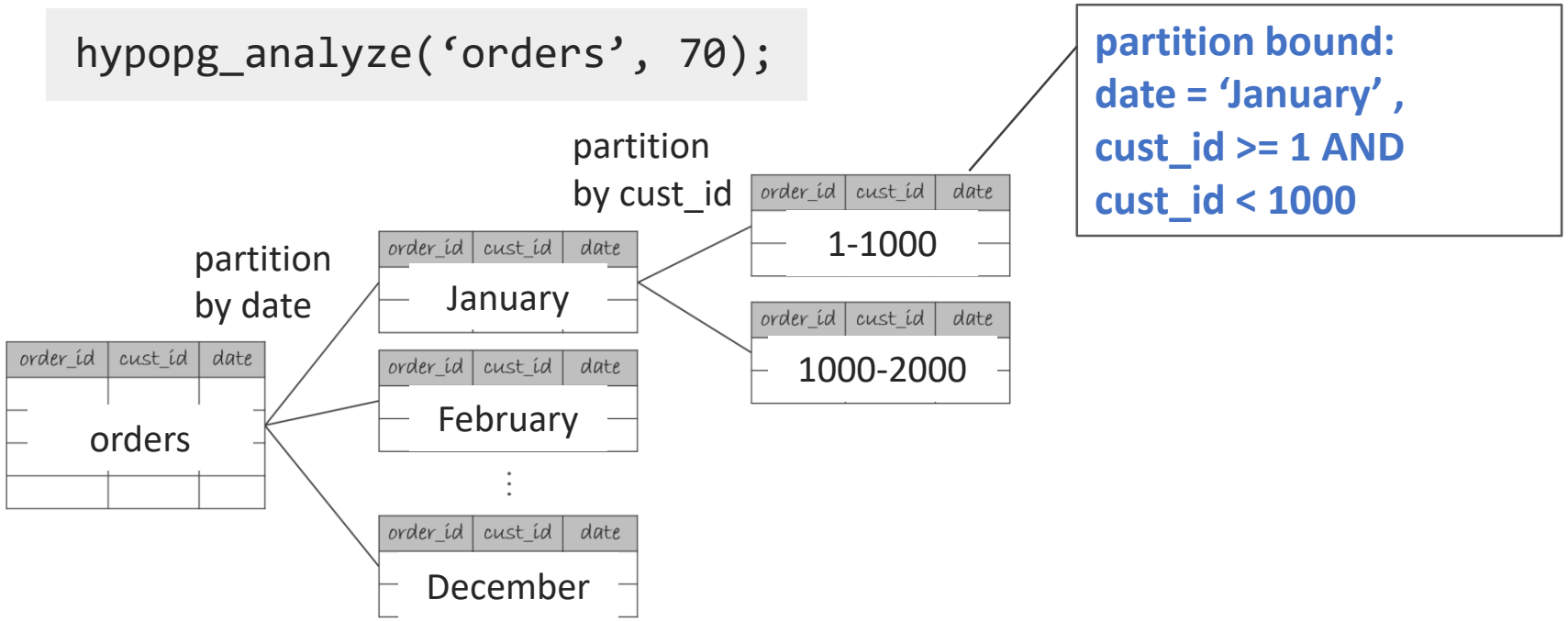
```
hypopg_analyze
('table_name', percentage for sampling)
```

- The table specified in the first argument of this function must be a root table
- Statistics of all leaf partitions, which are related to the given table, are created

How hypopg_analyze() Works

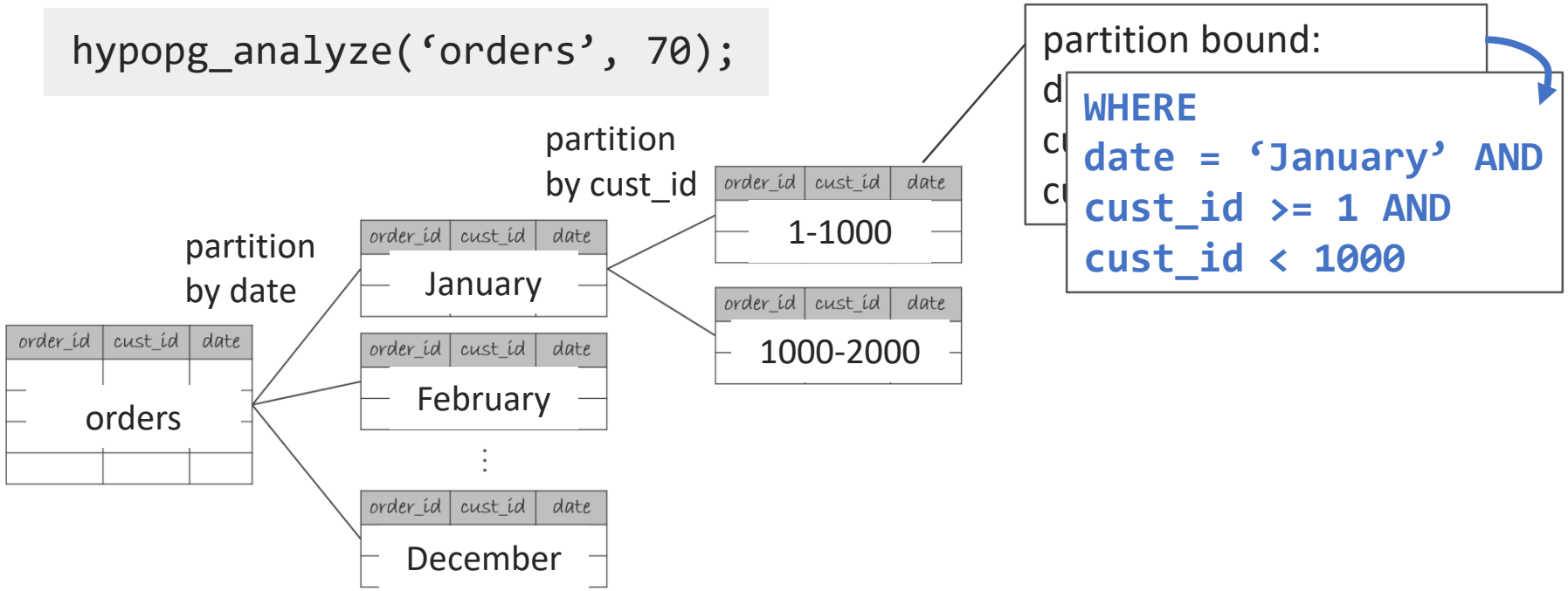
- For each partition, get the partition bounds including its ancestors if any

```
hypopg_analyze('orders', 70);
```



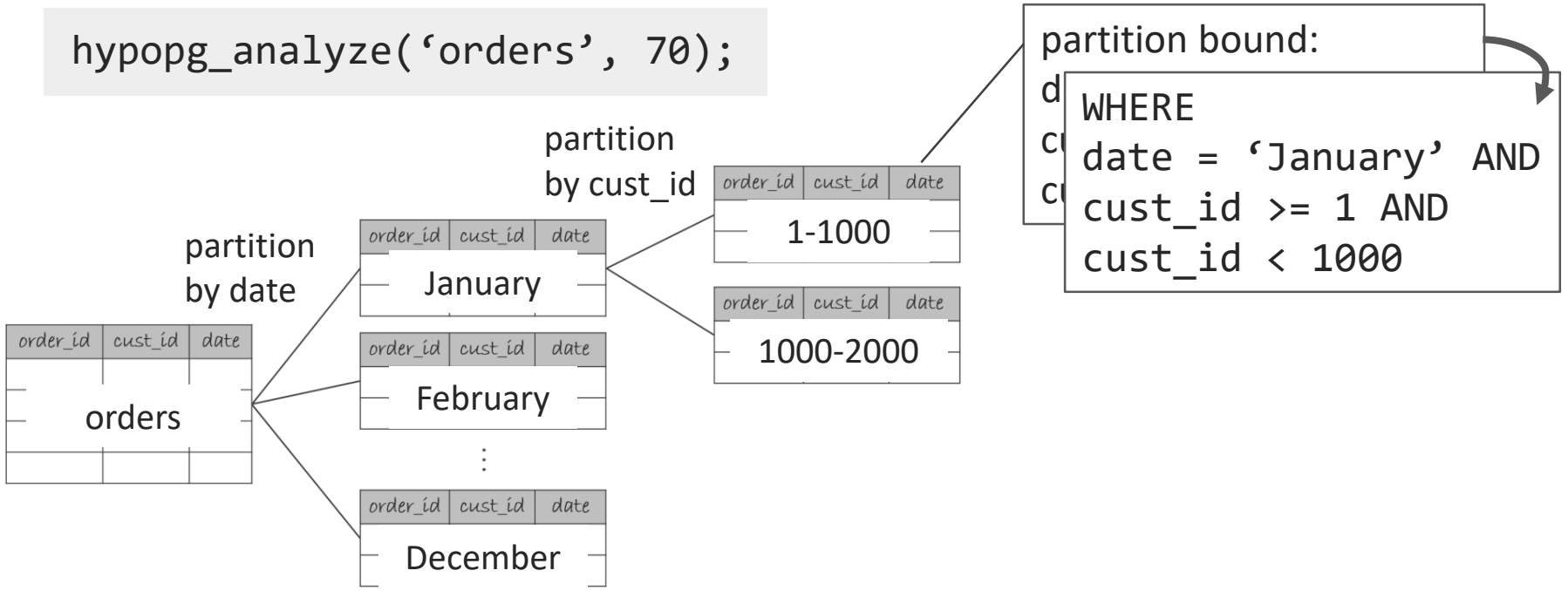
How hypopg_analyze() Works

- For each partition, get the partition bounds including its ancestors if any
- Generate a WHERE clause according to the partition bound



How hypopg_analyze() Works

- For each partition, get the partition bounds including its ancestors if any
- Generate a WHERE clause according to the partition bound
- Compute stats for sampling data got by TABLESAMPLE and the WHERE clause



TABLESAMPLE Clause

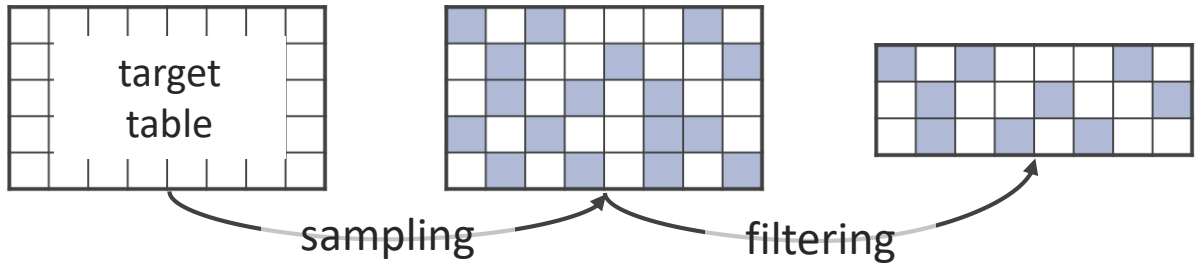
Get sampling data from a target table according to percentage

```
SELECT select_expression FROM table_name
TABLESAMPLE sampling_method(percentage) WHERE condition
```

✓ Two kinds of sampling method
- SYSTEM: pick data by the page
- BERNOULLI: pick data by the row

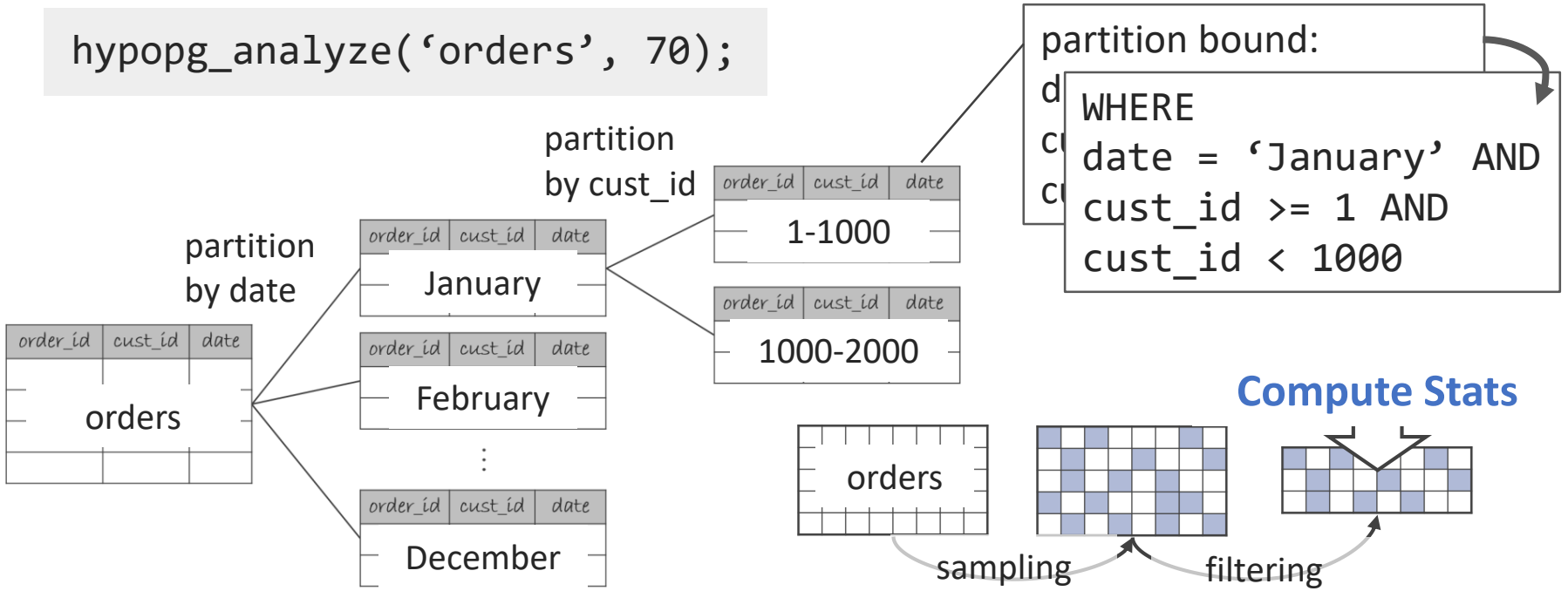
✓ Specify WHERE clause

eliminate data that doesn't satisfied WHERE condition



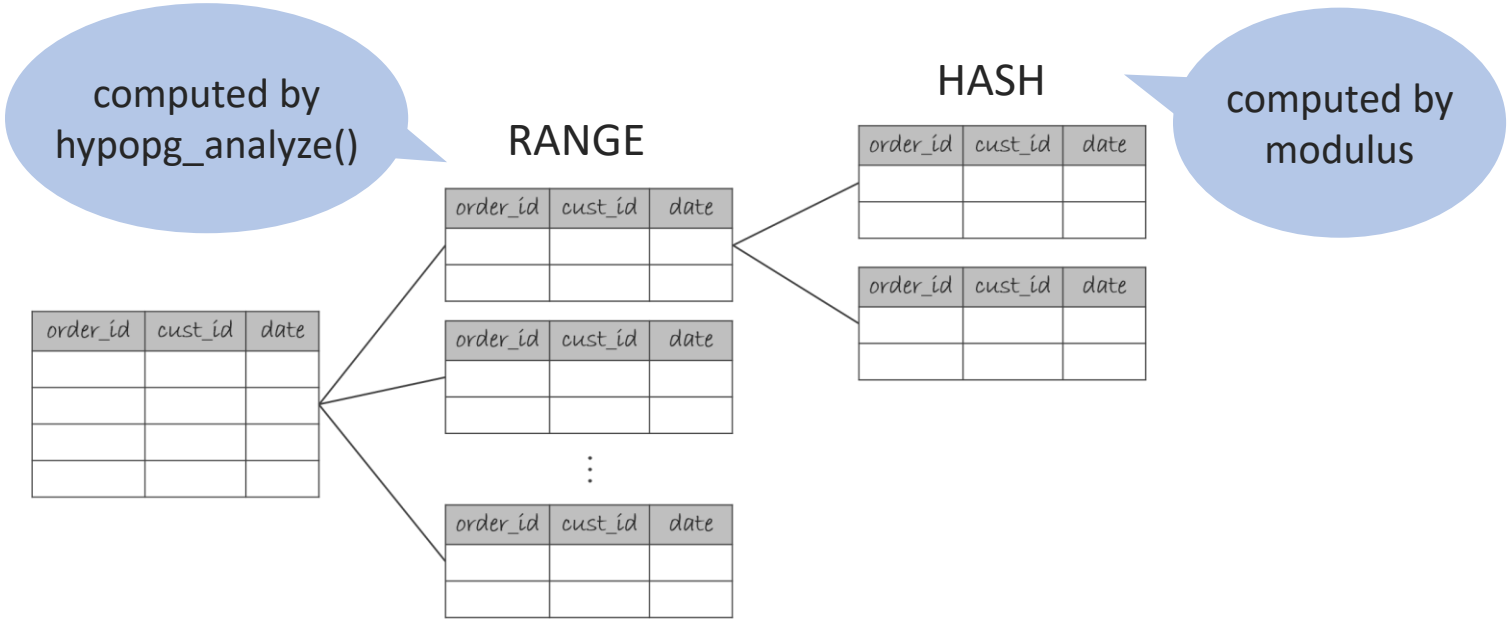
How hypopg_analyze() Works

- For each partition, get the partition bounds including its ancestors if any
- Generate a WHERE clause according to the partition bound
- Compute stats for sampling data got by TABLESAMPLE and the WHERE clause



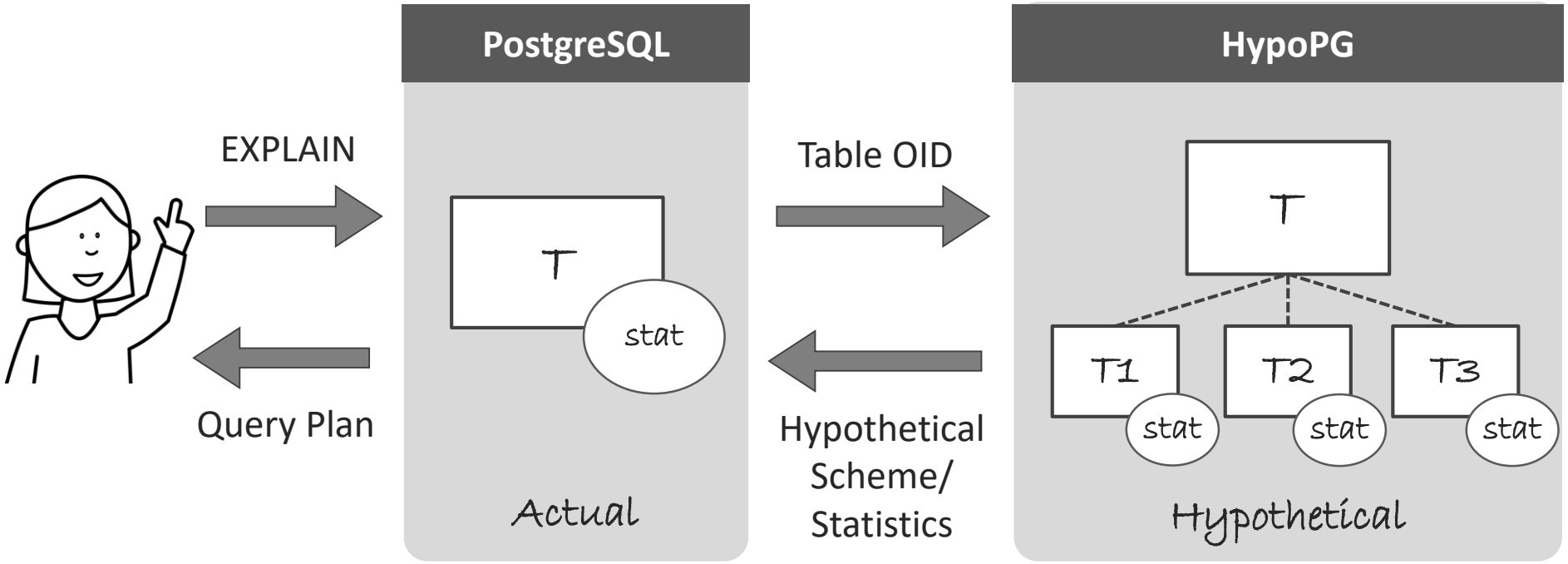
Exception

- hypopg_analyze() won't retrieve constraints for a hash partition
- instead,
  - sum the fractions of rows each level of partition defined with a hash partitioning scheme
  - do a simple ratio of the previously computed values (for all non-hash partitions)



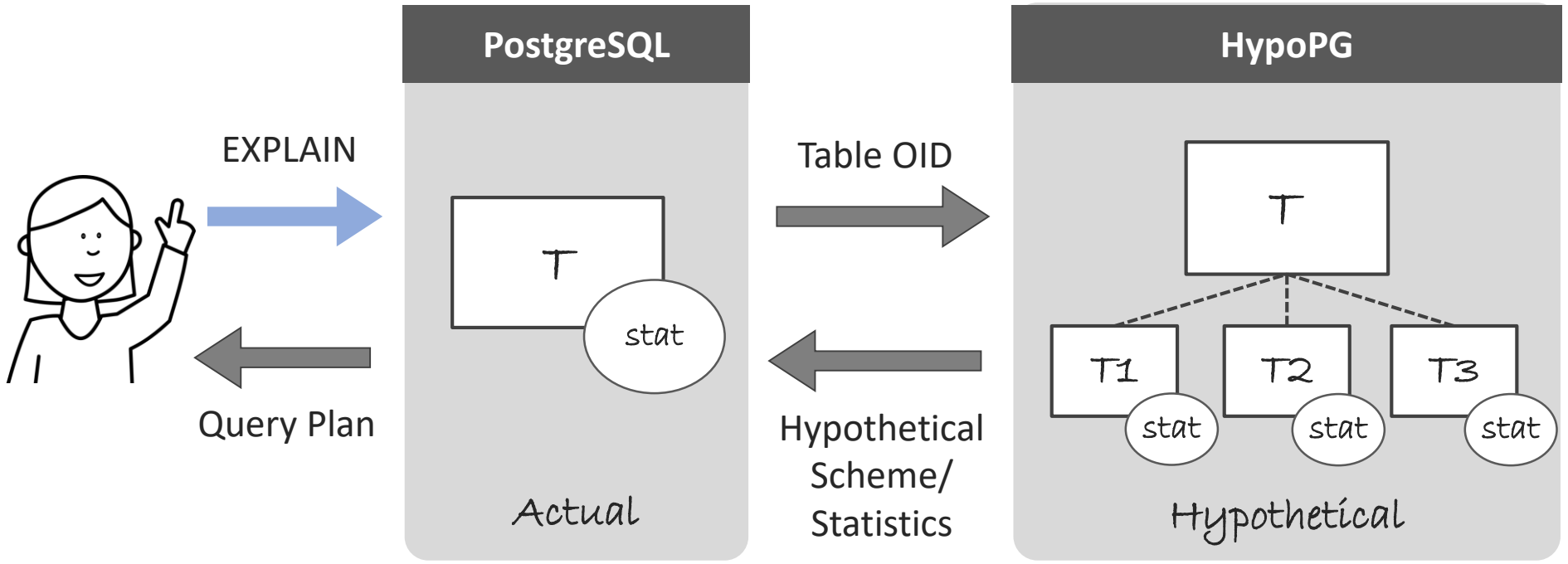
Architecture

HypoPG uses four kinds of hooks
(five kinds of hooks for PostgreSQL 10)



Architecture

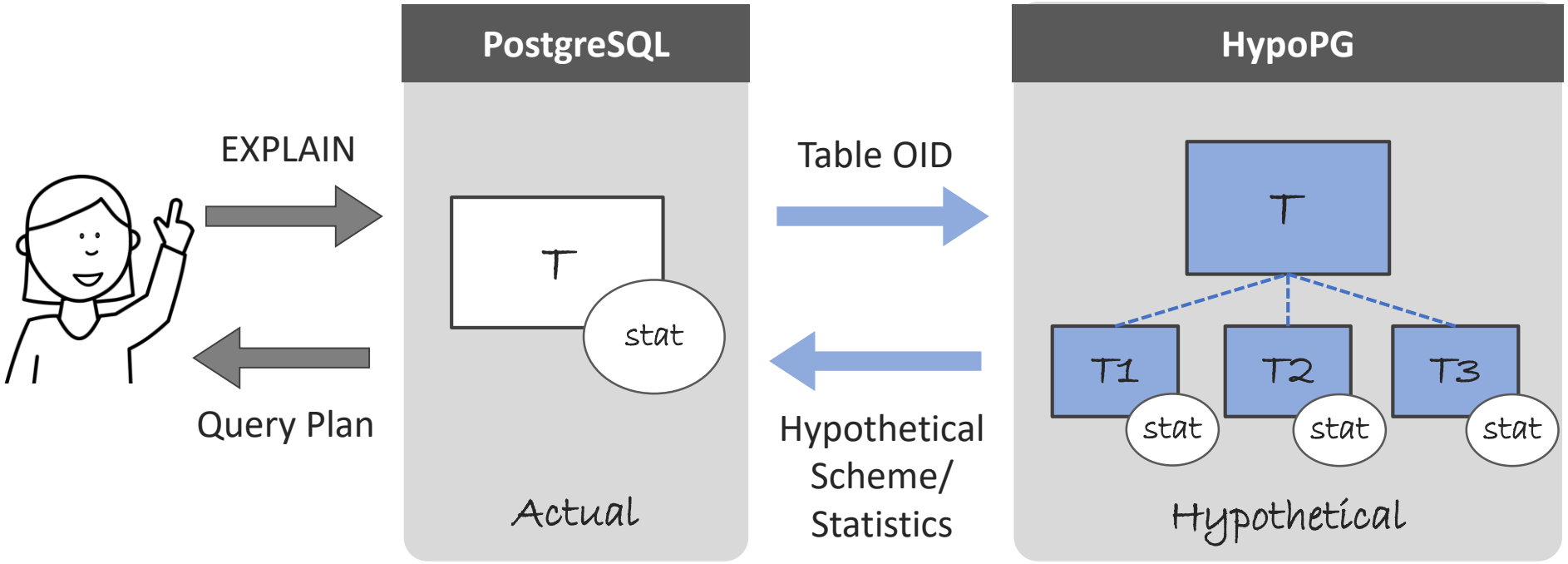
1 Using **ProcessUtility_hook**, HypoPG detects an EXPLAIN without ANALYZE option and saves it in a local flag for following steps



Architecture

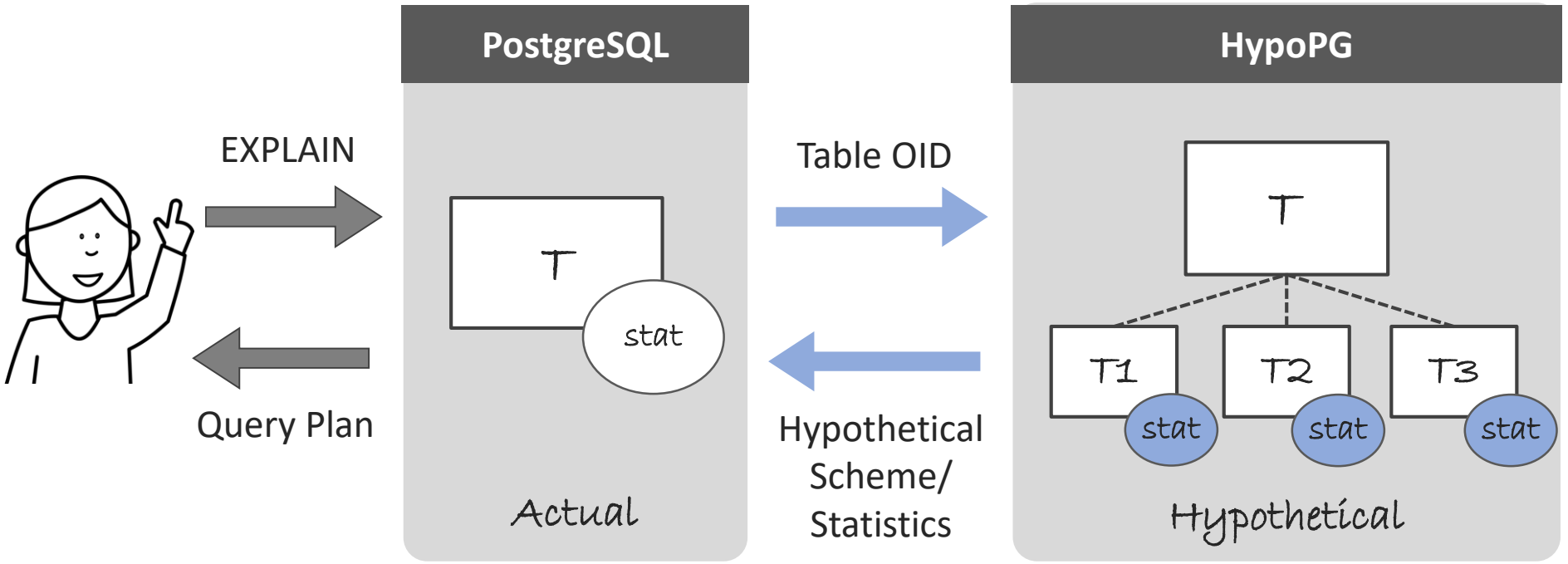
2

Using **get_relation_info_hook**, hypothetical partitioning scheme is injected if the target table is partitioned hypothetically. This involve modifying a lot of internal structure to make them identical to what real partitioning would have generated.



Architecture

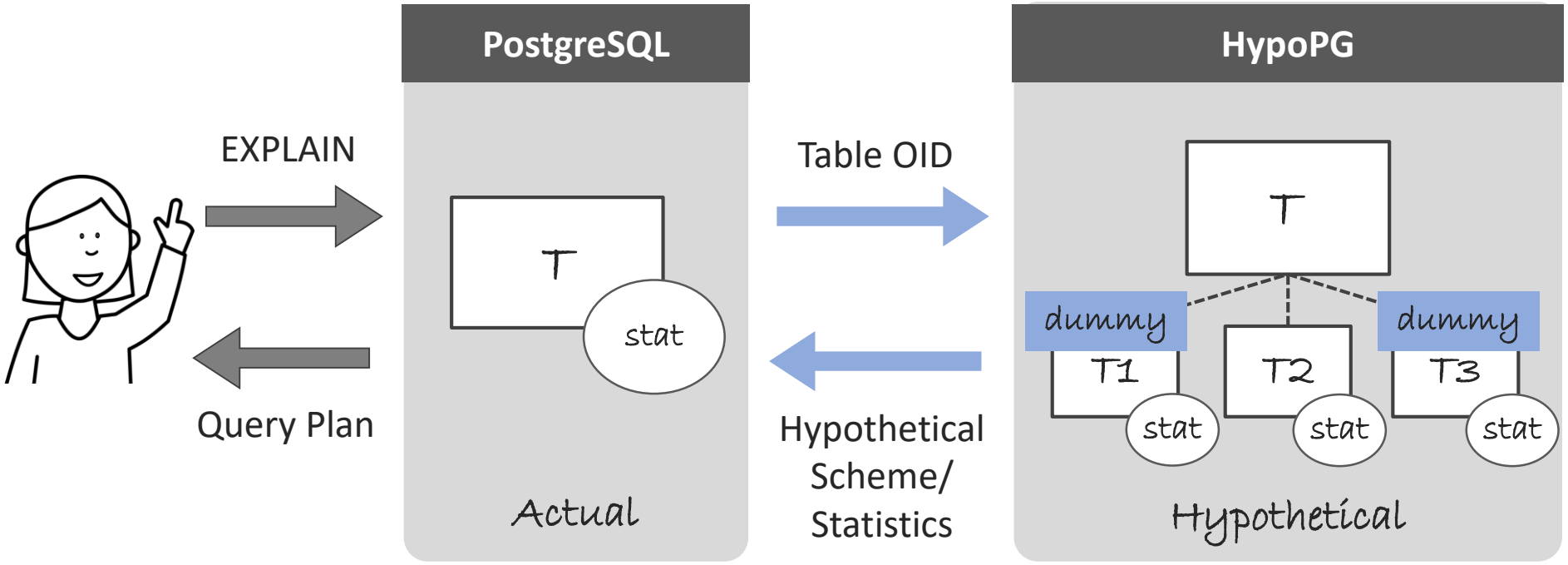
3 Using **get_relation_stats_hook**, hypothetical statistics are injected to estimate correctly if they were created in advance



Architecture

3'

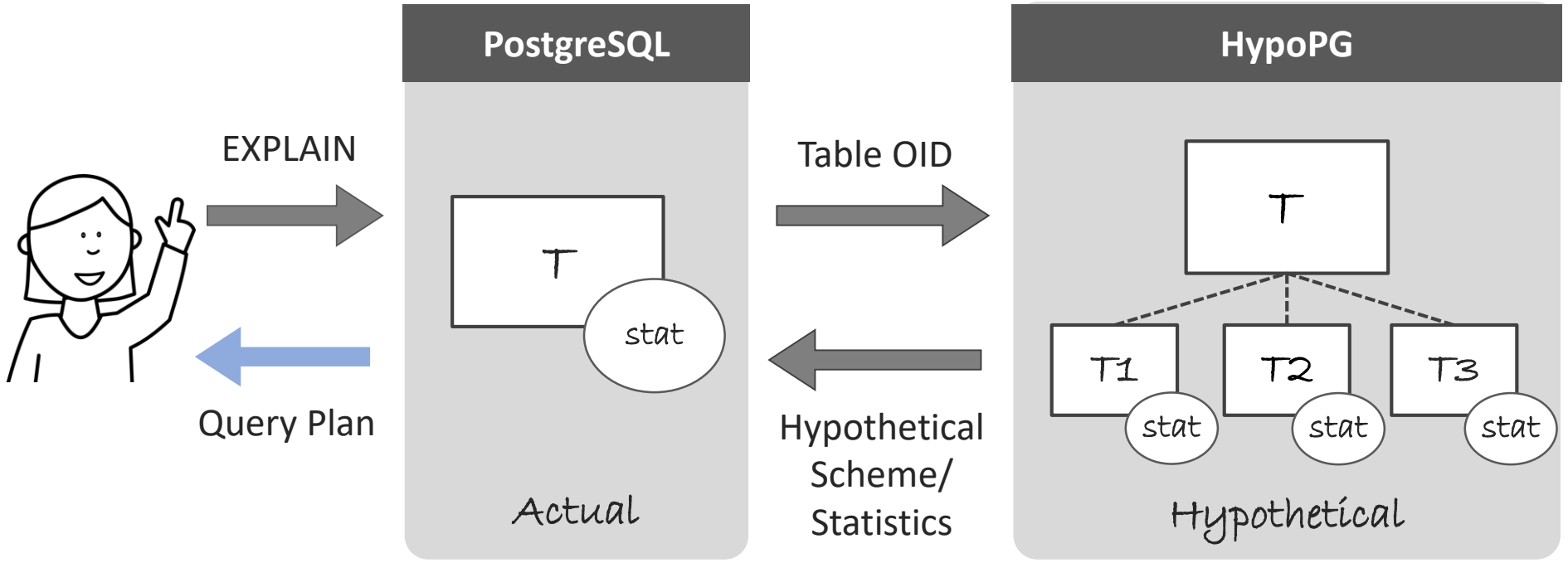
(for PostgreSQL 10)
Using **set_rel_pathlist_hook**, a partition which is need not be scanned is marked as dummy for partition pruning



Architecture

4

Using **ExecutorEnd_hook**, the EXPLAIN flag is removed.
Finally a **query plan** using hypothetical partitioning schemes is displayed!!



DEMO

DEMO time!

- Create a hypothetical partitioning scheme and execute some simple queries

  - A simple **customers** table to be partitioned **actually**:

```
customers (cust_id integer PRIMARY KEY,
           name TEXT, address TEXT)
```

  - A simple **orders** table to be partitioned **hypothetically**:
    NOTE: for convenience, this table is named **hypo_**orders to quickly identify it in the plans.

```
hypo_orders (orders_id int PRIMARY KEY,
             cust_id int, price int, date date)
```

What You Can Do

- Simulate RANGE/LIST/HASH Partitioning
- Simulate SELECT queries
  Partition Pruning, Partitionwise Join/Aggregation,
  N-way Join, Parallel Query
- Simulate INSERT queries
- Simulate multi-level hypothetical partitioning
- Simulate a default partition (which can also be partitioned)
- Simulate indexes on hypothetical partitions
  Both actual and hypothetical indexes

Limitations

- Only for plain tables, not on already partitioned tables
  Inheritance based and declarative partitioning
- Table name can’t be changed in explain, an alias is used instead to show the hypothetical partition name
- Do not support UPDATE/DELETE queries
- Do not support PostgreSQL 12 yet

Future Works & Summary

What’s next?

- Have a better integration in PostgreSQL core to ease our limitations
  - Support UPDATE/DELETE queries
  - Support already partitioned tables
- Support PostgreSQL 12 (and future versions)
- Add automated advisor feature
  pg_qualstats can help to find columns used in all queries on a given table if any

Summary

HypoPG2 supports hypothetical partitioning

- Allows users to try/simulate different hypothetical partitioning schemes on real tables and data
- Outputs queries' plan/cost with EXPLAIN using hypothetical partitioning schemes

Hypothetical partitioning helps you all to design partitioning schemes

- You can quickly check how your queries would behave if certain tables were partitioned without actually partitioning any tables and wasting any resources
- You can try different partitioning schemes in a short time

We'd be happy to have some feedbacks!

THANK YOU!

Any questions?

Julien Rouhaud
rjuju123@gmail.com
@rjuju123

Yuzuko Hosoya
hosoya.yuzuko@lab.ntt.co.jp
@pyyycha