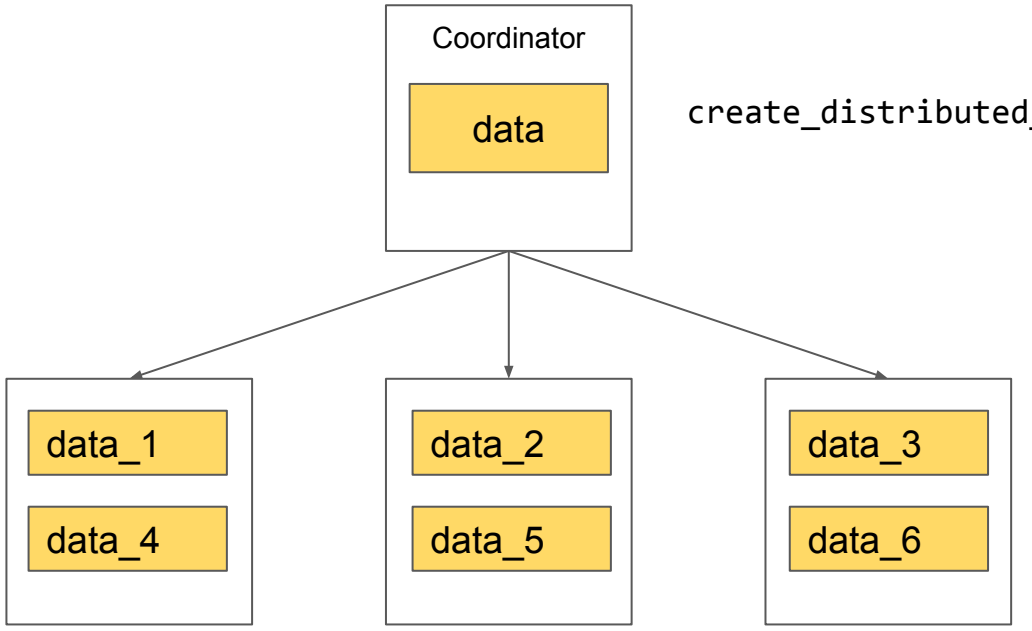
Distributing Queries the Citus Way

Fast and Lazy

Marco Slot <marco@citusdata.com>

What is Citus?

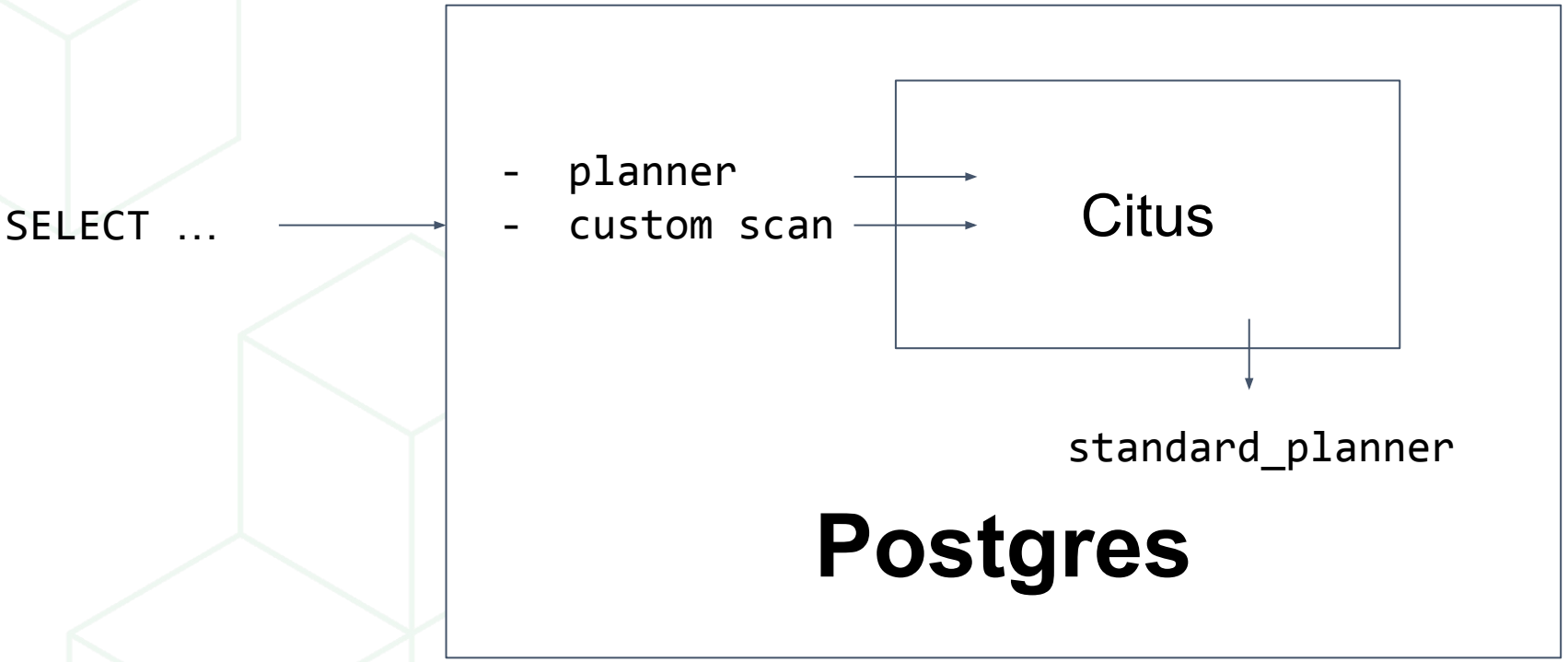
Citus is an open source extension to Postgres (9.6, 10, 11) for transparently distributing tables across many Postgres servers.




```
create_distributed_table('data', 'tenant_id');
```

How does Citus work?

Citus uses hooks and internal functions to change Postgres' behaviour and leverage its internal logic.



citusdata

Different use cases for scaling out

There are different use cases that can take advantage of distributed databases, in different ways.

Examples:

- Multi-tenant SaaS app needs to scale beyond a single server
- Real-time analytics dashboards with high data volumes
- Advanced search across large, dynamic data sets
- Business intelligence

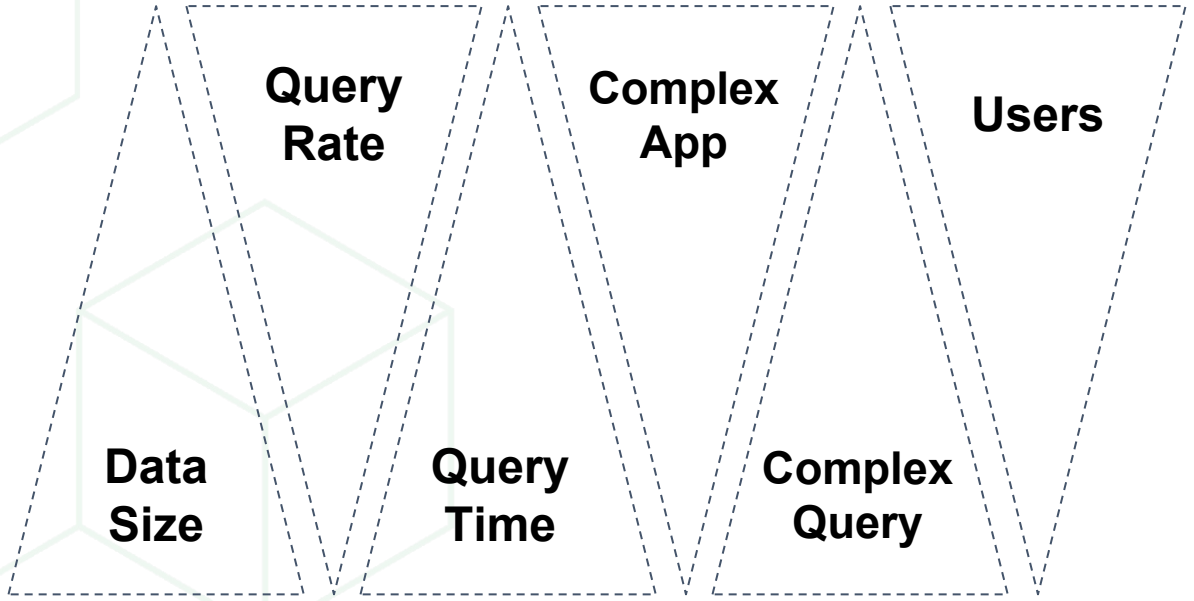
Citus planner(s)

Layered planner accommodates different workloads.

Router planner	Multi-tenant OLTP
Pushdown planner	Real-time analytics, search
Recursive (Subquery/CTE) planning	Real-time analytics, data warehouse
Logical planner	Data warehouse

Citus planner(s)

Layered planner accommodates different workloads.



Multi-tenant OLTP

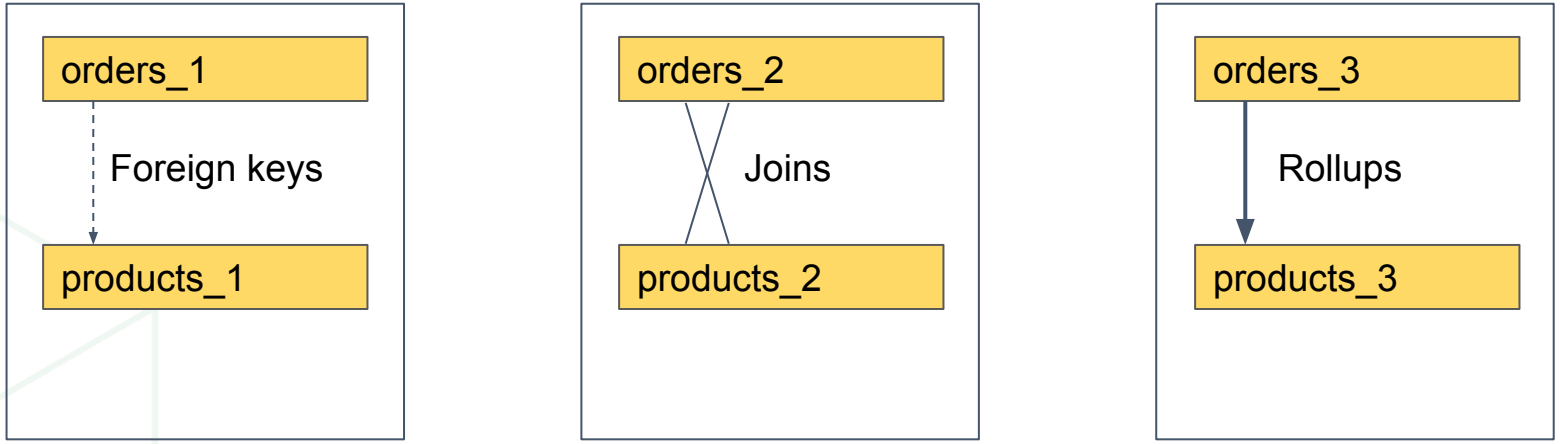
Real-time analytics, search

Real-time analytics, data warehouse

Data warehouse

Co-located distributed tables

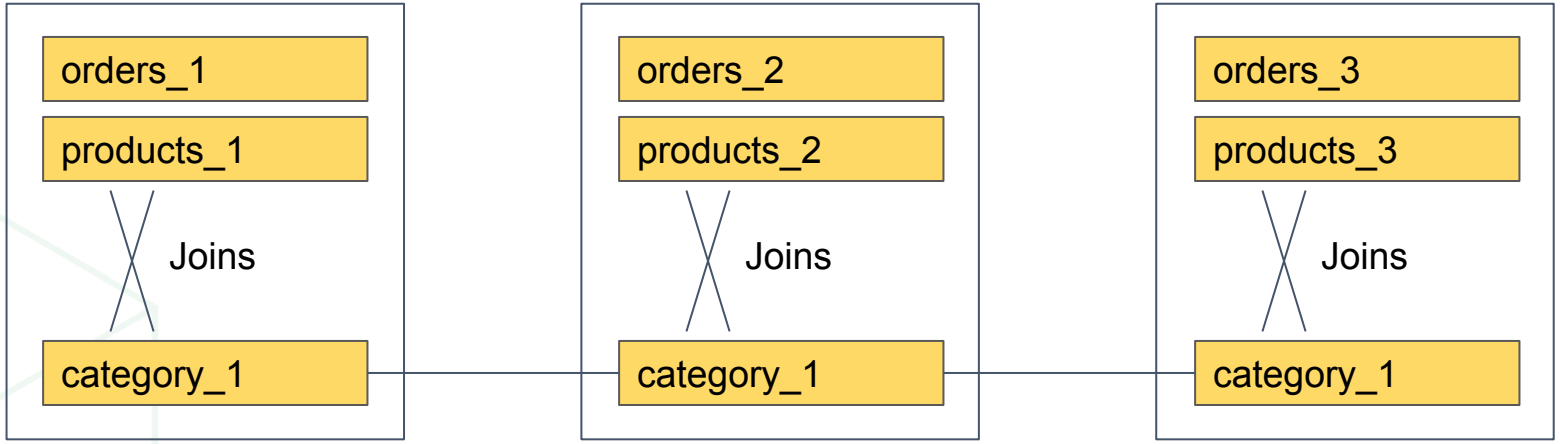
Tables are automatically assigned to co-location groups, which ensure that rows with the same distribution column value are on the same node.



This enables foreign keys, direct joins, and rollups (INSERT...SELECT) that include the distribution column.

Reference tables

Reference tables are replicated to all nodes such that they can be joined with distributed tables on any column.



citusdata

Router planner

How to be a “drop-in” distributed database

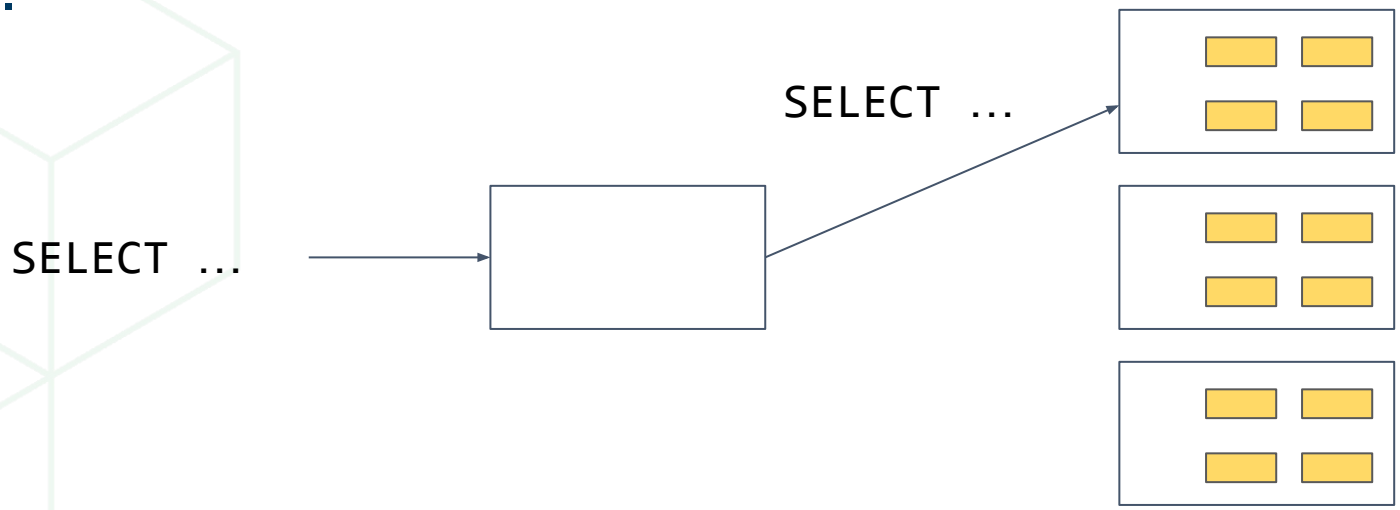
citusdata

Routable queries

Technical observation:

> If a query has `<distribution column> = <value>` filters that (transitively) apply to all tables, it can be “routed” to a particular node.

Efficiently provides full SQL support, since full query can be handled by Postgres.



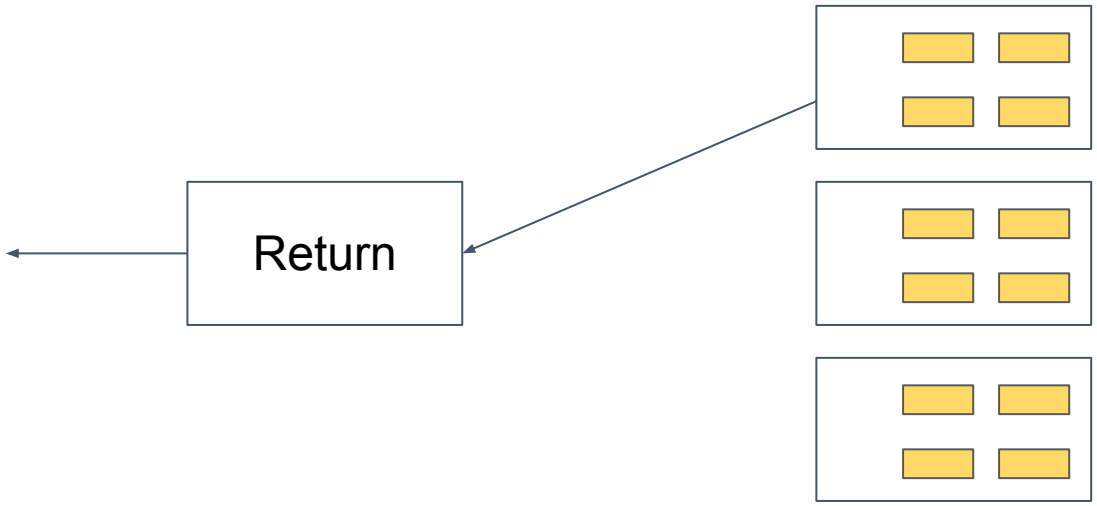
citusdata

Routable queries

Technical observation:

> If a query has `<distribution column> = <value>` filters that (transitively) apply to all tables, it can be “routed” to a particular node.

Efficiently provides full SQL support, since full query can be handled by Postgres.



citusdata

Scaling Multi-tenant Applications

Use case observation:

In a SaaS (B2B) application, most queries are specific to a particular tenant.

Can add tenant ID column to all tables and distribute by tenant ID.

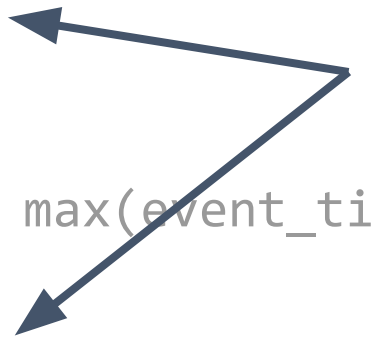
Most queries are router plannable:

Low overhead, low latency, full SQL capabilities of Postgres, scales out

Router planner with explicit filters

Can explicitly provide filters on all tables:

```
SELECT app_id, event_time
FROM (
  SELECT tenant_id, app_id, item_name
  FROM items
  WHERE tenant_id = 1783
)
LEFT JOIN (
  SELECT tenant_id, app_id, max(event_time) AS event_time
  FROM events
  WHERE tenant_id = 1783
  GROUP BY tenant_id, app_id
)
USING (tenant_id, app_id) ORDER BY 2 DESC LIMIT 10;
```



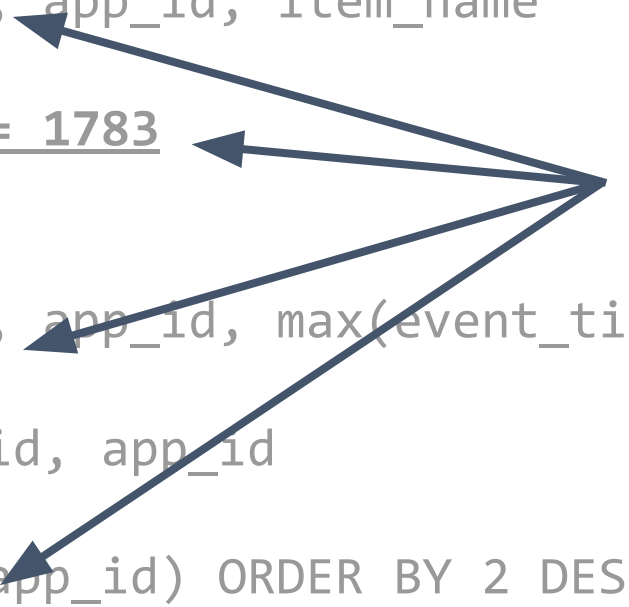
All distributed tables have filters by the same value

citusdata

Router planner with inferred filters

Citus can infer distribution column filters from joins:

```
SELECT app_id, event_time
FROM (
  SELECT tenant_id, app_id, item_name
  FROM items
  WHERE tenant_id = 1783
)
LEFT JOIN (
  SELECT tenant_id, app_id, max(event_time) AS event_time
  FROM events
  GROUP BY tenant_id, app_id
)
USING (tenant_id, app_id) ORDER BY 2 DESC LIMIT 10;
```



Filter on orders can be inferred from joins

Extracting relation filters

What does Citus need to do to infer filters?

Be lazy and call the Postgres planner:

```
planner()
-> citus_planner()
   -> standard_planner()
```

Obtain filters on all relation from Postgres planning logic

Pushdown planning

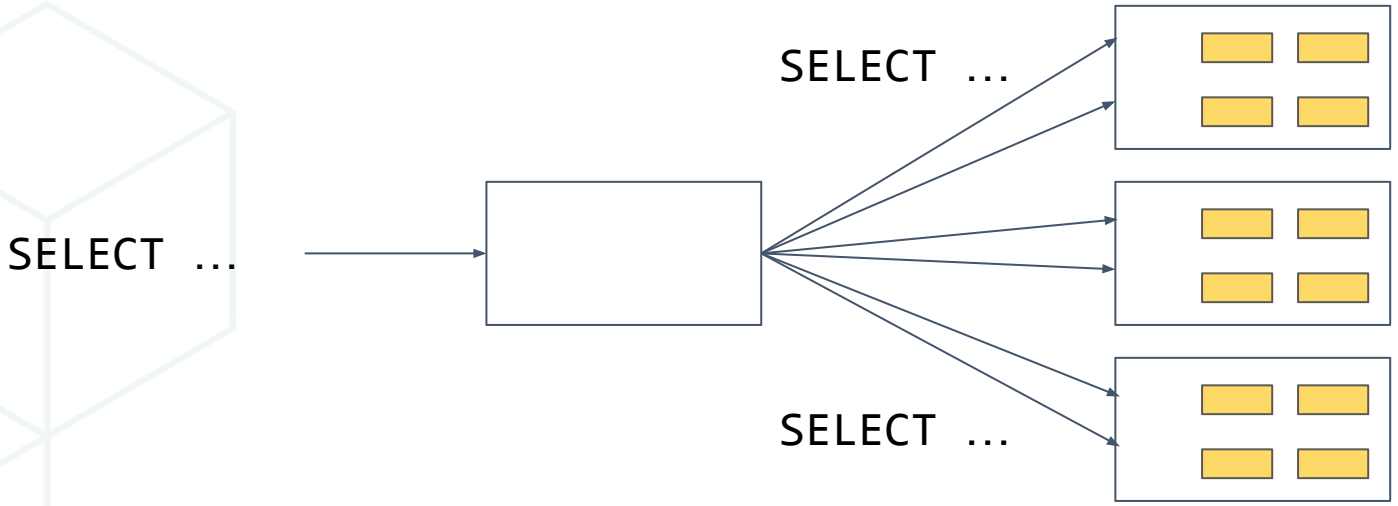
Make your workers work

citusdata

Distributed queries

Technical observation:

Most common SQL features (aggregates, `GROUP BY`, `ORDER BY`, `LIMIT`) can be distributed in a single round.

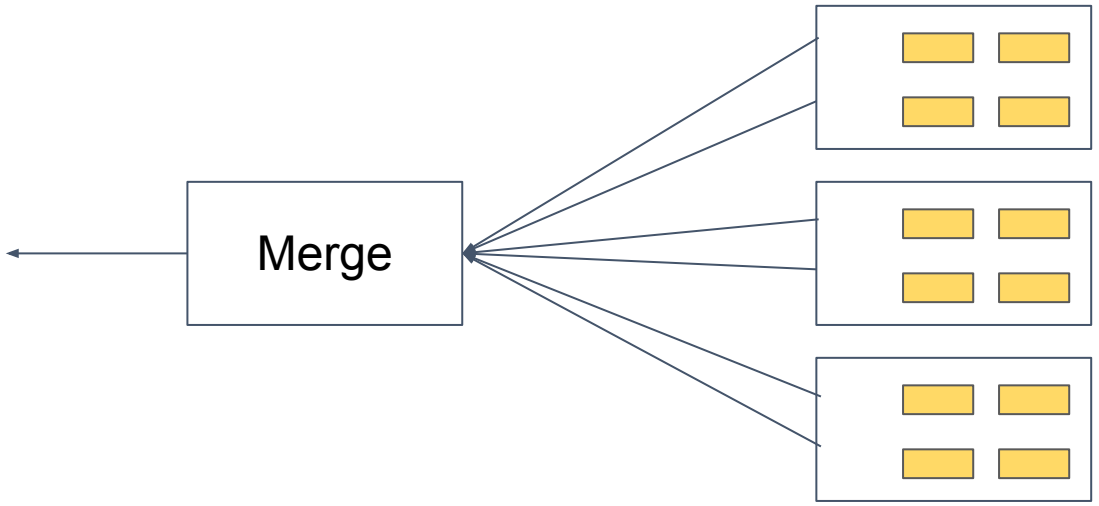


citusdata

Distributed queries

Technical observation:

Most common SQL features (aggregates, `GROUP BY`, `ORDER BY`, `LIMIT`) can be distributed in a single round.



citusdata

Merging query results

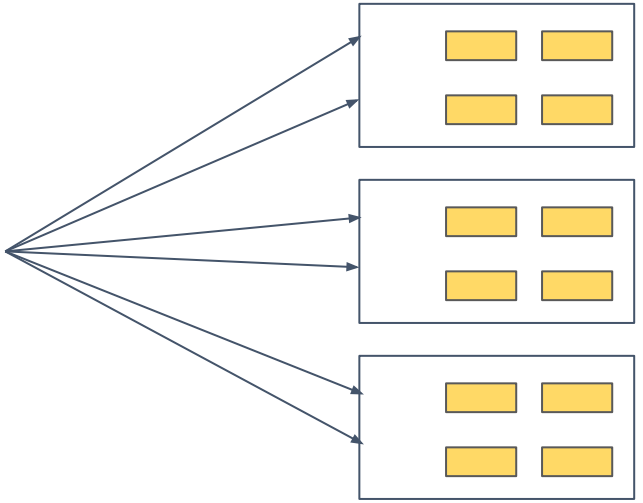
Get the top 10 pages with the highest response times:

```
SELECT page_id, avg(response_time)
FROM page_views
GROUP BY page_id
ORDER BY 2 DESC
LIMIT 10
```

Queries on shards

Queries on shards when `page_id` is the distribution column:

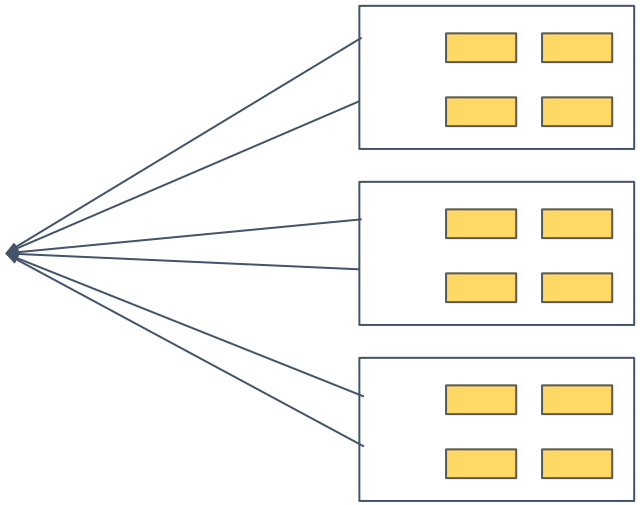
```
SELECT page_id, avg(response_time)
FROM page_views_102008
GROUP BY page_id
ORDER BY 2 DESC
LIMIT 10
```



Merging query results

When `page_id` is the distribution column: get top 10 of top 10s.

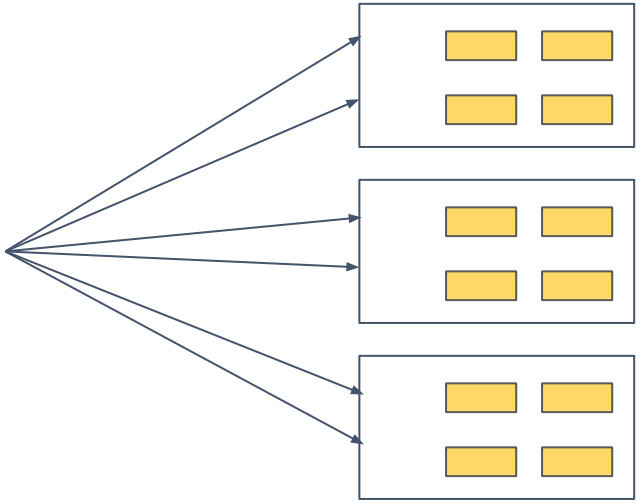
```
SELECT page_id, avg
FROM
  Concatenated results of queries on shards

ORDER BY 2 DESC
LIMIT 10
```



Queries on shards

Queries on shards when `page_id` is **not** the distribution column:

```
SELECT page_id, sum(response_time),
       count(response_time)
FROM page_views_102008
GROUP BY page_id
```



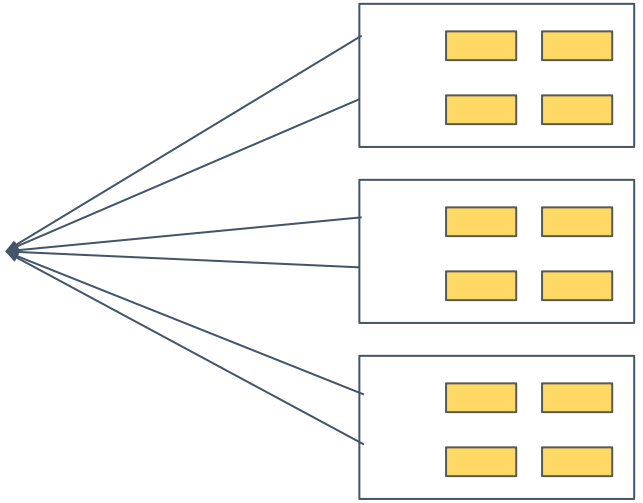
Merging query results

When `page_id` is **not** the distribution column: merge the averages

```
SELECT page_id, sum(sum) / sum(count)
FROM
```

Concatenated results of queries on shards

```
GROUP BY page_id
ORDER BY 2 DESC
LIMIT 10
```



What about subqueries?

Instead of a table, we can have joins or subqueries:

```
SELECT page_id, response_time
FROM (
  SELECT page_id
  FROM pages
  WHERE site = 'www.citusdata.com'
) p
JOIN (
  SELECT page_id, avg(response_time) AS response_time
  FROM page_views
  WHERE view_time > date '2018-03-20' GROUP BY page_id
) v
USING (page_id)
ORDER BY 2 DESC LIMIT 10;
```

Distributed queries

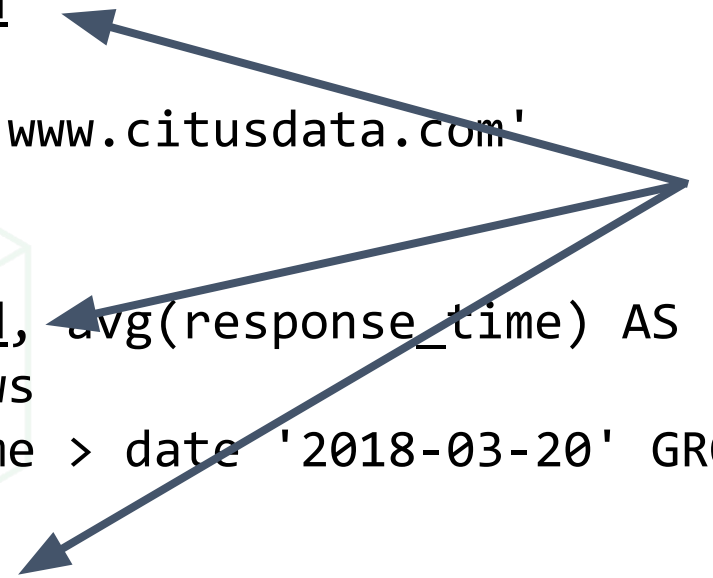
Technical observation:

A query that **joins all distributed tables by distribution column** with subqueries that **do not aggregate across distribution column** values can be distributed in a single round.

Pushdown planner

Determine whether distribution columns are equal using Postgres planner:

```
SELECT page_id, response_time
FROM (
  SELECT page_id
  FROM pages
  WHERE site = 'www.citusdata.com'
) p
JOIN (
  SELECT page_id, avg(response_time) AS response_time
  FROM page_views
  WHERE view_time > date '2018-03-20' GROUP BY page_id
) v
USING (page_id)
ORDER BY 2 DESC LIMIT 10;
```



Distribution column equality

Pushdown planner

Subquery results need to be partitionable by distribution column:

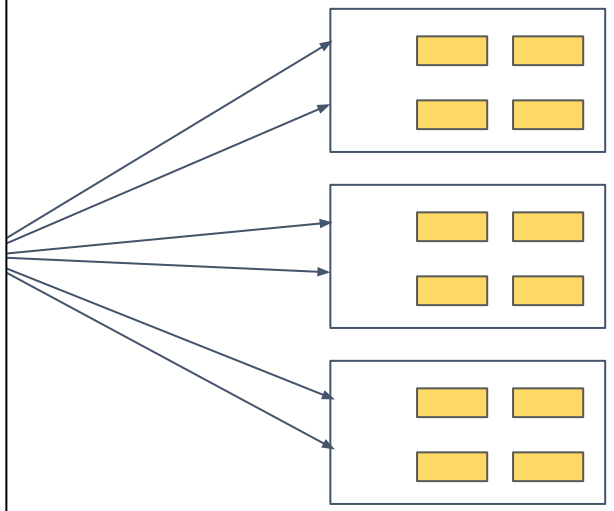
```
SELECT page_id, response_time
FROM (
  SELECT page_id
  FROM pages
  WHERE site = 'www.citusdata.com'
) p
JOIN (
  SELECT page_id, avg(response_time) AS response_time
  FROM page_views
  WHERE view_time > date '2018-03-20' GROUP BY page_id
) v
USING (page_id)
ORDER BY 2 DESC LIMIT 10;
```

No aggregation across distribution column values.

Pushdown planner

Subqueries can be executed across co-located shards in parallel:

```
SELECT page_id, response_time
FROM (
  SELECT page_id
  FROM pages_102670
  WHERE site = 'www.citusdata.com'
)
JOIN (
  SELECT page_id, avg(response_time) AS response_time
  FROM page_views_102008
  WHERE view_time > date '2018-03-20' GROUP BY page_id
)
USING (page_id)
ORDER BY 2 DESC LIMIT 10;
```



citusdata

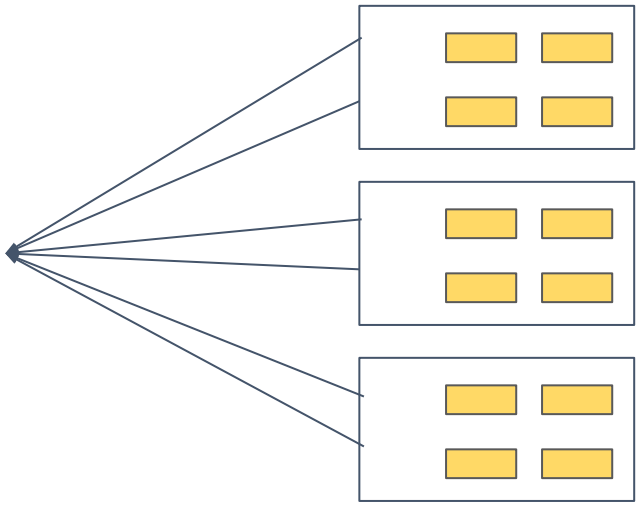
Merging query results

Merge the results on the coordinator:

```
SELECT page_id, response_time
FROM
```

Concatenated results of queries on shards

```
ORDER BY 2 DESC
LIMIT 10
```



Scaling Real-time Analytics Applications

Use case observation:

> Real-time analytics dashboards need sub-second response time, regardless of data size.

Single-round distributed queries are powerful, fast and scalable.

In practice:

- Maintain aggregation tables using parallel INSERT...SELECT
- Dashboard selects from the aggregation table

Complex subqueries

What about subqueries with merge steps?

```
SELECT
  product_name, count
FROM
  products
JOIN (
  SELECT product_id, count(*) FROM orders GROUP BY product_id
  ORDER BY 2 DESC LIMIT 10
) top10_products
USING (product_id)
ORDER BY count;
```

Recursive planning

Have a query you can’t solve? Call the Postgres planner!

Recursive planning

Technical observation:

> Subqueries and CTEs that cannot be pushed down can often be executed as distributed queries.

Pull-push execution:

- Recursively call `planner()` on the subquery
- During execution, stream results back into worker nodes
- Replace the subquery with a function call that acts as a reference table

Recursive planning

Separately plan CTEs and subqueries that violate pushdown rules:

```
SELECT
  product_name, count
FROM
  products
JOIN (
  SELECT product_id, count(*) FROM orders GROUP BY product_id
  ORDER BY 2 DESC LIMIT 10
) top10_products
USING (product_id)
ORDER BY count;
```

Recursive planning

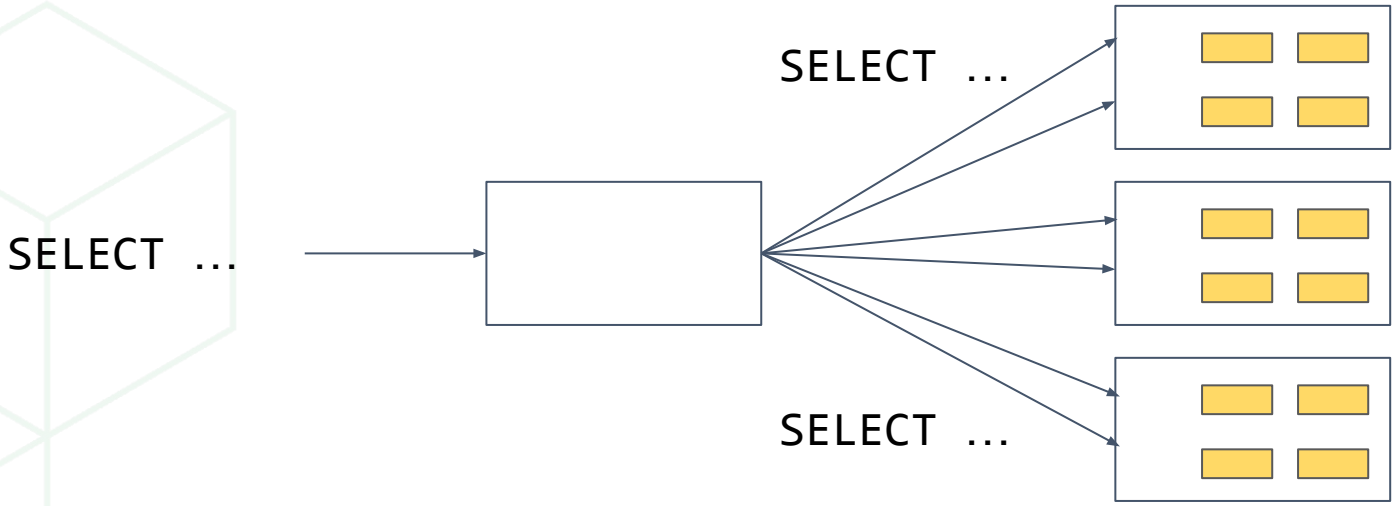
In the outer query, replace subquery with intermediate result, treated as reference table:

```
SELECT
  product_name, count
FROM
  products
JOIN (
  SELECT * FROM read_intermediate_result(...) AS r(product_id text, count int)
) top10_products
USING (product_id)
ORDER BY count;
```

Pull-push execution

Execute non-pushdownable subqueries separately:

```
SELECT product_id, count(*) FROM orders GROUP BY product_id ORDER BY 2 DESC
LIMIT 10
```

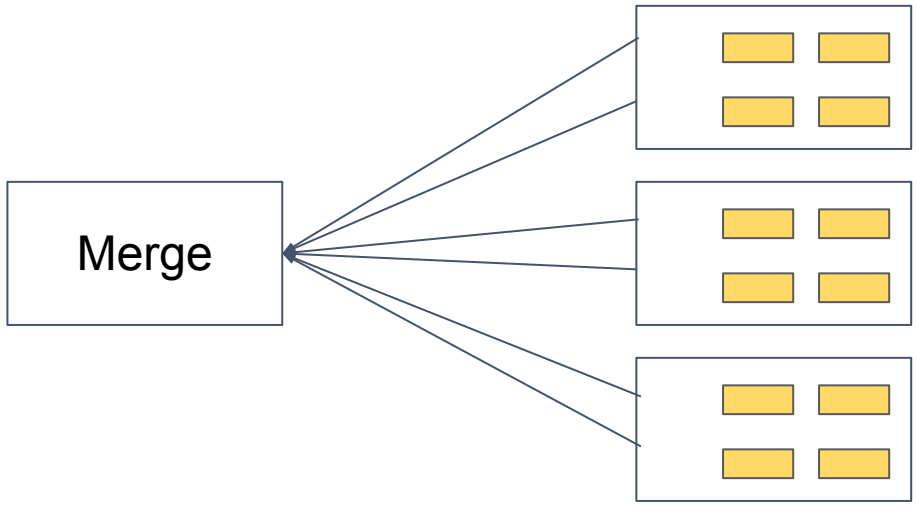


citusdata

Pull-push execution

Execute non-pushdownable subqueries separately:

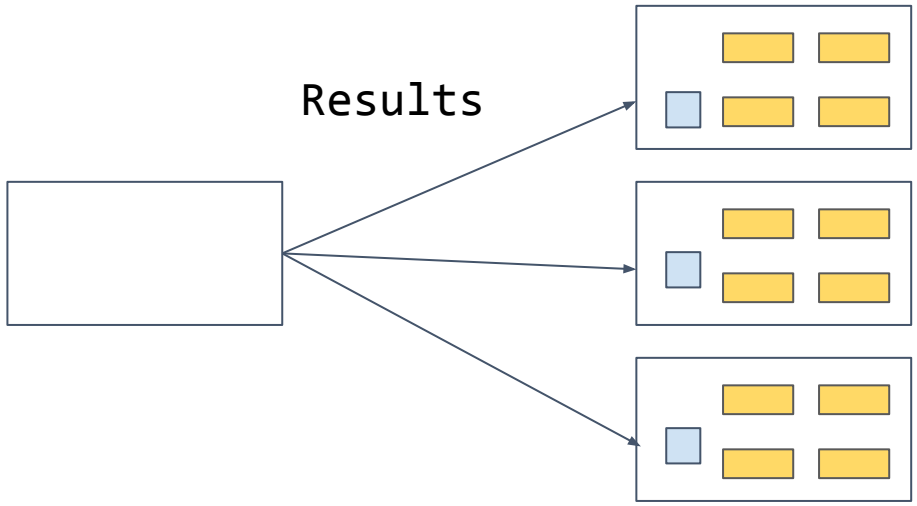
```
SELECT product_id, count(*) FROM orders GROUP BY product_id ORDER BY 2 DESC
LIMIT 10
```



Pull-push execution

Execute non-pushdownable subqueries separately:

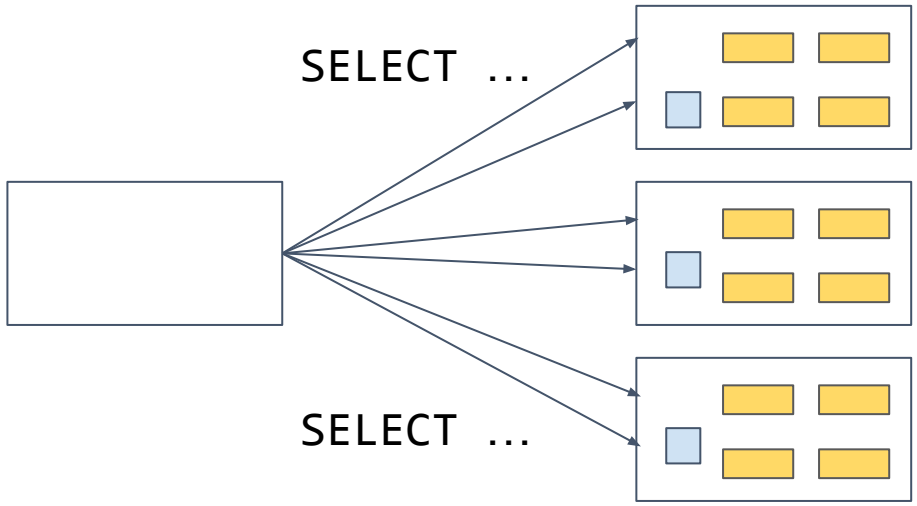
```
SELECT product_id, count(*) FROM orders GROUP BY product_id ORDER BY 2 DESC
LIMIT 10
```



Pull-push execution

Execute non-pushdownable subqueries separately:

```
SELECT product_name, count FROM products JOIN (SELECT * FROM
read_intermediate_result(...) …) …;
```

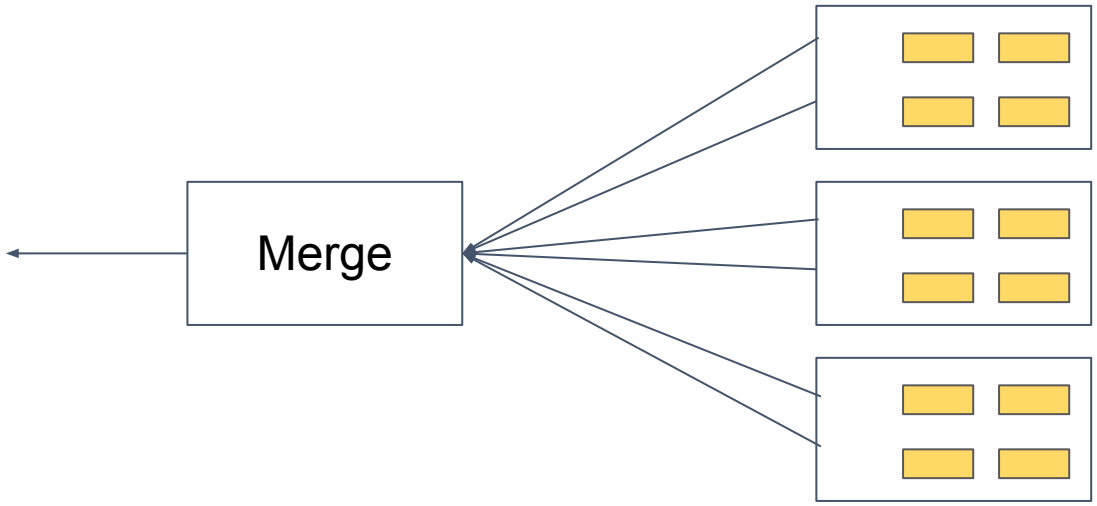


citusdata

Pull-push execution

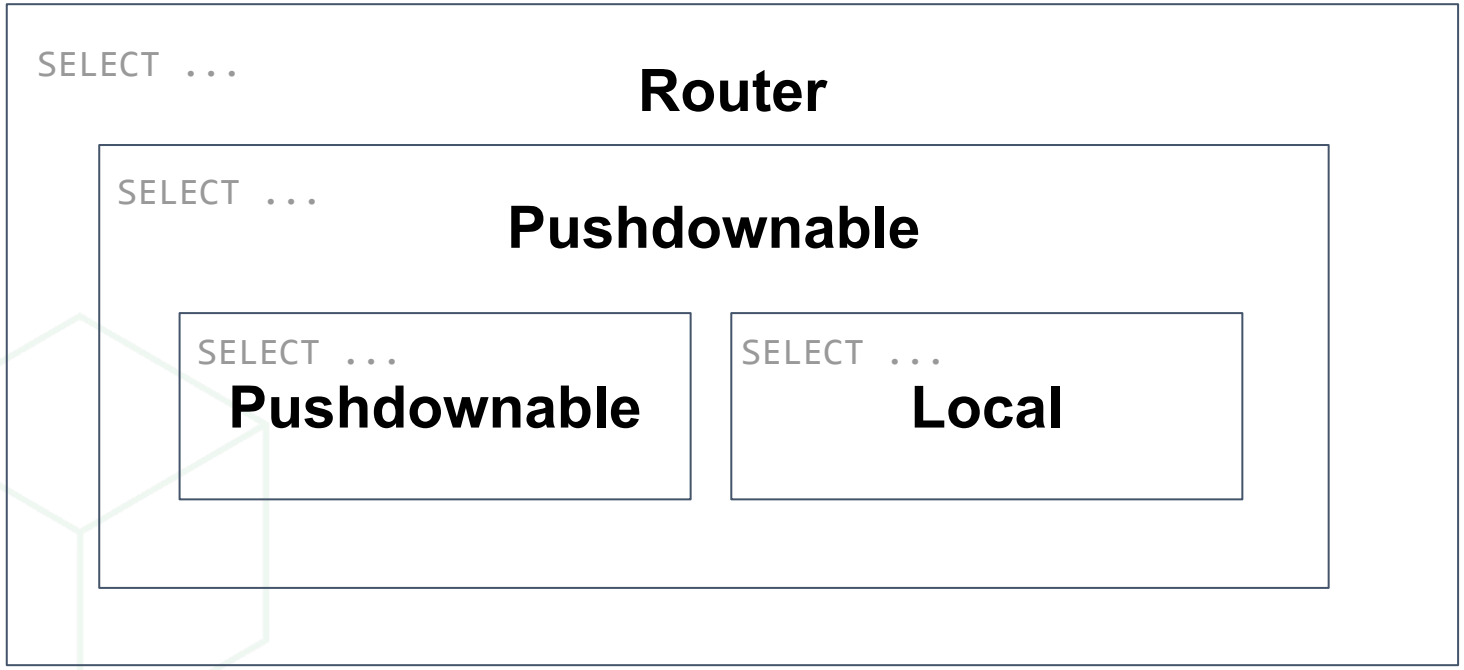
Execute non-pushdownable subqueries separately:

```
SELECT product_name, count FROM products JOIN (SELECT * FROM
read_intermediate_result(...) …) …;
```



Recursive planning

Different parts of a query can be handled by different planners.



citusdata

Joins between tables and intermediate results

Technical observation:

Intermediate results of CTEs and subqueries are treated as reference tables: can use any join column.

```
WITH
    distributed_query AS (...)
SELECT
    ...
    distributed_query JOIN distributed_table USING (any_column)
    ...
```

Joins between intermediate results

Technical observation:

Queries with only intermediate results (CTEs or subqueries) are router plannable: full SQL in a single round-trip.

```
WITH
    distributed_query_1 AS (...),
    distributed_query_2 AS (...)
SELECT
    ...
    distributed_query_1 ... distributed_query_2
    ...
```

Can use any SQL feature without further merge steps

Scaling Real-time Analytics Applications

Use case observation:

Real-time analytics applications want versatile distributed SQL support

Recursive planning provides nearly full, distributed SQL support in a small number of network round trips.

Logical planner

Handling non-co-located joins through relational algebra

Non-co-located joins

Business intelligence queries may join on non-distribution columns.

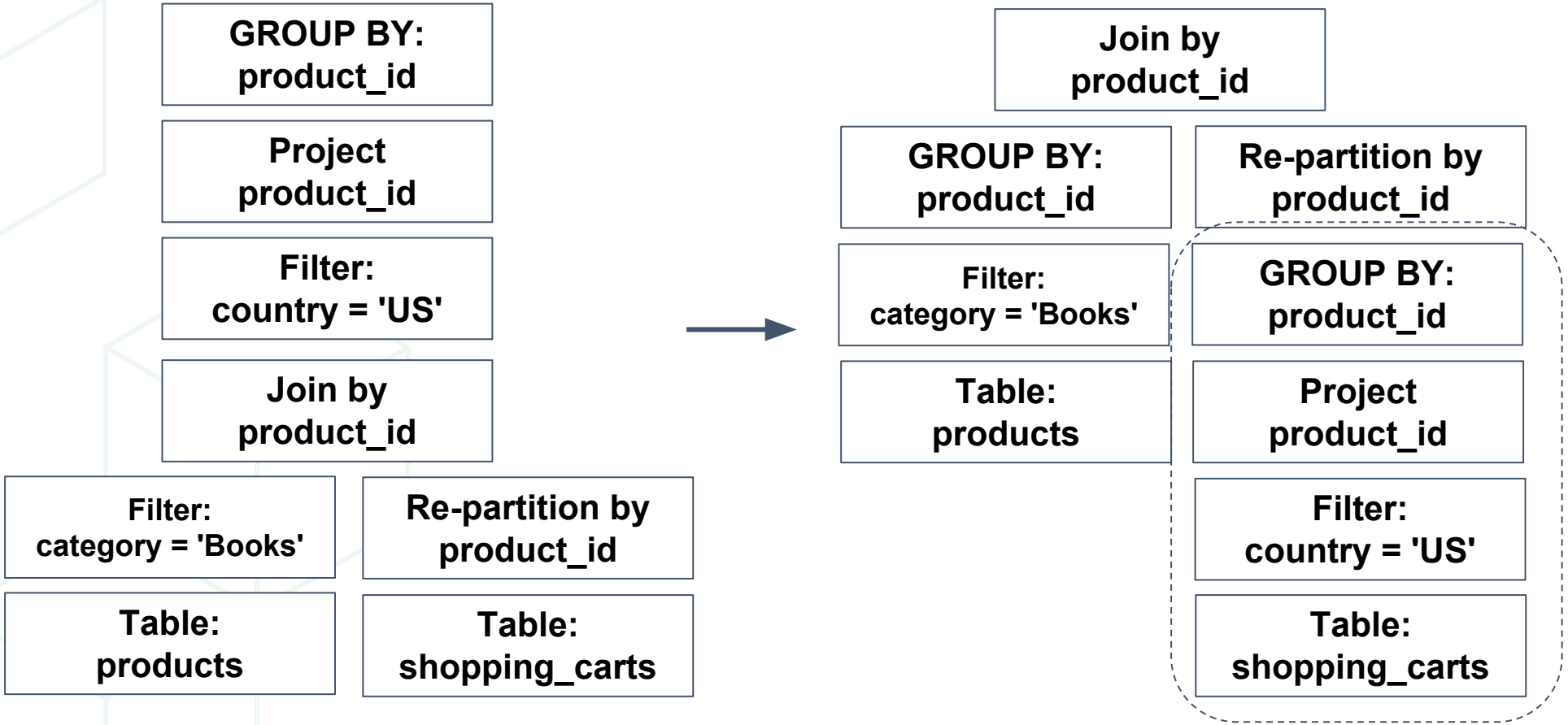
```
SELECT
  product_id, count(*)
FROM
  shopping_carts JOIN products USING (product_id)
WHERE
  shopping_carts.country = 'US' AND products.category = 'Books'
GROUP BY
  product_id;
```

Distributed by product_id

Distributed by customer_id for fast lookup of shopping cart

Distributed query optimisation

Apply operations that reduce data size before re-partitioning.

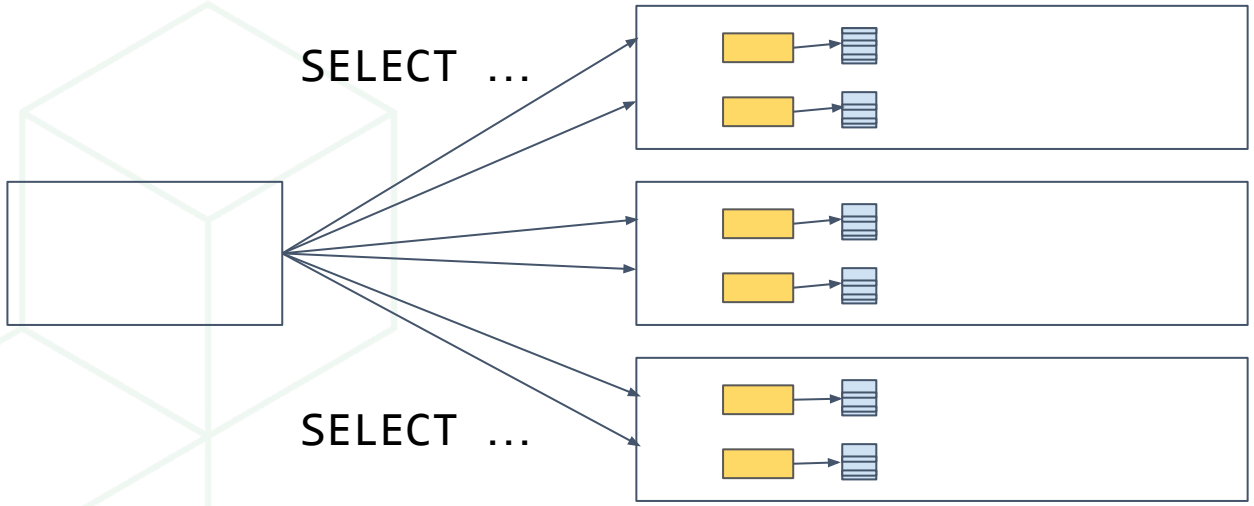


citusdata

Re-partitioning

Split query results into buckets based on `product_id`

```
SELECT partition_query_result($$
  SELECT product_id, count(*) FROM shopping_carts_1028 WHERE country = 'US'
  GROUP BY product_id
$$, 'product_id');
```

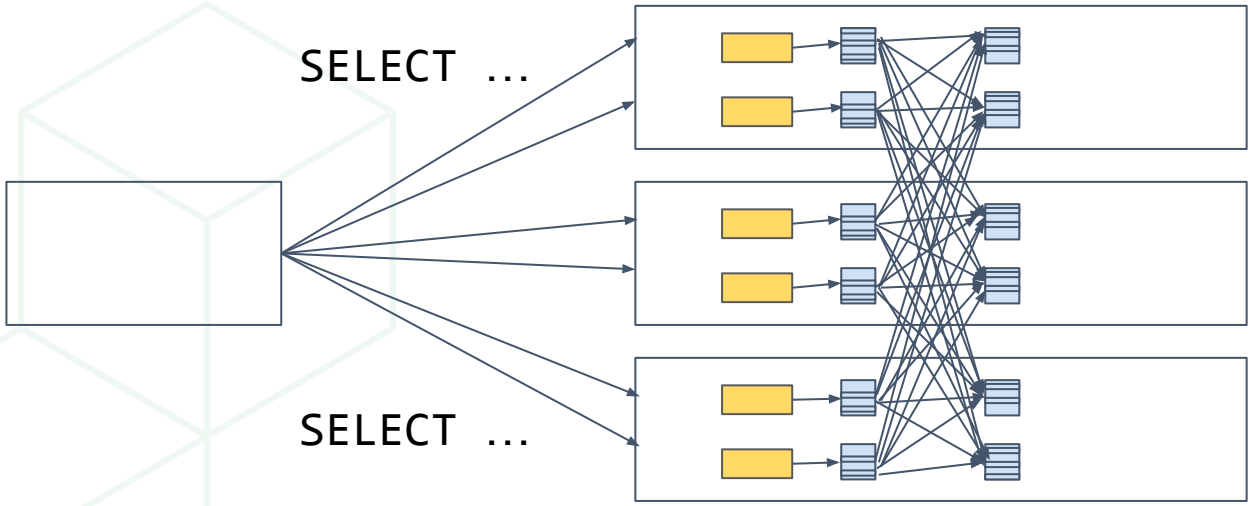


citusdata

Re-partitioning

Fetch `product_id` buckets to the matching `products` shards.

```
SELECT fetch_file(...);
```

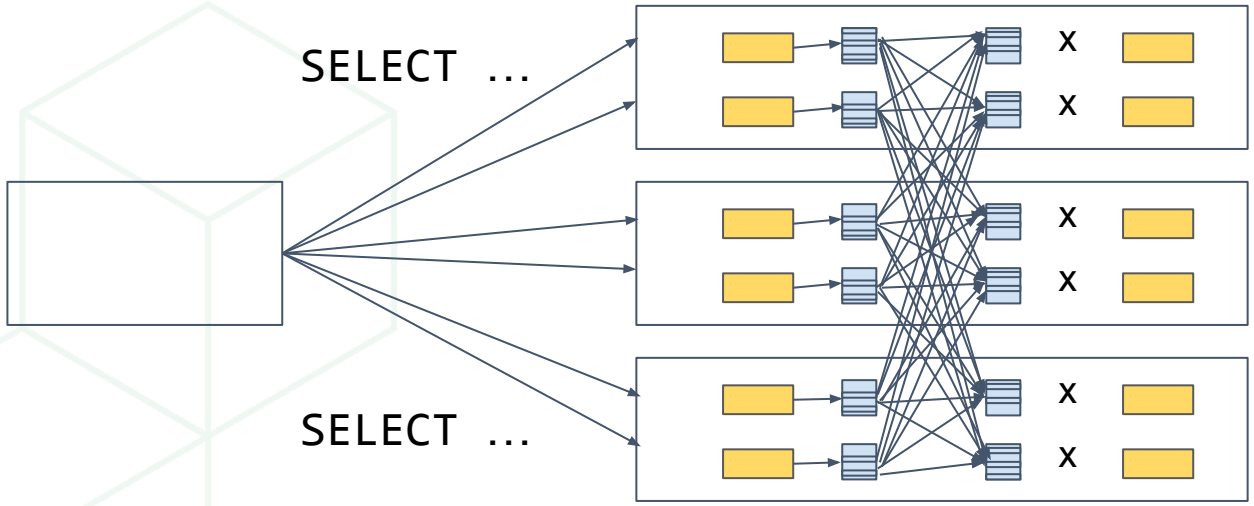


citusdata

Re-partitioning

Join merged buckets with products table

```
SELECT product_id, count FROM fragment_2138 JOIN products_102008 USING
(product_id) WHERE products.category = 'Books';
```



citusdata

Join order planning

Joins across multiple tables should avoid re-partitioning when unnecessary:

```
orders JOIN shopping_carts JOIN customers JOIN products
```

Bad join order:

orders x shopping_carts	→ re-partition by customer_id
join result x customers	→ re-partition by product_id
join result x products	→ query result

Good join order:

shopping_carts x customer	→ re-partition by product_id
join result x orders x products	→ query result

Evolution of distributed SQL

CitusDB:	Joins, aggregates, grouping, ordering, etc.	
Citus 5.0:	Outer joins, HAVING	(2016)
Citus 5.1:	COPY, EXPLAIN	
Citus 5.2:	Full SQL for router queries	
Citus 6.0:	Co-location, INSERT...SELECT	
Citus 6.1:	Reference tables	(2017)
Citus 6.2:	Subquery pushdown	
Citus 7.0:	Multi-row INSERT	
Citus 7.1:	Window functions, DISTINCT	
Citus 7.2:	CTEs, Subquery pull-push	(2018)
Citus 7.3:	Arbitrary subqueries	
Citus 7.4:	UPDATE/DELETE with subquery pushdown	

Thanks!

marco@citusdata.com

citusdata