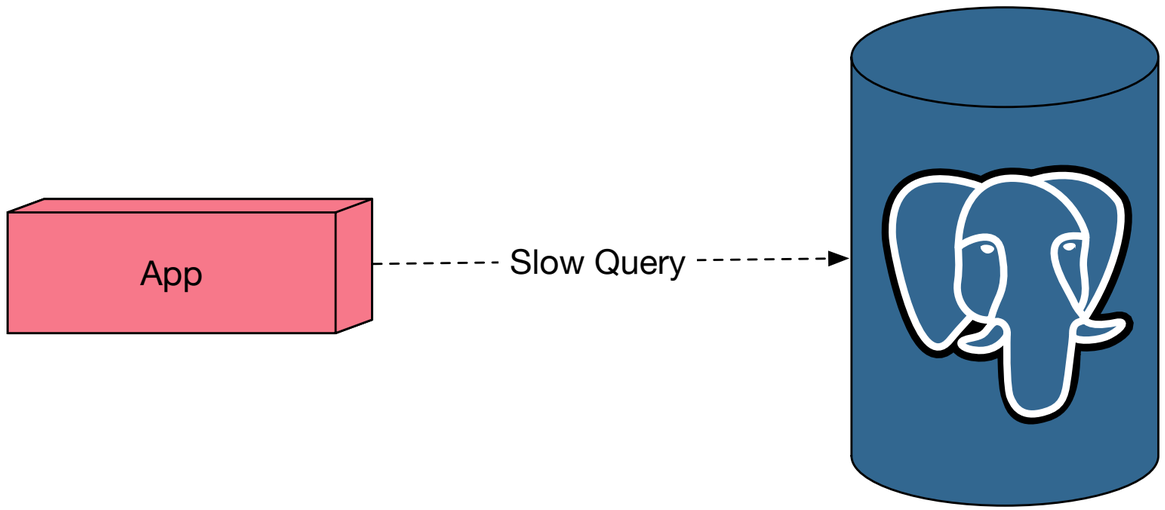
DYI Parallel Queries

Problem

- **Got:** Slow reporting query (~10s)
- **Want:** Good UX (< 1s on latest data)

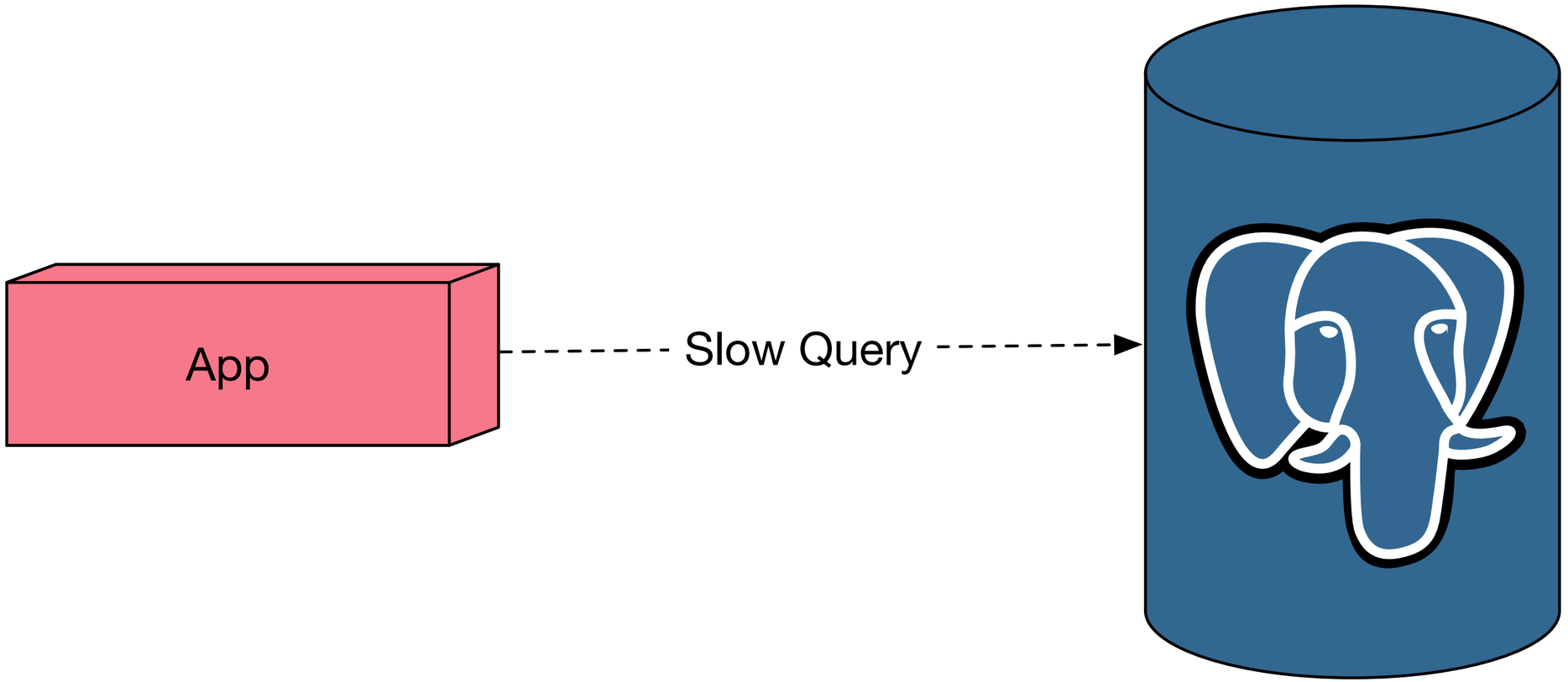


Solutions

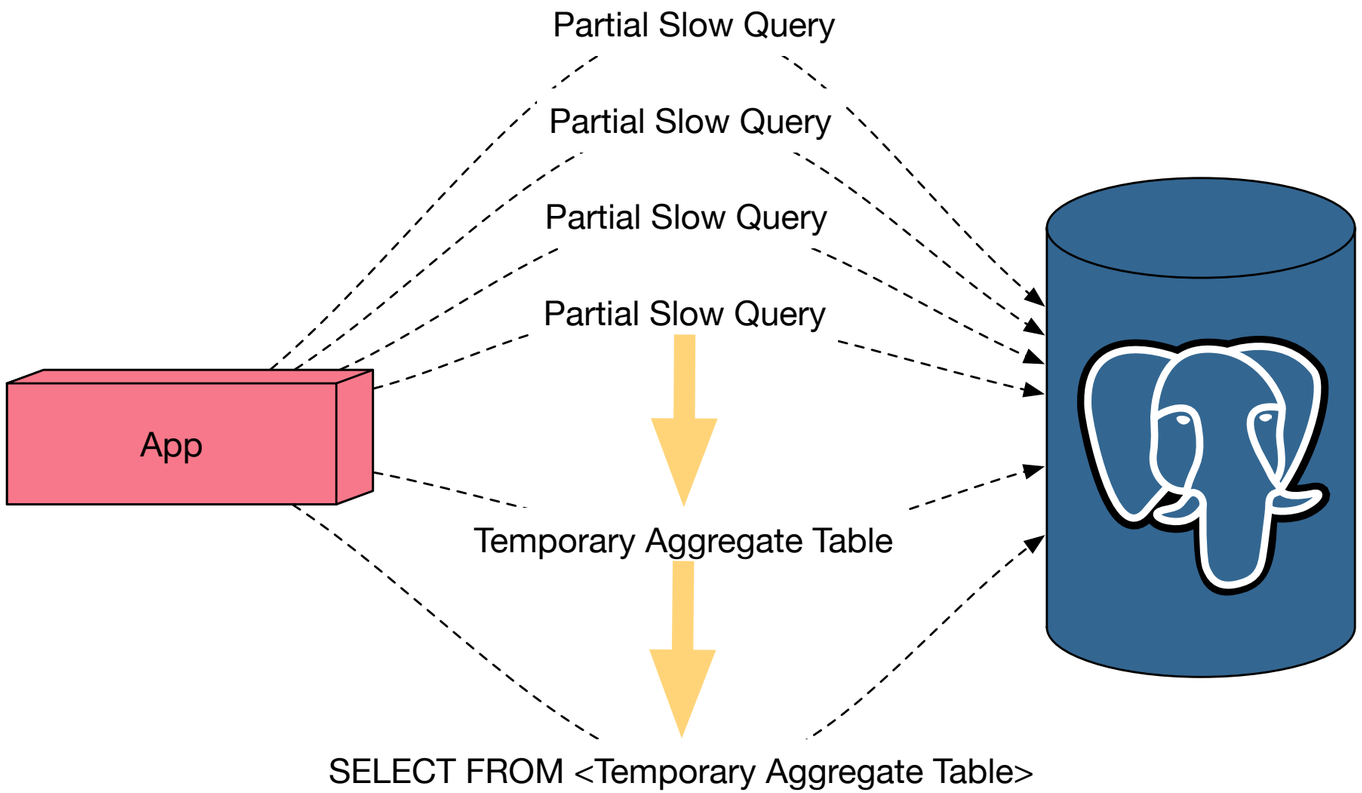
- **Query optimization:** Sometimes not enough. 😢
- **Cache computation with triggers:** Sometimes not possible. 💔
- **Postgres 9.6 Parallel Query:** Limited support. 🐤
- **CitusDB, PostgresXL, ...:** Big commitment. 💰

So what can I ~~hack together~~ build in a day?

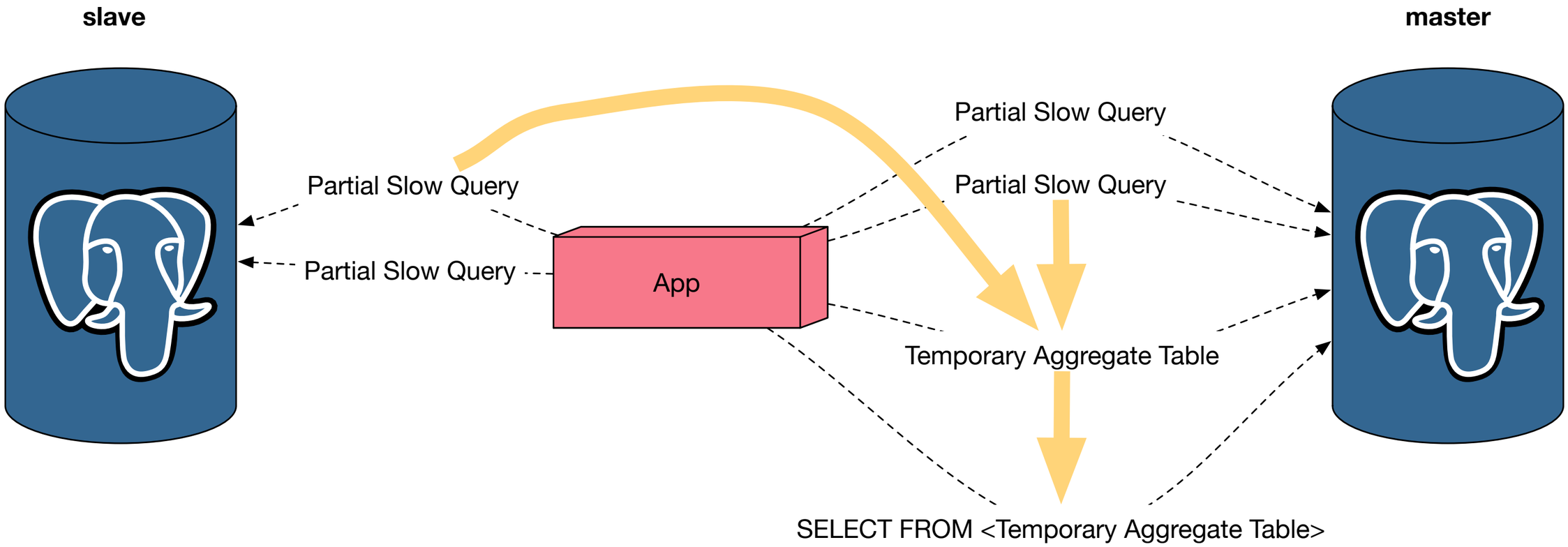
Before: One query



After: Multiple queries



After II: Multiple Machines!



Let's look at an example!

Slow example query

```
SELECT user_id
FROM (
    SELECT
        user_id,
        dough > lag(dough) OVER w AS more_dough,
        row_number() OVER w
    FROM transactions
    WHERE category_id = 1
    WINDOW w AS (PARTITION BY user_id ORDER BY id)
) s
WHERE row_number > 1
GROUP BY 1
HAVING
    bool_and(more_dough) = true
    AND count(*) >= 3;
```

Driven by a index scan node

```
GroupAggregate  (...) (...)
  Group Key: s.user_id
  Filter: (bool_and(s.more_dough) AND (count(*) >= 3))
  Rows Removed by Filter: 706445
  ->  Subquery Scan on s  (...) (...)
        Filter: (s.row_number > 1)
        Rows Removed by Filter: 917968
        ->  WindowAgg  (...) (...)
              ->  Sort  (...) (...)
                    Sort Key: transactions.user_id, transactions.id
                    Sort Method: quicksort  Memory: 204569kB
                    ->  Bitmap Heap Scan on transactions  (...) (...)
                          Recheck Cond: (category_id = 1)
                          Heap Blocks: exact=63695
                          ->  Bitmap Index Scan on transactions_partition_idx  (...) (...)
                                Index Cond: (category_id = 1)
Planning time: 0.564 ms
Execution time: 9302.230 ms
```

Index Adjustments

E.g. changing this index:

```
CREATE INDEX ON transactions(category_id);
```

to this index:

```
CREATE INDEX ON transactions(category_id, (user_id % 1000));
```

Allows us to quickly retrieve partition ranges from our index.

Query Adjustments

Instead of executing a single query with:

```
WHERE category_id = 1
```

We execute e.g. 4 concurrent queries like this:

```
WHERE category_id = 1 AND user_id % 1000 BETWEEN 0 AND 249
WHERE category_id = 1 AND user_id % 1000 BETWEEN 250 AND 499
WHERE category_id = 1 AND user_id % 1000 BETWEEN 500 AND 749
WHERE category_id = 1 AND user_id % 1000 BETWEEN 750 AND 999
```

Temporary table

The table needs to have the same columns that the query produces, e.g.:

```
CREATE TEMP TABLE results (
  user_id integer
) ON COMMIT DROP;
```

Then, as results from the concurrent query come in, insert them into the temporary table. Ideally using `COPY`.

Final query against temporary table

Can be as simple as:

```
SELECT * FROM results;
```

Or in more complex cases even perform some more aggregations.

Performance (Example Query, Mid 2012 4-Core MBP)

- Original query: ~8.3s
- DYI Parallel Query (c=8): ~2.5s

~3.3x faster 

Performance (Report Query, 24 Core IBM Server)

- Original query: ~9s
- DYI Parallel Query (c=20): ~0.9s

~10x faster 

Disclaimer: YMMV

Thanks!

- Slides & Code (Golang): github.com/felixge/pquery
- Twitter: @felixge
- E-Mail: pg@felixge.de