

A Roadmap to Continuous Materialized Views

PgConf US

April 2018

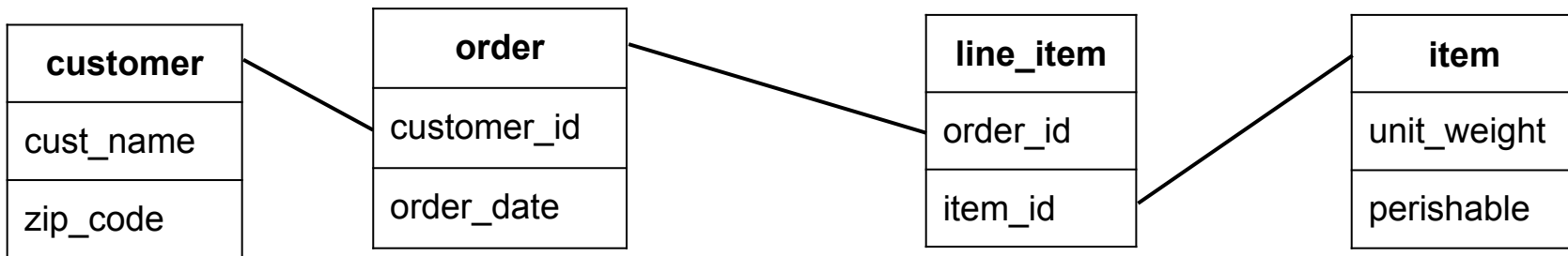
Corey Huinker

Corlogic

What's a View?

A stored query fragment

```
CREATE VIEW order_shipping_overview AS
SELECT    c.cust_name, c.zip_code,
          o.order_date,
          sum(i.unit_weight * l.qty) as tot_weight,
          bool_or(i.perishable) as use_refrigeration
FROM      order o
JOIN      customer c ON c.id = o.customer_id
JOIN      line_item l ON l.order_id = o.id
JOIN      item i ON i.id = l.item_id
GROUP BY c.cust_name, c.zip_code, o.order_date;
```



How do Views Work? Query Rewrite

Consider the following query

```
SELECT      sum(oso.tot_weight) AS total_weight_alphabet_city_this_week
FROM        order_shipping_overview oso
WHERE       oso.zip_code = '10009'
AND         oso.order_date >= CURRENT_DATE - 7;
```



Filters, aka "Quals"

How do Views Work? Query Rewrite

Rewrite "inlines" the view definition

```
SELECT      sum(oso.tot_weight) AS total_weight_alphabet_city_this_week
FROM        (
            SELECT      c.cust_name, c.zip_code, o.order_date,
                        sum(i.unit_weight * l.qty) as tot_weight,
                        bool_or(i.perishable) as use_refrigeration
            FROM        order o
            JOIN        customer c ON c.id = o.customer_id
            JOIN        line_item l ON l.order_id = o.id
            JOIN        item i ON i.id = l.item_id
            GROUP BY   c.cust_name, c.zip_code, o.order_date ) oso
WHERE       oso.zip_code = '10009'
AND         oso.order_date >= CURRENT_DATE - 7;
```

Original
view
definition

Same quals

How do Views Work? Query Rewrite

Predicates that follow the grain of the group by are "pushed down"
Planner eliminates un-referenced columns, avoids un-referenced aggregates.
Group by either becomes trivial or eliminated entirely

```
SELECT sum(oso.tot_weight) AS total_weight_alphabet_city_this_week
FROM (
  SELECT
    e.cust_name, e.zip_code, o.order_date,
    sum(i.unit_weight * l.qty) as tot_weight,
    bool_or(i.perishable) as use_refrigeration
  FROM
    order o
  JOIN
    customer c ON c.id = o.customer_id
  JOIN
    line_item l ON l.order_id = o.id
  JOIN
    item i ON i.id = l.item_id
  WHERE
    c.zip_code = '10009'
  AND
    o.order_date >= CURRENT_DATE - 7
  GROUP BY e.cust_name, e.zip_code, o.order_date) oso
WHERE
  oso.zip_code = '10009'
AND
  oso.order_date >= CURRENT_DATE - 7;
```

Eliminate unused columns

Push quals inside the view



How do Views Work? Query Rewrite

Aggregate of aggregate either becomes trivial or is eliminated, and you get:

```
SELECT      sum(i.unit_weight * l.qty) AS total_weight_alphabet_city_this_week
FROM        order o
JOIN        customer c ON c.id = o.customer_id
JOIN        line_item l ON l.order_id = o.id
JOIN        item i ON i.id = l.item_id
WHERE       c.zip_code = '10009'
AND         o.order_date >= CURRENT_DATE - 7;
```

Views +/-

Pros:

- Don't Repeat Yourself
- Saves time and reduces errors in ad-hoc queries.
- Take up no storage
- Can act as a security barrier, hiding sensitive columns, etc
- Can reflect an object that exists in another schema (search_path jail)

Cons:

- No benefits to the query plan over having written it yourself
- Depending on how you write it, could present an optimization fence

What's a Materialized View?

- An unlogged table (No WAL, No durability)
- That cannot be modified with INSERT / UPDATE / DELETE
- But is instead populated via a query result
 - which looks exactly like a view definition

```
CREATE MATERIALIZED VIEW order_shipping_overview AS
SELECT    c.cust_name, c.zip_code, o.order_date,
          sum(i.unit_weight * l.qty) as tot_weight,
          bool_or(i.perishable) as use_refrigeration
FROM      order o
JOIN      customer c ON c.id = o.customer_id
JOIN      line_item l ON l.order_id = o.id
JOIN      item i ON i.id = l.item_id
GROUP BY  c.cust_name, c.zip_code, o.order_date;
```

```
CREATE UNIQUE INDEX ON order_shipping_overview(cust_name, zip_code, order_date);
CREATE INDEX brrrrrr ON order_shipping_overview(zip_code, order_date)
WHERE use_refrigeration;
```


Benefits of Materialized Views

- Costly joins and filters have already been done
- Costly aggregates have already been done.
- Can be indexed
- Have collectable statistics
- A named object conveys meaning / intention to the user
- A place to hang more descriptive object comments

Limitations of Materialized Views

- Are not WAL logged
 - Must be regenerated in event of a crash
- Must be refreshed all-or-nothing
- refresh is expensive
 - if it weren't, then you didn't need it in the first place
- Data is instantly stale
- Either re-populate during object creation (causing a lag in your deployment)...
- ...or if created with WITH NO DATA, then they present a production mini-outage
- You have to know they exist
- You have to know they're populated
- You have to believe they're up to date (or close enough)
- Refreshing an mview takes it offline...
- ...unless you have a unique key and REFRESH MATERIALIZED VIEW CONCURRENTLY...
- ...in which case you create a new view...
- ...and compare that row-by-row with the old version, updating the old one
- ...which leads to mview table bloat
- ...and. is. slow.

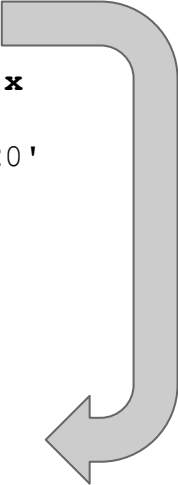
Oracle Feature: Query Rewrite

```
SELECT      o.order_date,  
            sum(i.unit_weight * l.qty) as tot_weight  
FROM        order o  
JOIN        customer c ON c.id = o.customer_id  
JOIN        line_item l ON l.order_id = o.id  
JOIN        item i ON i.id = l.item_id  
WHERE       c.zip_code LIKE '1000_'  
AND         o.order_date BETWEEN '2016-12-01' and '2016-12-20'  
GROUP BY   o.order_date  
ORDER BY   o.order_date;
```

Oracle Feature: Query Rewrite

```
SELECT      o. x.order_date,  
            sum(i.unit_weight * l.qty x.tot_weight) as tot_weight  
FROM        order o  
JOIN        customer c ON c.id = o.customer_id  
JOIN        line_item l ON l.order_id = o.id  
JOIN        item i ON i.id = l.item_id order_shipping_overview x  
WHERE       o. x.zip_code LIKE '1000_'  
AND         o. x.order_date BETWEEN '2016-12-01' and '2016-12-20'  
GROUP BY   o.order_date  
ORDER BY   o.order_date;
```

```
SELECT      order_date, sum(tot_weight) as tot_weight  
FROM        order_shipping_overview  
WHERE       zip_code LIKE '1000_'  
AND         order_date BETWEEN '2016-12-01' and '2016-12-20'  
GROUP BY   order_date  
ORDER BY   order_date;
```



Recognize
subplan that
matches
Materialized
View

Query Rewrite Benefits

- Can be utilized with no application-level coding changes
- Instant benefits to BI tools (Cognos, etc)
- Essentially an index on a query
- If Materialized View is offline, BI query continues to work, just at original slow rate
- Allows for tuning experimentation.

Query Rewrite Limitations

- Materialized view must be up to date with all child tables...
- ...or flagged as stale-ok (yuck)
- Infrastructure needed to detect stale-ness (ora_rowscn, on commit checks)
- Query must filter on columns exposed by the materialized view
- Query must aggregate to the same grain as the mview or a less granular derivative
- Parts of the query outside the scope of the materialized view must still be queried
- Any updates to source tables potentially mark the materialized view as stale...
- ...or trigger a potentially expensive recalculation of some or all of the materialized view

Roadmap

- Phase 1
 - Detect changes in source tables on DML Statement (Easy - triggers do that)
 - Reflect those changes in the materialized views (Medium/Hard)
 - Update views every statement
 - Create Continuous Materialized View Syntax
- Phase 2 (Hard)
 - Add the ability of to have Continuous Materialized Views Inherit from a regular view
 - ...or another Continuous Materialized View.
 - Change rewrite rules to allow for replacing the view with the best-choice CMV
- Phase 3 (Harder)
 - Add session setting to check for mview replacement in generic queries (Easy)
 - Recognize usage pattern matching existing and fresh materialized view (Quite Hard)
 - Rewrite portion of query to use base view (Medium/Hard) [*]

[*] see https://github.com/d-e-n-t-y/pg_fdw_mv_rewrite by John Dent

[*] see Kevin Kemper's FDW talk also given at PgConf US

Triggers

- These might not be real triggers, but it's helpful to think about them as such
- For every materialized view, create a custom trigger on each table which comprises that view
- Triggers must fire on INSERT and DELETE, but only need fire on UPDATE of columns that participate in the materialized view

Triggers Strategy: One Way Street

- Disallow any DELETE or UPDATE operations on source tables.
- New records are fed through the materialized view definition
- Resulting records are fed into materialized view as if on INSERT ON CONFLICT

```
INSERT INTO line_item (order_id, item_id, qty) VALUES (9987, 1212, 10);
```

	id	order_id	item_id	qty
NEW	11223344	9987	1212	10

Triggers Strategy: One Way Street

```
INSERT
INTO      order_summary_overview AS dest
SELECT   c.cust_name, c.zip_code, o.order_date,
         sum(i.unit_weight * NEW.qty),
         bool_or(i.perishable)
FROM      transition_table_new n
JOIN      order o ON o.order.id = n.order_id
JOIN      item i ON i.id = n.item_id
JOIN      customer c ON c.id = o.customer_id
GROUP BY c.cust_name, c.zip_code, o.order_date
ON CONFLICT (cust_name, zip_code, order_date) DO UPDATE
SET      dest.tot_weight = dest.tot_weight + EXCLUDED.tot_weight,
         dest.use_refrigeration = dest.use_refridgeration OR EXCLUDED.use_refrigeration;
```

line_item is missing,
replaced by values in
the new transition
table with a
user-defined name

Triggers Strategy: One Way Street

Strategy Employed By:

- pipelinedb
- projections in Vertica

Pros:

- Minimal I/O. Only looks at existing materialized view rows and accumulated "new" rows through the mview definition.
- Materialized view definition easily extractable from catalog
- Few other catalog changes required other than identifying which columns in table affect which materialized views
- Works with PER ROW triggers fairly easily as well (if doing an extension for ≤ 9.6)

Cons:

- Any UPDATES on referenced columns or DELETES invalidate the view requiring a full rebuild
- That's a show-stopper for many use-cases

Triggers Strategy: Reversion

- "Invert" aggregated values in OLD rows
- `INSERT ON CONFLICT UPDATE` of OLD and NEW rows to apply changes
- Keep tally of source rows of each table per destination rows to detect when a materialized view row can be deleted outright.

	id	zip_code	...	tot_weight
OLD	23456	10009		3.4
NEW	23456	90210		3.4

```
INSERT INTO order_summary_overview(...)
SELECT
FROM VALUES (23456, '10009', ..., -3.4), (2345, '902010', ..., 3.4) as t
JOIN order_summary_overview_src s ON ...
ON CONFLICT (customer_name, zip_code, order_date)
DO UPDATE SET tot_weight as excluded.tot_weight;
```

Triggers Strategy: Reversion

Pros:

- Can handle updates, deletes

Cons:

- Aggregate functions don't currently have a concept of "inverse", and it may not be possible for some aggregate types
- Un-doing rows may not be accomplishable in an auto-generated fashion
- Delete detection and row source tracking is complicated, potentially expensive.
- PER ROW triggers would have no concept of what other rows might aggregate to the same row, thus losing ability to reduce I/O by pre-aggregating results

Triggers Strategy: Scope and Recalc

- Accumulate set of keys or partial keys which were touched by the update
- Delete all rows from materialized view matching any of those keys or key patterns
- Run the Materialized view creation query, but filtered by that same set of keys/patterns
- Insert those rows into the Materialized view

```
UPDATE customer SET zip_code = '90210' WHERE id = 23456;
```

	id	zip_code	...	tot_weight
OLD	23456	10009		3.4
NEW	23456	90210		3.4

```
DELETE FROM order_summary_overview WHERE zip_code in ('10009', '90210');
```

```
INSERT INTO order_summary_overview SELECT * FROM order_summary_overview_src  
WHERE zip_code in ('10009', '90210');
```

Triggers Strategy: Scope and Recalc

Pros:

- Much simpler than Reversion
- Not much room for introducing errors

Cons:

- Materialized view definition might have optimization barriers making recalc worse
- Low cardinality values will affect a great many rows in materialized view - might be cheaper to just recalculate whole thing
- PER ROW triggers would need a work table to coalesce keys in to avoid doing multiple recalculations
- PER STATEMENT triggers might need this as well, if recalculation is deferred until COMMIT, otherwise just use the transition tables.

Modifications:

- Can modify this strategy to use One Way Street for pure INSERTS
- Can further modify to bypass existing data entirely if new data is known to not overlap

Syntax

```
CREATE OR REPLACE MATERIALIZED VIEW blah  
REFRESHED CONTINUOUSLY  
AS SELECT ...;
```

- Need to generate pseudo-triggers for all tables touched by the materialized view
- May want to consider excluding CTEs or other things that may make the pseudo-triggers complicated, at least as a first pass
- Will need to change application SQL to take advantages of mviews.
- Applications have to account for whether the mview is offline.

Alternate Syntax

```
CREATE VIEW base_view AS  
SELECT x.a, x.b, y.c, z.d  
FROM x  
JOIN y ON y.xid = x.id  
JOIN z on z.yid = y.id;
```

All continuous materialized views must be a simple filter and/or aggregation of a single view. When that view is referenced in a query. The optimizer can pick which CMV to use based on the quals in the query. Offline CMVs are excluded from consideration, and in the worst case, the base view is used.

```
CREATE MATERIALIZED VIEW cmv_filtered_1 REFRESHED CONTINUOUSLY  
AS SELECT * FROM base_view WHERE z.d IN ('red', 'green');
```

```
CREATE MATERIALIZED VIEW cmv_agg_1 REFRESHED CONTINUOUSLY  
AS SELECT a, b, c, COUNT(*) as num_z FROM base_view GROUP BY a, b, c;
```

```
CREATE MATERIALIZED VIEW cmv_agg_1 REFRESHED CONTINUOUSLY  
AS SELECT a, b, COUNT(*) as num_yz FROM base_view GROUP BY a, b;
```

```
CREATE MATERIALIZED VIEW cmv_agg_2 REFRESHED CONTINUOUSLY  
AS SELECT a, COUNT(*) as num_red FROM base_view WHERE z.d = 'red' GROUP BY a, b;
```

Phase 3: Transparent Usage

- Add session setting `enable_cmv_rewrite`. Default to false.
- Save parse tree of base view in an easy to retrieve format inside catalog
 - or a signature of the oids of all source relations and the quals
- Save replacement parse subtree in catalog too for each CMV
- Enable parser to recognize parse node trees which match existing mview trees
 - hard and potentially very inefficient
- Swap query parse subtree with mview subtree

Lessons From pg_fdw_mv_rewrite

On 2018-04-05, John Dent posted to pg-hackers announcing https://github.com/d-e-n-t-y/pg_fdw_mv_rewrite

Features:

- Tolerates stale materialized views, no new updating infrastructure
- Code currently re-parses the SQL-subset, but could use the quals provided by the FDW pushdown
- It essentially uses foreign tables as the stand-in for the base view I had envisioned
- uses the foreign server to group mviews into a set of possible stand-ins where I was envisioning a dependency link or inheritance of some kind
- works with existing postgresql, no core patches whatsoever
- incurs extra result set marshalling even though the tables are probably on the local server

Lessons:

- Validates the idea that application programmers would willingly re-code to use a view/table that would optionally exploit the right grain of dependent mview, reducing the need for transparent query rewrite
- shows that the planner logic is well worth the cost

Questions? Volunteers?