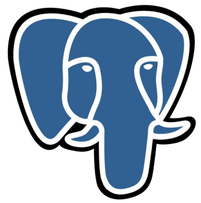
Caching long lasting queries

Laszlo Forro @ thinkproject.com



PostgreSQL

thinkproject

#constructionintelligence

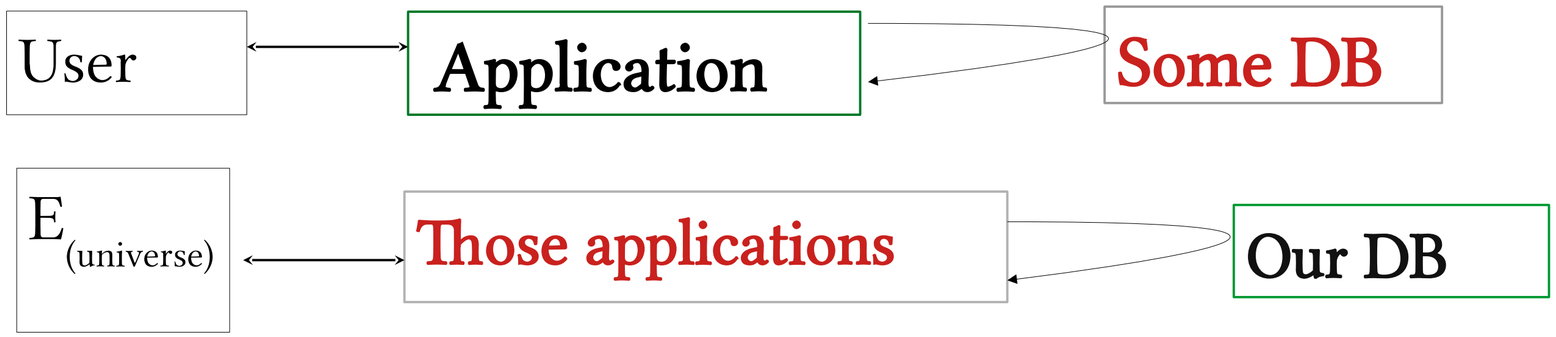
Serve *It* *Fresh* and if we are not *Fast* fast enough – Cache *It*

Rules of caching on the back end:

1) Do not cache
2) Cache is last resort, so try not to cache

3) Before caching consider if there is better data model or architecture

Side note



- the database and the user are separated
- the entities are how the application developers imagine it
- cognitive gap

Gavin Powell: Beginning Database Design, Wiley, 2006,
Ch.2.: Database Modelling in the Workspace

Serve *It* Fresh and Fast

It: what is the *use pattern* of the data we need to serve?

- Can we say that the most recently used?
- Can we say that the most frequently used?
- Or neither of those above?

Serve *It* Fresh and Fast

It : what is the *identity* of the data we serve *regarding the data source*?

- Can be identified simply
- Or a product of function like any aggregate (sum, count, aggregate average), max, min
- Or can’t be identified, eg. a result of a DISTINCT

Serve It *Fresh* and Fast

Fresh : what is the allowed *difference* of the data we serve and the *data source*?

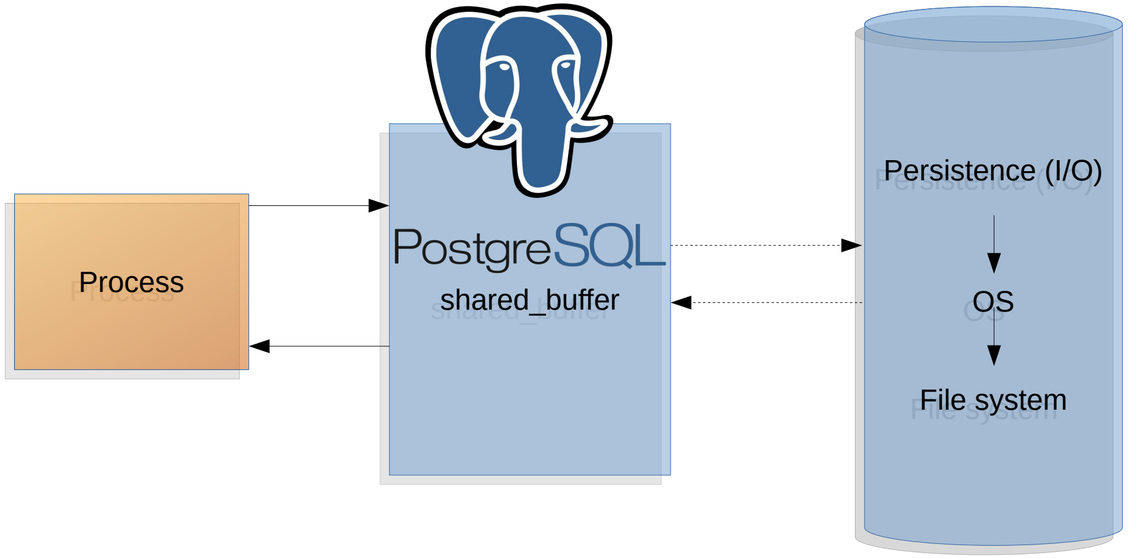
- Do we have to serve the latest and greatest for everyone
- Or we can let the user trigger an action for himself?
- What latency can we permit?

Serve It Fresh and *Fast*

Fast : which *system component* is identified as ‘slow’?

- Where should the optimization take place?
- Can a change in the architecture/data model result in better optimization ?
- What is the implication of Cache Invalidation requirements ?
  (change ratio, cache complexity)

Serve It From *Cache*



- Caching in shared_buffer
- Clock-sweep using access counter to release memory
- Managing inserts and updates
- Low level on blocks / tuples
- Most Frequently Used/ Most Recently Used
- It is about *all* access to the database, not only that we consider user relevant.

If we need cache, it is already not sufficient…

See the excellent chapter on PostgreSQL buffer management.
See also PostgreSQL memory components @severalnines: Architecture and Tuning of Memory in PostgreSQL Databases
See also the Cache replacement policies page on the Wikipedia.

Serve *It* From *Cache*

Incremental View Maintenance

A materialized views represent a result set of a query.

On change try to add only the change instead of replaying the query.

Assumption:
- the change can be propagated, because there are entities that can be identified as changed
- the change propagation is cheaper than replaying the query
  - factors: change ratio, algorithmic efficiency of view delegation

Serve *It* From *Cache*

Incremental View Maintenance

A *very* simple example:

1. Let's say a view is $V = A \bowtie_\theta B$
2. Change in A means a V diff table: $V_\delta = A_\delta \bowtie_\theta B$
3. Add the changes of V_δ to V using some Update Propagation Query

Serve *It* From *Cache*

Incremental View Maintenance

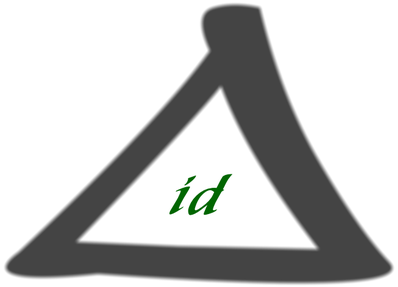
UPQ for INSERT can be $V \cup V_\delta$ or $V - V_\delta$ on DELETE,
$(V - V_\delta) \cup V_\delta$ on UPDATE *generally speaking.*

thinkproject

#constructionintelligence

Serve *It* From *Cache*

IVM – Self Maintainable View



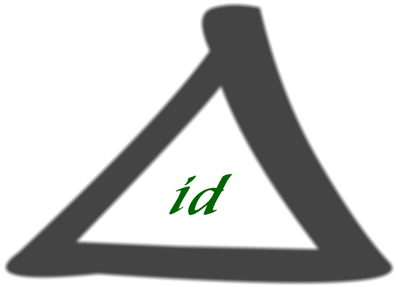
- there is a relation in the View and Update/Delete definition that makes possible to Update/Delete the View without replaying the whole query
- Inserts must be still evaluated

Serve *It* From *Cache*

IVM – Self Maintainable View

The *very* simple example

$V = A \bowtie_{\theta} B \longrightarrow V_{\delta} = A_{\delta} \bowtie_{\theta} B \longrightarrow$ Update V with V_{δ}



would look like

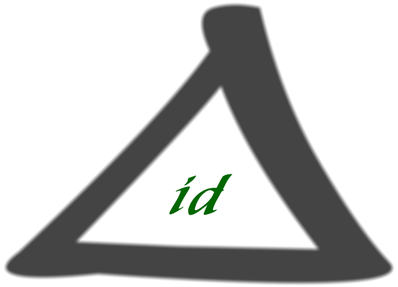
1. $V = A \bowtie_{a.id = b.aid} B$
2. Then when updating $A_{a1..aN}$, V can also be updated on the same ID references

Eg.: Katsis et al. : Utilizing IDs to AccelerateIncremental View Maintenance

Serve *It* From *Cache*

IVM – Self Maintainable View

A possible algorithm for PostgreSQL:



- on change of an entity in table A check
    1. if there is a dependent MV view on A and if yes,
    2. check if that view has a column corresponding to A.id
- make an UPDATE ... RETURNING A.id on A
- UPDATE the view with the NEW values by A.id

- might still need to parse a good amount of SQL
    - eg. A.id = B.aid (foreign key)
    - SELECT A.id as foo

Etc.

Serve *It* From *Cache*

IVM – Self Maintainable View

Yugo Nagata: Incremental View Maintenance,
presented at 2018 PgConf EU, Lissabon.
Implemented as an experimental PostgreSQL fork.

- The problem to identify entities is solved by using the OIDs
- Use a mapping table between the OIDs of the source table tuples and the View tuples for each tuple

Serve *It* From *Cache*

Incremental View Maintenance

- for a generic solution we need to identify an Update Propagation Query
- it grows more and more complex as more and more tables are involved
- needs efficient algorithm for DISTINCT, aggregate
  , min-max etc. functions

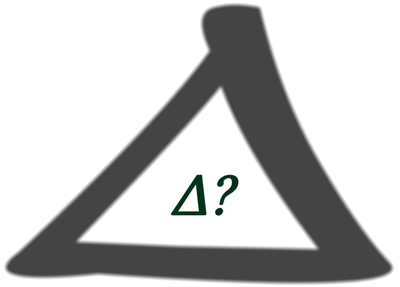
- how can it be efficiently implemented?

Database Query Optimization problem!

For the generic theory and the query optimizations see the thesis of Dimitra Vista:
Optimizing Incremental View Maintenance Expressions in Relational Databases, Toronto, 1996.

Serve *It* From *Cache*

Incremental View Maintenance



The approaches above

- create a Deltas table:
  that needs maintenance
- need some IDs to identify the changed entities:
  that is not always available for the whole View Definition
- apply the changes via some automated method
- track parallel changes…

Leads to a complex query optimization problem.

Serve *It* From *Cache*

A Logical Decoding Based Solution

- listener on logical decoding socket filtering the stream
- change distributor forwards the data to workers (domain filtering and semantics)
- the worker output is the cached data structure

Blagoj Atanasovski: Use Logical Decoding to build your own application cache, Presented at PgConf EU 2018, Lissabon

Serve *It* From *Cache*

A Logical Decoding Based Solution

Advantages:
- Consistency and invalidation is trivial
- No complex update query problem
- Application developer friendly

Blagoj Atanasovski: Use Logical Decoding to build your own application cache, Presented at PgConf EU 2018, Lissabon

Serve *It* From *Cache*

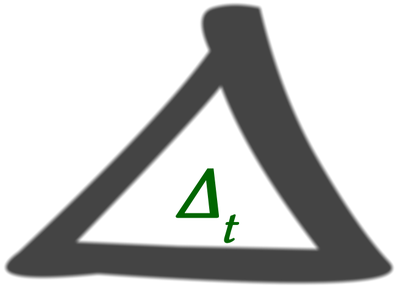
Our scenario:

- dynamically generated complex queries
- the identity of the entities depend on the query and not DB relatios
- the use case does not define hot data
- the only correlation is: the bigger a result set the more INSERTS
- no DELETE
- must be always *Fresh*
- max 1 sec user wait is ok.

Serve *It* From *Cache*

Incremental View Maintenance

A different angle: a change TS is always incremental.
Let A have an attribute, *t*, that acts as a 'last_changed' TS.



T_0: Let's say a view is $V = A \bowtie_\theta B$

Save T_0 as metadata for the View update TS.

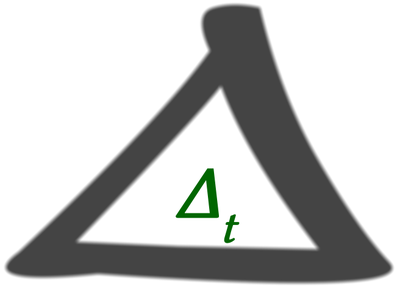
T_1: Update A as usual, but the $t = T_1$ for each element of A_δ

T_1: Update V: $V \leftarrow (\sigma_{(t > T0)} A) \bowtie_\theta B$, using T_0 from the View metadata table.

Save T_1 as metadata for the View update TS.

Serve *It* From *Cache*

Incremental View Maintenance



To identify an entity to Update, a Unique key is needed on the V table.

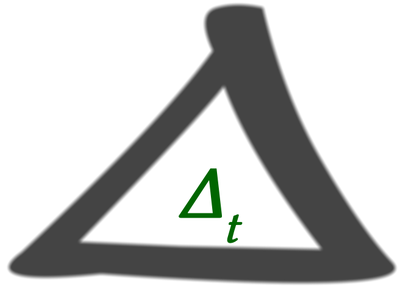
INSERT – UPDATE implementation: UPSERT.
If there is a key violation, it means that the entity already exists in the V table,
so it needs UPDATE.

What the client must provide:

- some way to recognize the condition for TS check in the V query
- the Unique key for the V table

Serve *It* From *Cache*

query_cache : non self maintaining view with deferred update

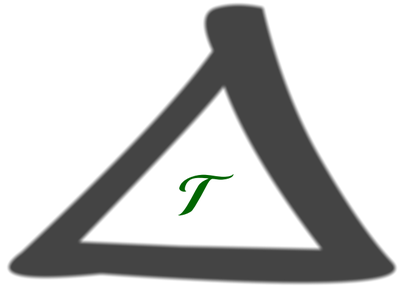


What we have not solved / did not want to solve
/ avoided to solve / couldn't solve / did not even think of to solve:

- a generic solution for the Update Propagation Query, instead we let the Query Optimizer of the Database to do the job.
- heruistics: the user must provide the Unique key.
- DELETION: it must be implemented as a change.
- optimization for aggregates
- ...

Serve *It* From *Cache*

query_cache : non self maintaining view with deferred update



TODOs:

- implement DELETE
- implement AGGREGATE, MIN/MAX, COUNT, AVG etc. cache
- rethink of the API
- some automatism (eg. if outer query has DISTINCT that can act as Unique key)
- possibly make it more generic

Thank you !

PostgreSQL

thinkproject

query_cache@thinkproject github