PosDB — a distributed column-store with late materialization

George Chernishev

Saint-Petersburg State University,
National Research University Higher School of Economics

chernishev@gmail.com

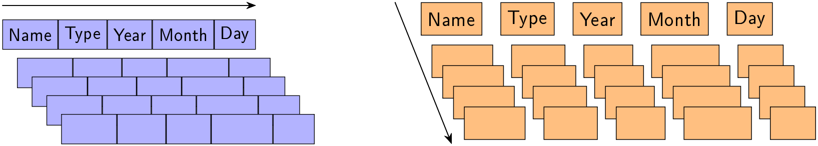
December 2019

Scope

- Column-oriented query processing
- Late materialization overview
- Existing systems and PosDB Motivation
- Query plans in PosDB
- Data acquisition: access methods and data readers
- PosDB as a research platform
- Conclusion: team, state, future plans

Column-store: an idea

- Data is stored in columns, not in rows:



- Only requested columns is read;
- Efficient use of hardware, e.g. vectorization.

Column-stores:

+ Excel at handling read-only queries, useful for handling OLAP;
+ Also popular in recently emerged HTAP applications;
– Updates are costly, HDD-based OLTP suffers.

Columnar approach: contemporary view

Nowadays many industrial systems call themselves “columnar”.

They treat column-orientation as storage level-only:

- processing is usually organized as follows: “read, decompress data, construct tuples, continue to work as usual”;
- allows to read only requested columns;
- efficient column-oriented compression.

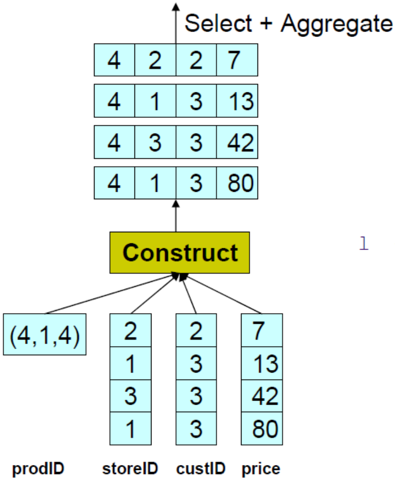
Columnar approach: founders vision

However, founders proposed not only column-oriented data ***storage***, but also column-oriented data ***processing***:

- query plans allow operators exchange not only data, but also positions;
- an option to select tuple reconstruction time: transition from positions to records
  - early materialization
  - late materialization
- operating directly on compressed data.

$\longrightarrow$ novel operators, novel query plans.

Query execution: EM example



QUERY:

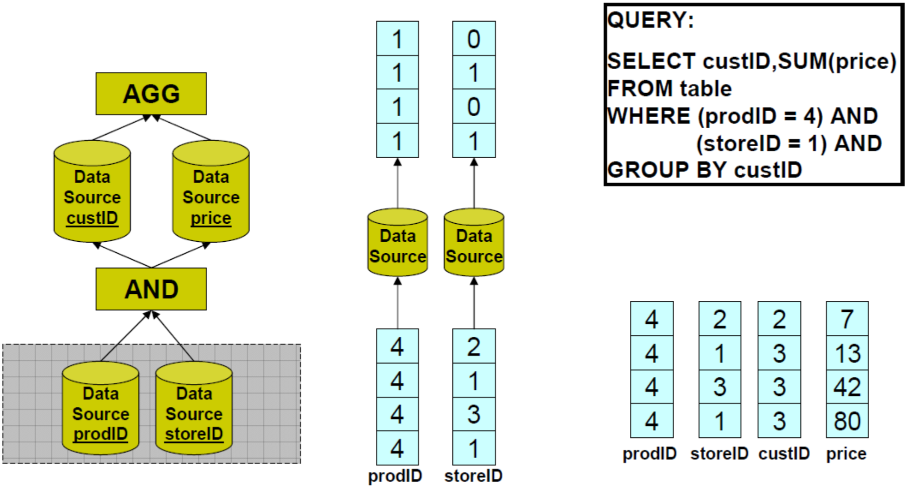
```
SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
      (storeID = 1) AND
GROUP BY custID
```

- **Solution 1: Create rows first (EM). But:**
  - **Need to construct ALL tuples**
  - **Need to decompress data**
  - **Poor memory bandwidth utilization**

[1]

[1] Image is taken from D. Abadi, P. Boncz, and S. Harizopoulos. Column-oriented database systems. Proc. VLDB Endow. 2, 2 (August 2009)

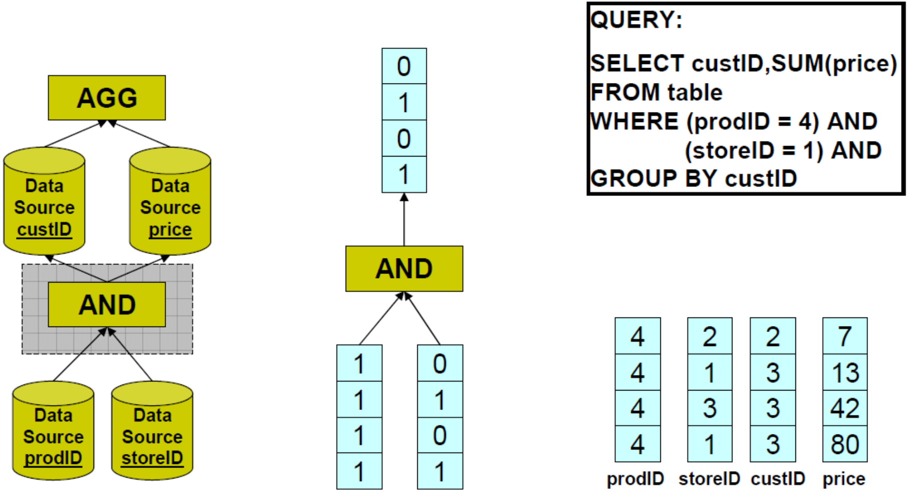
Query execution: LM example I



[2]

[2] Image is taken from D. Abadi, P. Boncz, and S. Harizopoulos. Column-oriented database systems. Proc. VLDB Endow. 2, 2 (August 2009).

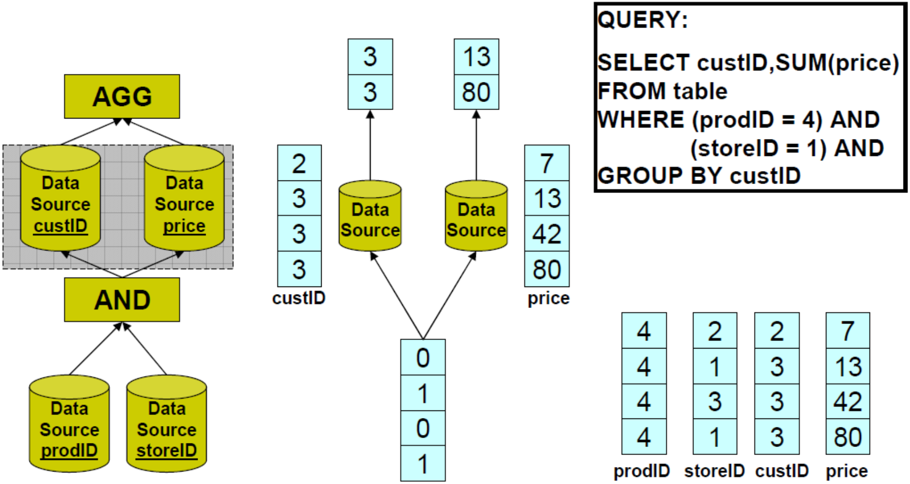
Query execution: LM example II



[3]

[3] Image is taken from D. Abadi, P. Boncz, and S. Harizopoulos. Column-oriented database systems. Proc. VLDB

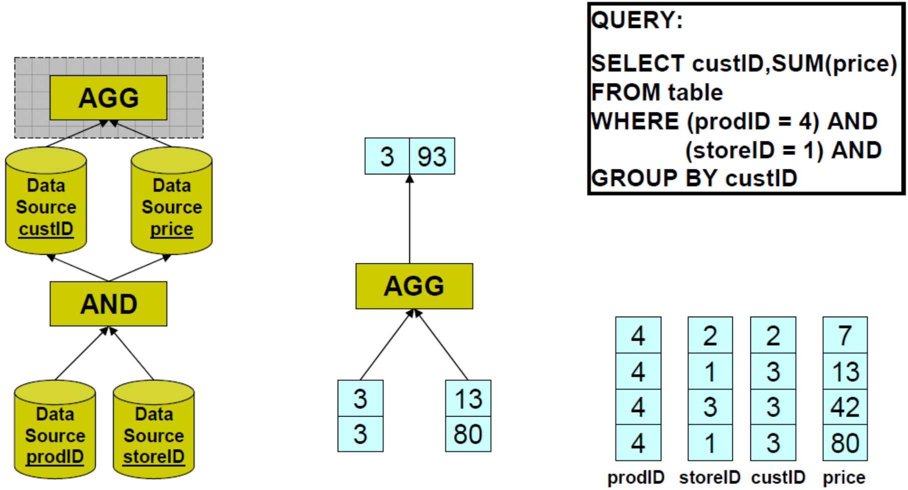
Query execution: LM example III



[4]

[4] Image is taken from D. Abadi, P. Boncz, and S. Harizopoulos. Column-oriented database systems. Proc. VLDB Endow., 2, 2 (August 2009).

Query execution: LM example IV



[5]

[5] Image is taken from D. Abadi, P. Boncz, and S. Harizopoulos. Column-oriented database systems. Proc. VLDB

Existing column-stores

System	Centralized or distributed	Sources Available?	Disk-based or in-memory?
C-Store MonetDB	Centralized	Yes	Disk-based in-memory
Vertica Vector	Distributed	No	Disk-based Disk-based with in-memory tuning
ScaMMDB DCODE	Distributed	No	in-memory
ClickHouse	Distributed	Yes	Disk-based

...and a lot of other commercial (mostly centralized) systems.

Motivation behind PosDB

- Studies related to many problems were not finished: aggregation, subqueries, ...
- Distributed processing for column-stores is not studied, ScaMMDB and DCODE projects are abandoned

$$\Downarrow$$

Need a prototype of a new distributed column-store

- Since ***disk-based*** distributed column-stores were not studied at all, we decided to concentrate on this type.

PosDB

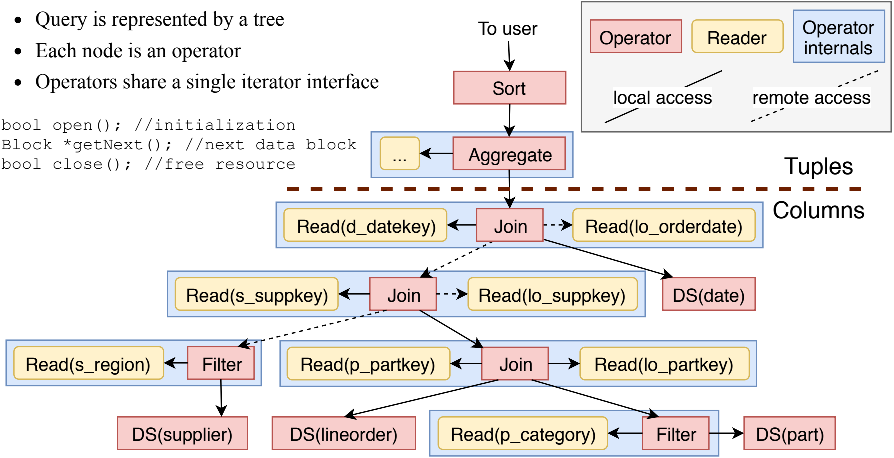
A distributed disk-based column-store for research purposes:

- Relies on Volcano block-based iterator model.
- Columnar: operators exchange not only data, but also positions (PosDB).
- Disk-based: data $>>$ main memory.
- Distributed: has send & receive operators. Not mediator-based, but "true" distribution of data and queries.
- Parallel: any operator sub-tree can be executed in a separate thread.
- Currently relies on late materialization, however, recently elements of hybrid materialization were added.

Query plans, Volcano model, Late materialization

- Query is represented by a tree
- Each node is an operator
- Operators share a single iterator interface

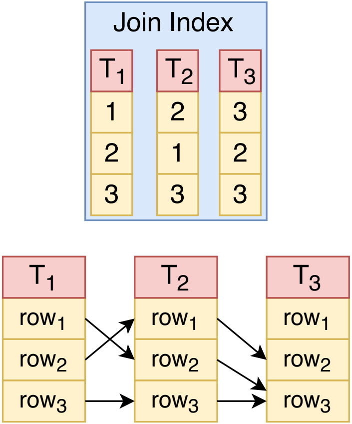
```
bool open(); //initialization
Block *getNext(); //next data block
bool close(); //free resource
```



Join index

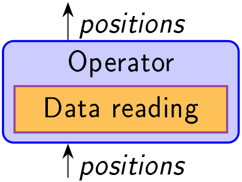
Query may contain joins, therefore a special data block is required:

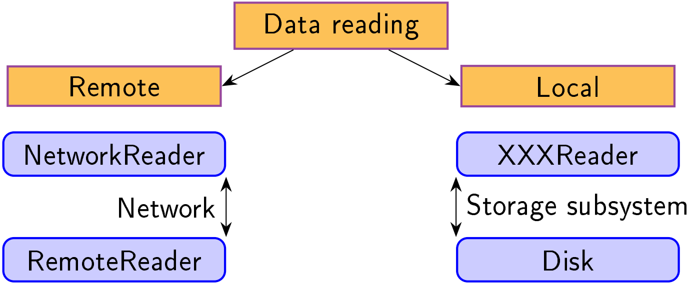
- Use classic data structure;
- Position lists, one per table;
- Two tables: a map of positions of T_1 into positions of T_2;
- N tables: a map of positions of T_1, T_2, T_{N-1} into positions of T_N



Acquiring data and positions

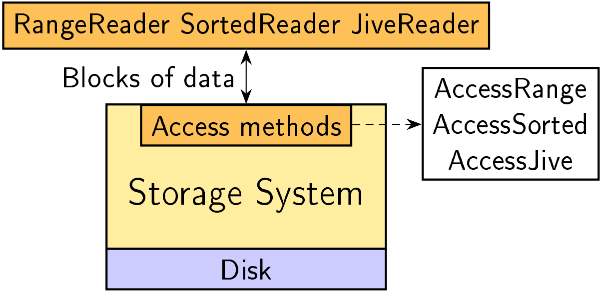
1. Initial `JoinIndex` acquisition happens in leaves of a tree via `DataSource` operator
2. All necessary data is read inside operators





Data access

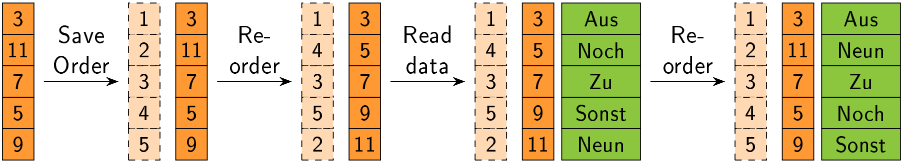
- **Reader**: global strategy to get data for a position *stream*
- **Access method**: *local* access to data for a position *block*



Access methods

For a position block access data *efficiently*:

1. contiguous position range (`AccessRange`): `for` loop, sequential pages
2. sparse sorted list of positions (`AccessSorted`): similar to `for` loop, page number always increasing (one-direction file read)
3. sparse unordered list of positions (`AccessJive`) – ?



Mixed stream of positions

What if stream of positions is local but heterogeneous?

- Select an optimal access method for each individual block (dynamically)
- `AccessJive` worse `AccessSorted` worse `AccessRange`

```
class MixedReader {
   AccessRange   range;
   AccessSorted  sorted;
   AccessJive    jive;
   AccessMethod *current; // points to  range, sorted or jive
}
```

Current State

(+)

- Supports Star Schema Benchmark;
- Distribution and parallelism on a plan level:
  - Both inter- and intra- query parallelism;
  - Both data fragmentation and replication;
  - Distributed operators;
- Lots of positional- and value- operators;
- Recently implemented buffer manager;

(−)

- Up until recently we concentrated on query executor; no rewriter and query optimizer, statistics subsystem;
- (Mostly) No compression;
- No vectorized primitives and expression compilation;
- (Mostly) Late materialization;

PosDB as a research platform

Currently, PosDB is used as a research platform for studying various query processing aspects, e.g.:

- Distributed operators:
  - distributed join
  - distributed aggregation
- Aggregation:
  - on-the-fly filtering of groups
  - window functions
- Caching of intermediates
- ...

Distributed operations

General idea

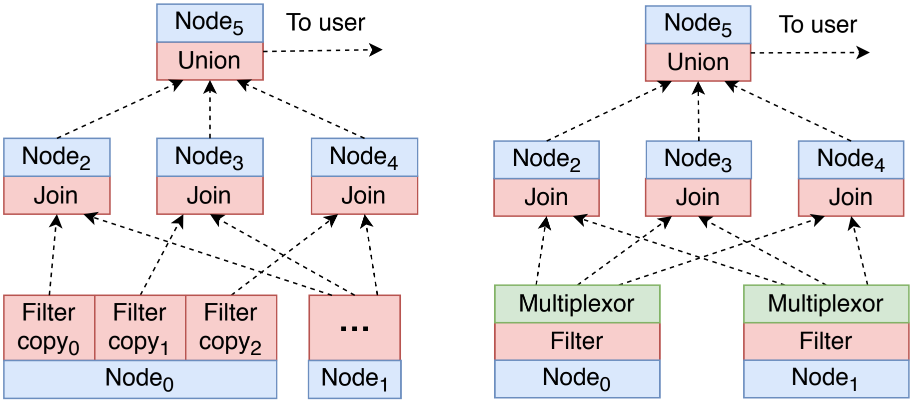
1. Appropriately distribute data
2. Compute local intermediates
3. Merge them to obtain total result

Problems

- Minimize network I/O
- Avoid DAG query model
- Provide flexible and efficient distributed data model

1. Distributed join: reshuffle, local join, union
2. Distributed aggregation: decompose, local preaggregate, combine

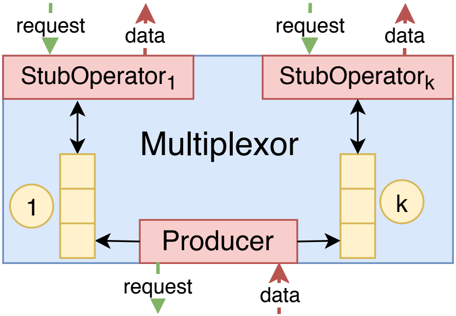
Distributed join: DAG?



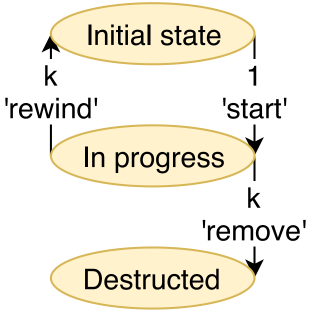
(a) Duplicate approach

(b) Multiplexor approach

Multiplexor: 1 to k

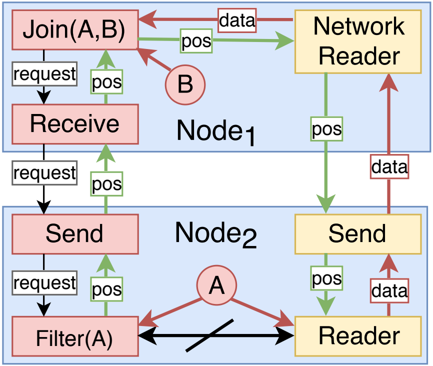


(a) Module architecture

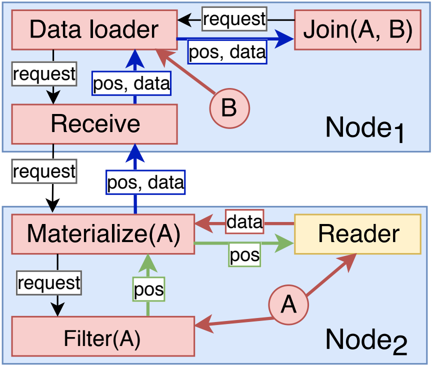


(b) State diagram

Hybrid materialization: optimize network I/O

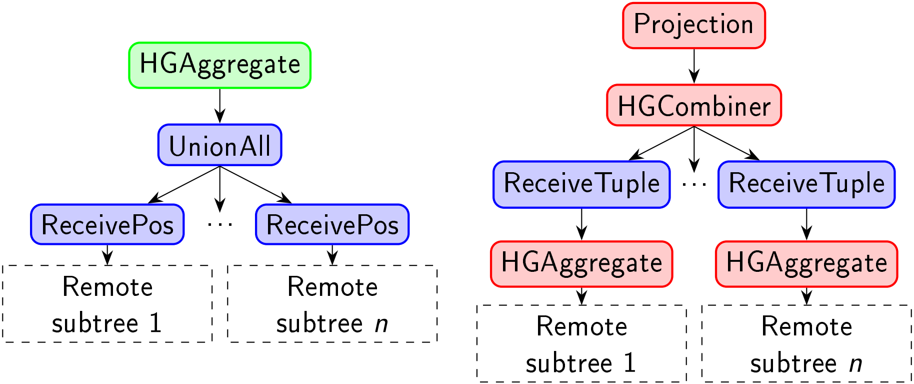


(a) Late materialization



(b) Hybrid materialization

Distributed aggregation: earlier and now



Decomposable aggregation functions

Definition (from "Efficient Evaluation of Aggregates on Bulk Types")

A scalar aggregation function $f : bulk(\tau) \rightarrow \mathcal{N}$ is called *decomposable*, if there exist functions

$$\begin{aligned} \alpha : bulk(\tau) &\rightarrow \mathcal{N}' \\ \beta : \mathcal{N}', \mathcal{N}' &\rightarrow \mathcal{N}' \\ \gamma : \mathcal{N}' &\rightarrow \mathcal{N}, \end{aligned}$$

with

$$f(Z) = \gamma(\beta(\alpha(X), \alpha(Y)))$$

for all X, Y, and Z with $Z = X \cup Y$.

From algorithmic perspective α, β, and γ are phases of processing:

- α — preaggreggation on remote nodes
- β — combining of preaggregation results from remote nodes
- γ — projection, final transformation

On-the-fly aggregation results filtering

- ***Idea:*** there is a class of `HAVING`-predicates, which allow runtime pruning of groups
- ***Requires:*** monotonicity and codomain analysis for aggregation expressions, minor redesign of aggregation algorithm
- It is really useful in a column-store with on-demand data reading:
  - Save I/O: do not read non-grouping attributes for pruned groups
  - Save CPU: if group is already pruned, we do not evaluate aggregation expressions and `HAVING`-predicate
  - But pruning condition should be checked for every record from non-pruned groups, so additional CPU work is required too

---

Details can be found in the paper "On-the-Fly Filtering of Aggregation Results in Column-Stores", available at `http://ceur-ws.org/Vol-2135/SEIM_2018_paper_37.pdf`

Experimental evaluation

```
SELECT
   l_returnflag , l_linestatus ,
   SUM(l_quantity) as sum_qty ,
   SUM(l_extendedprice) as sum_base_price ,
   SUM(l_extendedprice * (1 − l_discount))
                    as sum_disc_price ,
   SUM(l_extendedprice * (1 − l_discount) *
            (1 + l_tax)) as sum_charge ,
   AVG(l_quantity) as avg_qty ,
   AVG(l_extendedprice) as avg_price ,
   AVG(l_discount) as avg_disc ,
   COUNT(*) as count_order
FROM lineitem
GROUP BY l_returnflag , l_linestatus
HAVING having-clause
```

where ***having-clause*** is

- `l_linestatus = 'O'` for Q1;
- `having count(*) < 100000` for Q2;
- `(l_returnflag = 'A') OR (l_linestatus = 'O')) AND (MIN(l_tax) > MAX(l_discount)` for Q3;
- `SUM(l_quantity) < 1000000` for Q4.

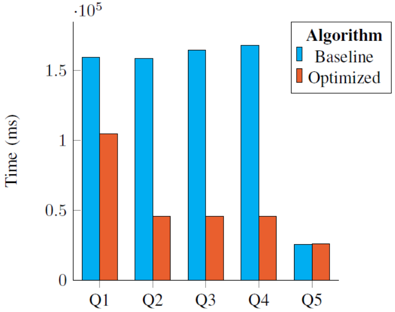


Fig. 1. Comparison of algorithm performance for SF=50

```
SELECT l_returnflag , l_linestatus
FROM lineitem
GROUP BY l_returnflag , l_linestatus
HAVING l_returnflag = 'A'
```

Q5

Window functions: syntax and concepts

```
window_function(column) OVER (
    [ PARTITION BY column [ , ... ] ]
    [ ORDER BY column [ ASC | DESC ] ]
    [ { ROWS | RANGE } BETWEEN frame_start AND frame_end ]),
```

where *frame_start* may be

- `UNBOUNDED PRECEDING`
- ***offset*** `PRECEDING`
- `CURRENT ROW`

and *frame_end* may be

- `CURRENT ROW`
- ***offset*** `FOLLOWING`
- `UNBOUNDED FOLLOWING`

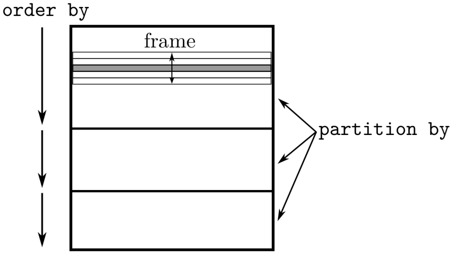
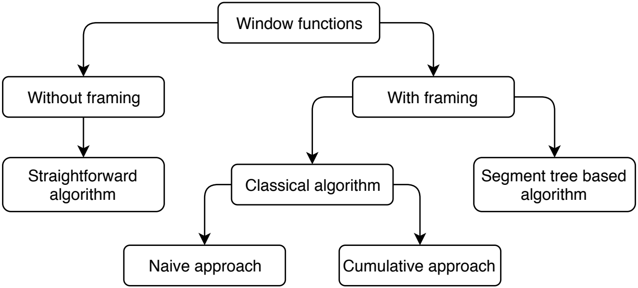


Image is taken from the paper “Efficient Processing of Window Functions in Analytical SQL Queries” of Viktor Leis et al., VLDB, 2015

Window functions: processing strategies

- Partitioning
  - using hash table
  - using sorting (can be combined with ordering)
- Ordering
  - separate for each group
- Evaluation

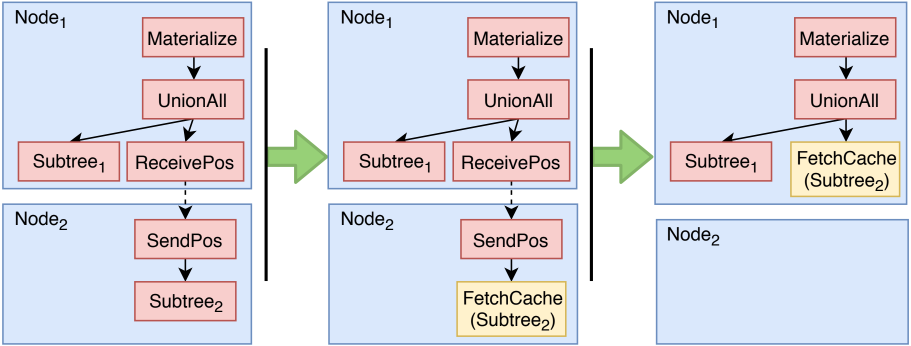


Window functions: our contribution

- Considering possible materialization strategies and memory consumption models for them
- Segment tree generalization
- Segment tree application for evaluation of `RANGE`-based window functions

Details can be found in the paper "Implementing Window Functions in a Column-Store with Late Materialization", available at `https://link.springer.com/chapter/10.1007%2F978-3-030-32065-2_21`

Intermediate results caching



- Before materialization; only positions are stored
- Intermediates are stored in-memory
- Allows compression of stored results to reduce memory footprint

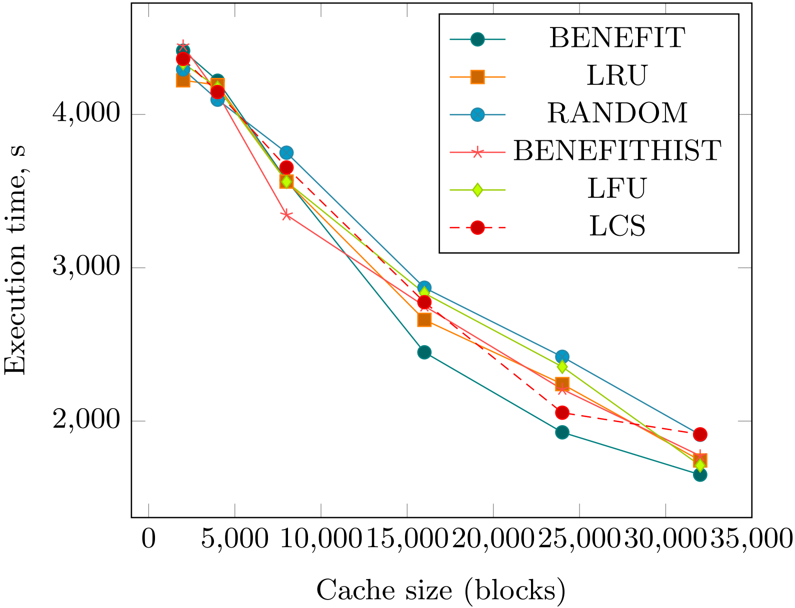
Intermediate results caching: the main idea

```cpp
struct QueryDescription {
    set<Partition> partitions;
    set<pair<Column, Column>> joins;
    set<ConstPredicate> const_predicates;
    set<ValueList> specific_values;
    Buffer plan;

    bool contains(const QueryDescription &other);
    double complexity();
    size_t expectedBlocks();
};
```

1. Reduce every subplan to a descriptive structure
2. Keep track of N last queries
3. Estimate every result's benefit as a function of computational complexity and size

Intermediate results caching: performance



Our team

Current members:

- Viacheslav Galaktionov
- Valentin Grigorev
- Evgeniy Klyuchikov
- Kirill Smirnov
- George Chernishev
- Nadezhda Mukhaleva

Ex-members:

- Anastasia Tuchina
- Evgeniy Slobodkin

System State: 2017 ⟷ 2018 ⟷ 2019

Technology stack: C++17, Git, Google Test, JSON, PostgreSQL (for result verification)

Characteristic	2017	2018	2019
Size	824 KB	1252 KB	2100 KB
Lines of code	11.5K	17.8K	28.4K
Classes	116	198	366
Modules	about 75	about 100	185
Tests	about 80	about 200	about 300
Commits	800+	1300+	1600+

Future plans

- Extending class of supported queries, move towards TPC-H and TPC-DS support;
- Query optimizer;
- Query adaptivity;
- Compression support;
- Technical stuff: REPL, visualization, parser (partially completed now)
- ...