Adaptive SQL Query Optimization in Distributed Stream Processing: A Preliminary Study

Darya Sharkova[1], Alexander Chernokoz[2], Artem Trofimov[3], Nikita Sokolov[3], Ekaterina Gorshkova[4], Igor Kuralenok[3], and Boris Novikov[1]

[1] HSE University, Saint Petersburg, Russia
[2] IFMO University, Saint Petersburg, Russia
[3] Yandex, Saint Petersburg, Russia
[4] JResearch Software, Prague, Czech Republic
sharkovadarya@gmail.com, chernokoz@hotmail.com,
{tomato,faucct}@yandex-team.ru, cathy@jresearch.org, solar@yandex-team.ru,
borisnov@acm.org

Abstract. Distributed stream processing is widely adopted for real-time data analysis and management. SQL is becoming a common language for robust streaming analysis due to the introduction of time-varying relations and event time semantics. However, query optimization in state-of-the-art stream processing engines (SPEs) remains limited: runtime adjustments to execution plans are not applied. This fact restricts the optimization capabilities because SPEs lack the statistical data properties before query execution begins. Moreover, streaming queries are often long-lived, and these properties can be changed over time.

Adaptive optimization, used in databases for queries with insufficient existing data statistics, can fit the streaming scenario. In this work, we explore the main challenges that SPEs face during the adjustment of adaptive optimization: retrieving and predicting statistical data properties, execution graph migration, misfit of SPEs programming interfaces, etc. We demonstrate the feasibility of the proposed approach within an extension of the NEXMark streaming benchmark and outline our further work on this topic.

1 Introduction

Modern day data analytics commonly requires near real-time online processing of continuously changing data arriving from unbounded streams. A standard way of defining a stream processing pipeline is an execution graph, each node of which represents an operation performed over stream elements. However, a declarative approach to defining computations is widely preferred in data analytics, but it has not gained wide use in streaming systems. SQL, typically used for executing database queries, is an example of the declarative approach to data processing. Among the many advantages of SQL are its popularity and ease of adoption, as

well as its support of windowed aggregations and joins, which are common tasks in stream processing.

Applications of SQL in stream processing have been an area of active research for the last two decades; however, there have not been many productive attempts at proposing a standard for robust streaming SQL. One such attempt [**?**] is fairly recent (2019); its predecessors, such as CQL [**?**], haven't found much popularity, and modern stream processing engines (SPEs) typically implement only a subset of SQL features. We hope that with the recent efforts in providing a standard for streaming SQL, the declarative approach to stream processing gains popularity, which would mean that declarative streaming query optimization is a task of great current interest.

Optimization is one of the stages of executing a declarative query. The execution stages are as follows: first, a query is *parsed* into an abstract syntax tree, each node of which represents a relational algebra operator; second, the query is simplified during the *rewriting* phase into a logical plan (or graph; we will be using these two terms interchangeably in the following text); third, the logical plan is transformed into a physical plan during *optimization*; finally, the query is *executed* and the result is delivered to the user [**?**]. The correspondence between queries and plans is not one-to-one: for each query, there can be multiple execution plans. The purpose of query rewriting is to reduce the space of execution plans and to standardize and simplify the query for further processing [**?**]; as for optimization, the query planner transforms each logical operator into its physical implementation: for example, a join operation can be implemented using the hash join algorithm or the merge join algorithm [**?**], or a join operation in a distributed system might be preceded by sharding.

Database query optimization is a well-researched topic [**?**, **?**, **?**]. The two common approaches to query optimization are rule-based and cost-based. Rule-based optimization applies data-independent transformations that are guaranteed to always improve performance. One such transformation is pushing filtration and projection operators down as low as possible unless it could influence other operators. Cost-based optimization estimates a cost function value for each considered plan and selects the plan with the minimum cost value. The cost function is typically a linear combination of expected I/O and CPU costs, with CPU costs of each operation estimated based on relation cardinality and operator selectivity. Therefore, cost-based optimization requires knowledge of statistical information [**?**]. However, obtaining such knowledge in streaming systems presents certain difficulties, discussed further in the paper.

Efforts to optimize streaming queries execution in a distributed environment focus on finding a suitable mapping from a logical graph to a physical graph executed on a machine [**?**, **?**, **?**, **?**, **?**], but such optimizations are local; a global optimization would require finding the most suitable logical graph as well as physical, which is possible with a declarative approach but not with using a user-defined execution graph. The problem of logical level declarative query optimization is currently relevant and presents a challenge.

In this paper we:

- Present a detailed analysis of the problem of SQL queries optimization in distributed stream processing and discuss challenges that arise within this problem;
- Describe preliminary experiments that we have conducted in order to demonstrate feasibility of streaming SQL optimization.

The remainder of this paper is structured as follows. First, we state the problem as illustrated by a running example (Section 2). Second, we list the challenges in adaptive optimization of streaming SQL queries (Section 3). Then, we present the preliminary experiments we executed on our running example to demonstrate the feasibility of the proposed approach to query optimization and discuss the results (Section 4). Finally, we discuss related work including efforts on database and streaming query optimization (Section 5).

2 Problem statement

In this section, we illustrate the problem of streaming SQL query optimization using a running example of a query for a streaming system.

We are using the NEXMark benchmark [**?**] for our query. The NEXMark benchmark suite, designed for queries over continuous data streams, is an extension of the XMark benchmark [**?**] adopted for use with streaming data. The NEXMark scenario simulates an on-line auction system with three kinds of entities: people selling items or bidding on items, items submitted for auction, and bids on items. These kinds of entities will be referred to as `Person`, `Auction`, and `Bid` respectively. The original NEXMark benchmark includes eight queries which utilize the full spectrum of SQL features, but none of them contain more than one join operator. We add the following query based on the NEXMark model:

```
SELECT P.name, P.city, P.state,
       B.price, A.itemName
  FROM Person P
    INNER JOIN Bid B
      ON B.bidder = P.id
    INNER JOIN Auction A
      ON A.seller = P.id
```

Listing 1.1: The proposed NEXMark-based query in the Apache Calcite SQL dialect

This query selects all the people who have joined the auction as both bidders and sellers; for each such person their name, city and state of residence are selected, as well as the price of each of their bids and the name of each item they are selling at the auction.

This query contains two join operators, which means that there are at least two ways to execute this query: for example, first performing the join between `Person` and `Auction`, then joining the result with `Bid`, or vice-versa.

The logical and physical plans (with substituted variable names omitted; using Apache Beam transforms as operators) for our example query are as follows:

```
LogicalProject(name, city, state,
               price, itemName)
  LogicalJoin(condition, joinType=inner)
    LogicalJoin(condition, joinType=inner)
      BeamIOSourceRel(table=Person)
      BeamIOSourceRel(table=Bid)
    BeamIOSourceRel(table=Auction)
```

Listing 1.2: Logical plan

```
BeamCalcRel(name, city, state, price, itemName)
  BeamCoGBKJoinRel(condition, joinType=inner)
    BeamIOSourceRel(table=Bid)
    BeamCoGBKJoinRel(condition, joinType=inner)
      BeamIOSourceRel(table=Person)
      BeamIOSourceRel(table=Auction)
```

Listing 1.3: Physical plan

However, it is also possible for the query planner to generate the following physical plan for this query, with the two join operators in a different order:

```
BeamCalcRel(name, city, state, price, itemName)
  BeamCoGBKJoinRel(condition, joinType=inner)
    BeamIOSourceRel(table=Auction)
    BeamCoGBKJoinRel(condition, joinType=inner)
      BeamIOSourceRel(table=Person)
      BeamIOSourceRel(table=Bid)
```

Listing 1.4: Alternative physical plan

In unbounded data streams, elements can be grouped into windows, based on event time or the number of tuples in each window, and each window can be processed similarly to a SQL table. Cost-based optimization requires statistical knowledge about the data, such as the cardinality of each window, which can be inferred from element arrival rate in case of streaming data. In our example, the first plan, where `Person` and `Bid` are joined first, and then the result is joined with `Auction`, would be preferable if the arrival rate of auctions significantly exceeded the arrival rate of bids, meaning that while many items have been getting put up for auction, not many sellers have been making bids. If, however, after some time sellers started making a lot of bids, the second execution plan would have a lesser cost value. Thus, as data statistics change over the course of the query execution, a previously optimal plan might become inefficient.

Various DBMS have adapted a technique known as adaptive optimization. This technique utilizes a so-called adaptivity loop, during which an adaptive sys-

tems measures and evaluates data characteristics and then uses these evaluations to select a new query plan better suited to the current data [**?**].

It stands to reason that a similar technique could be useful in streaming queries optimization, seeing as streaming data statistics tend to change over time, rendering the previous query execution plan non-optimal. Thereby, in this paper we consider the problem of adaptive optimization of streaming SQL queries in distributed stream processing systems.

However, adaptive database optimization is not applicable to streaming SQL queries as is: in databases, such optimization is applied during the execution of a long-running query, with the queried data remaining the same, while in streaming systems the data is continuously updated, and the same query is executed on each subsequent window; therefore, it would make sense for streaming systems to optimize the execution graph between the windows. This, as well as other specifics of SPEs, presents certain challenges in implementing logical level streaming query optimization, which we explore in the following section.

3 Challenges

In this section, we present the challenges in adapting SQL optimization techniques to data streams. We categorize each challenge as either research or technical.

3.1 Fetching and predicting data statistics

Cost-based optimization requires statistical information on data in order to calculate cost function values for each plan. However, upon the start of a streaming query execution no information about the data, such as its arrival rate, is available. Therefore, in order to properly apply cost-based optimization to streaming SQL queries, it is necessary to collect data statistics over the course of query execution. Moreover, since we possess no definitive knowledge about the arriving data, in order to utilize an optimal plan for the upcoming windows, we need to predict statistics for each next window based on statistics for previous windows. To this end, we identify two challenges in using statistics for streaming query optimization:

- **Technical**: Statistical information on stream elements, such as their arrival rate, needs to be collected during execution at runtime without seriously affecting the performance of a distributed SPE.
- **Research** Next window statistics should be predicted using statistics collected for previous windows. We assume that previous window statistics would present a decent baseline; however, this assumption needs to be tested. it is a expected that a simple baseline wil be good enough in most simple cases but in general prediction is not an easy task.

Popular SPEs and frameworks for defining streaming workflows do not offer any statistics fetching or predicting. For example, Apache Beam, which utilizes

Apache Calcite, a dynamic data management framework, for its SQL processing functionality, passes constant values to the query planner instead of any actual data statistics.

3.2 Using statistics for streaming query optimization

The API of the current state-of-the-art systems typically utilizes a top-down approach to building a graph for query execution: first a logical graph is created, then it is transformed into a physical graph, which is later used for execution, leaving no opportunity to pass any data from the physical level to the logical level and use it to adapt the graph to the new data and therefore making any runtime adjustments to execution plans impossible. While progress has been made in applying various optimizations to the execution graphs at the physical level (see [**?**] or Google Cloud Dataflow optimizer), there are no significant results in logical level optimization yet, and the logical level allows to use more complex optimizations than the physical level. Moreover, in this paper we discuss distributed stream processing engines, which require additional consideration when it comes to query execution. To this end, we identify the following challenges:

- **Technical**: it is necessary to adapt the API of current SPEs to pass statistics collected or predicted during the query execution at the physical level to the query planner at the logical level.
- **Research**: the relational algebra and the planner cost model should be extended with new operators specific to distributed systems. For example, a join operation performed upon a stream with a low arrival rate of new elements would be broadcasted to all the nodes in the system, while a high arrival rate suggests distributing different keys across different nodes. Therefore, a new *distribution* operator should be included into the relational algebra, and the cost model should include the estimate of its cost. The cost model should consider the latency due to communication between nodes as well. Such cost models have been well-researched in distributed databases [**?**] but not in distributed streaming systems.

3.3 Execution graph migration in runtime

Streaming data is ever-changing: new data continues arriving indefinitely, and the statistics for the next window elements might differ significantly from the statistics for previous windows. Even if the optimal query execution plan was selected based upon statistics accumulated for previous windows, the new data statistics might be different enough to render the previous plan no longer optimal. In order to adapt to the changes in data, it's necessary to identify the moment in time in which the previous plan is no longer optimal for the current or upcoming data and to migrate the execution to a new graph. We identify the following challenges in migrating the execution graph in runtime:

- **Technical**: the mechanics of graph migration, particularly for stateful operations such as joins and aggregations, in runtime have been researched [**?**]; however, most current popular SPEs do not provide such functionality. In order to support graph migration in stream processing frameworks such as Apache Beam, one such strategy is the parallel track strategy described in [**?**]: a second graph can start the query execution along the first one, and the first one can terminate execution once the current window has been processed, so that all the subsequent windows are only processed by the new execution graph.
- **Research**: identifying the point in time at which it's feasible and beneficial to perform the graph migration is an open problem. First of all, it's necessary to be able to estimate the costs of graph migration at the current point in time. Secondly, we need to establish what qualifies as substantial enough change in statistics to warrant graph migration.

4 Preliminary experiments

In this section, we describe the preliminary experiments that we conducted in order to demonstrate how the choice of a query plan affects the performance. We aim to show that a well-timed switch from an execution graph that is no longer optimal for the current data to a more optimal one would improve performance significantly. First, we present the experiment setup and the configuration used; then, we demonstrate our results.

4.1 Setup

For our experiments, we used the same query that was described in Section 2 and the Apache Beam implementation of the NEXMark benchmark model. In this implementation, each entity (`Person`, `Auction`, or `Bid`) is represented via a subclass of the `Event` class. Each event is generated by an unbounded source in accordance to the provided configuration, which includes parameters such as the arrival rate for each event, the $|Person| : |Auction| : |Bid|$ ratio, the time-based window size, etc.

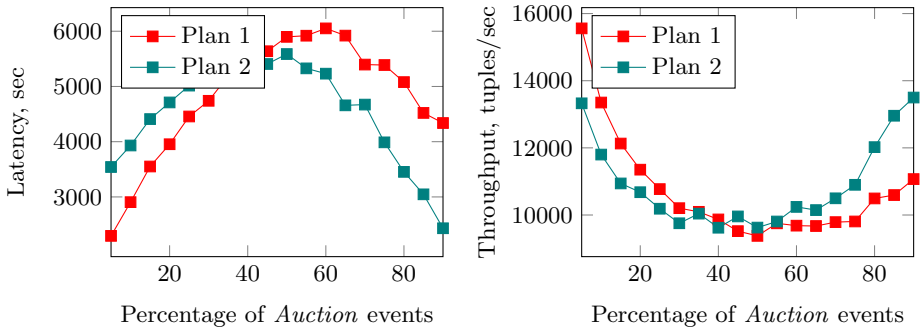
First, we execute this query using the plan in which `Auction` and `Person` are joined first, and the result is joined with `Bid`; then, we use the plan in which `Person` and `Bid` are joined first (see Section 2). For each run, we use a different $|Person| : |Auction| : |Bid|$ ratio.

To evaluate performance, we measure latency and throughput for each window. For a join result, we consider *latency* to be the difference between the maximum arrival time of each of the rows making up the join result and the output time of the resulting row; then, we select the maximum out of the latency values of all the rows in each window. The throughput that we measure is *sustainable throughput*, i.e. the maximum events arrival rate that a streaming system can handle without the continuous buildup of latency.

We have conducted our experiments on a local machine equipped with a 1.4 GHz Intel Core i5-8257U CPU (4 cores) and 8 GB of memory using the Apache Flink runner.

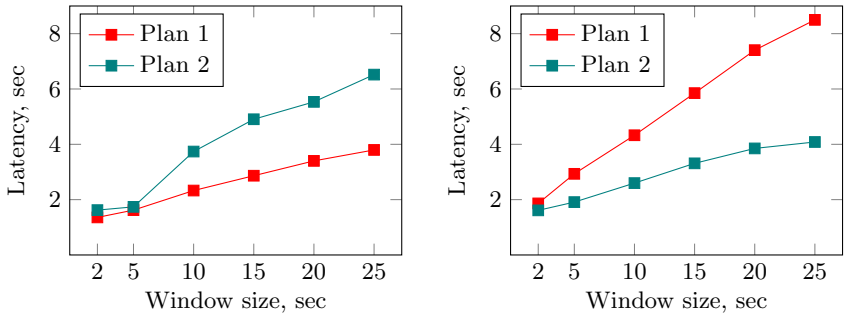
4.2 Results

In the subsequent text the plan which joins `Auction` and `Person` first is referred to as *Plan 1*; the plan which joins `Person` and `Bid` first is *Plan 2*.



Latency Throughput

Fig. 2: Latency and throughput for different ratios: out of 100 events, $|Person| = 5$, $|Auction|$ is the value on the x-axis, $|Bid| = 100 - |Person| - |Auction|$



$|Person| : |Auction| : |Bid| = 5{:}5{:}90$ $|Person| : |Auction| : |Bid| = 5{:}90{:}5$

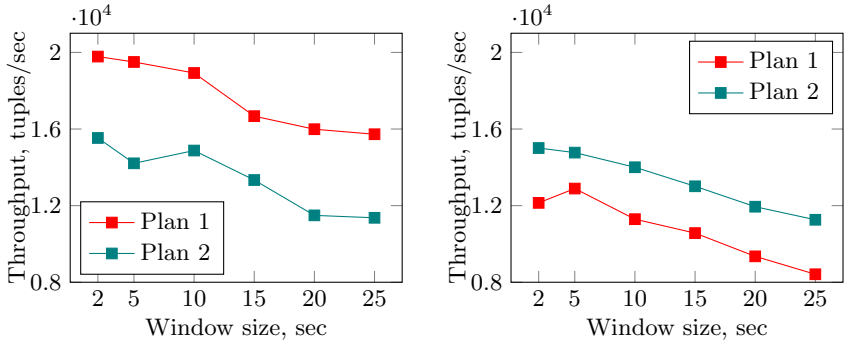
Fig. 4: Latency for different window sizes and $|Person| : |Auction| : |Bid|$ ratios

Latency We generated 1000000 events with the arrival rate of 10000 events per second and time-based windows of varying sizes.

Figure 1a demonstrates how latency changes depending on data characteristics. The $|Person| : |Auction| : |Bid|$ ratio impacts arrival rate for each kind of entities, thereby influencing latency. As expected, the plan in which `Person` and `Auction` are joined first delivers better results when the arrival rate of `Bid` records significantly overwhelms the rates of `Person` and `Auction` (the 5:5:90 ratio is an example of such a case), while the plan in which `Person` and `Bid` are joined first works best for cases where the rate of `Auction` records far exceeds those of `Person` and `Bid`.

As Figures 3a and 3b demonstrate, the latency does not grow linearly as the window size increases. This is due to the fact that the join operator processes the records as they arrive instead of starting to process them only after the last record in the window has arrived, thus the results are ready to emerge shortly thereafter the arrival of the last record in the window.

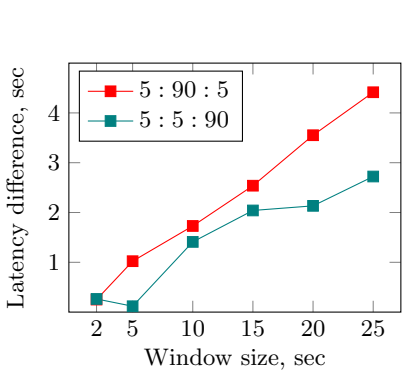
Figure 7a demonstrates how the difference in latency for the two execution plans changes with the window size. Since the difference grows as the window size increases, statistics-based optimization should provide an even bigger performance gain for larger windows.



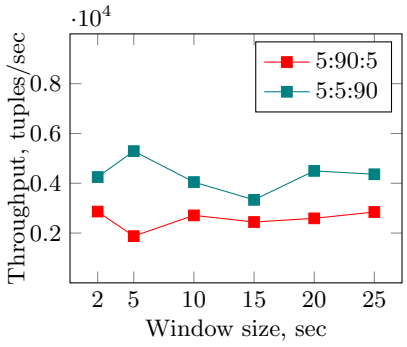
$|Person| : |Auction| : |Bid| = 5{:}5{:}90$ $|Person| : |Auction| : |Bid| = 5{:}90{:}5$

Fig. 6: Throughput for different window sizes and $|Person| : |Auction| : |Bid|$ ratios

Throughput The parameters for throughput estimation were the same as for the latency estimation. As Figure 1b shows, Plan 2 delivers higher throughput in case of the arrival rate of `Auction` significantly exceeding that of `Person` and `Bid`, while Plan 2 performs better in case of the arrival rate of `Bid` being significantly higher. This corresponds with the latency measurements. Throughput decreases with the increase of window size, as shown in Figures 5a and 5b.



Latency difference for Plans 1 and 2



Throughput difference for Plans 1 and 2

Fig. 8: Difference in latency and throughput for Plans 1 and 2

Throughput difference remains constant and does not depend on the window size, Figure 7b demonstrates.

4.3 Discussion

The results of our experiments demonstrate that streaming query execution performance depends on the plan used for the execution, and the optimality of the plan depends on the data characteristics, which proves the necessity of adaptive optimization of streaming queries. Particularly, the first steps towards adaptive optimization should be predicting statistics for each window and performing runtime graph migration, since the results of the experiments show that even the current planners, such as the Volcano query planner [**?**] used in Apache Calcite, which are unaware of whether the data comes from a stream or a table, could use those statistics to produce a better execution plan. These two challenges will be the focus of our future work. After that, the planner can be enhanced by introducing distributed streaming systems-specific operators and their costs.

5 Related work

We structure the related work into three areas: database query optimization, optimization of streaming queries, and predicting data statistics for queries using machine learning techniques.

Early efforts in database query optimization include the System R optimizer [**?**], which introduced a dynamic programming algorithm for finding an optimal execution plan; the Starburst optimizer [**?**], which proposed new rule-based optimization techniques; and others: a survey chapter on query optimization in relational databases can be found in [**?**]. Our particular interest lies in the

areas of query optimization in distributed database systems and adaptive query optimization. The former has been described in [?]; a detailed survey [?] is a good reference for the latter. Adaptive optimization of streaming SQL queries is different from database optimization: while in databases the execution plan is modified during the execution of a query, in streaming systems the plan should be changed after a window has been processed but before the next window processing has started; moreover, database data is static during the query execution while streaming data is continuously updated.

An overview of streaming query optimizations can be found in [?]: most of these optimizations can be classified as rule-based and are applied statically at compilation time, although dynamic versions are listed for several of these; we are interested in adaptive optimization at runtime. Various works explore adaptive optimization of streaming queries: [?] focuses on finding the optimal order of pipelined filter operators, with possible reordering at runtime; it uses the current known data statistics while we are interested in predicting the next window statistics and using those. Other works focus on physical level adaptive optimizations: one such study can be found in [?]; we are interested in logical level optimization instead. Another approach for physical optimization is discussed in [?]. It is based on the intermediate representation (IR) of the streaming queries, which supports a wide range of physical optimizations: operator redundancy elimination, operator separation, and fusion [?]. However, the optimizations based on operator reordering is limited within this method due to the lack of algebraic properties that is usually built around the relational model.

Previous works on execution graph reconfiguration in runtime, including Eddies project [?] and StreaMon [?], primarily study centralized stream processing, but, as we mentioned in Section 3, many issues originate from the distributed environment. The approach for state migration in distributed environment introduced in [?] can be applied as a building block for the general graph reconfiguration mechanism.

Some studies ([?,?]) use machine learning to predict optimal execution plans, while others explore join cardinality prediction. Database cardinality prediction techniques can be categorized as either query-driven or data-driven. Query-driven prediction models learn on sets of queries; among studies employing this approach are [?,?,?,?], which use neural networks. Data-driven prediction models, described in [?] and [?], are trained on data without queries and attempt to learn characteristics such as distribution of single attributes as well as joint distributions of multiple attributes. Neither of these approaches fit the streaming queries scenario: SPEs are executing the same query on each subsequent window, rendering query-driven approaches unusable, and the data is continuously updating, which means data-driven approaches are not employable either. Additionally, neural networks might not be the best choice of a learning model for a small amount of data contained in a single window; instead, it would potentially be more beneficial to use a model similar to [?], a fixed-size ensemble of heuristically replaced classifiers. Predicting statistics for streaming queries might yield better results than for database queries since the query is being run on differ-

ent subsequent windows for an extended period of time, unlike in databases, where each new query is run separately from its predecessors. This could be advantageous in predicting statistics not only for the next input window but for intermediate execution results as well.

6 Conclusion

In this paper, we considered the problem of adaptive optimization of streaming SQL queries in distributed stream processing engines. We argued the necessity of adaptive query optimization at the logical level and highlighted the differences between query optimization in database tables and data streams, which allowed us to identify the following challenges in adapting database optimization techniques for streaming queries: fetching and predicting data statistics, using predicted statistics for the query optimization, and migrating the execution graph in runtime once the statistics have changed enough that the previous execution plan becomes suboptimal. We presented a running example of a streaming SQL query and performed experiments on this query and its different possible execution graphs to demonstrate the possibility of a gain in performance should the graph be adapted to the data statistics.