Towards Adaptive SQL Query Optimization in Distributed Stream Processing

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 online processing of unbounded streams of continuously changing data. A standard way of defining a stream processing pipeline is an execution graph, each node of which represents an operation performed over stream elements. A preferred way of defining streaming pipelines is a declarative approach, widely used in bounded data analytics. SQL is one of the implementations of the declarative approach to data processing.

Application of SQL to stream processing has been an area of active research for the last two decades, however, there have not been many productive attempts at providing a standard for robust streaming SQL. One such attempt [2] is fairly recent (2019); its predecessors, such as CQL [1], haven't found much popularity, and modern stream processing engines (SPEs) typically implement only a subset of SQL features.

Seeing as there is no uniform standard for streaming SQL implemented in popular SPEs as of yet, not many attempts have been made at streaming SQL query optimization. Similar to database query execution, SPEs parse the query text into an abstract syntax tree with each node representing a relational algebra operator, which is later transformed into a logical graph, and then into a physical graph. Since one query can be represented by multiple execution graphs, a query planner (optimizer) must choose the best possible graph (i.e. the one providing the best performance). Database query optimization is a well-researched topic [this is the part where i say that it allows for complex optimizations but no such results have been achieved in stream processing where optimizations are performed on the physical level — cite the grizzly article — so it's limited by non-declarative approach]

Conference'17, July 2017, Washington, DC, USA
© 2021 ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Proposed contribution

In this paper we:

- Identify which features stream processing engines lack in order to support query execution graph optimization at the logical level;
- Describe preliminary experiments that we have conducted in order to demonstrate feasibility of streaming SQL optimization;
- Outline the next steps in implementing streaming SQL optimization support;

1.2 Paper structure

The rest of the paper is organized into the following sections:

- Section 2 introduces a running example of a streaming SQL query to demonstrate the optimization problem;
- Section 3 describes the challenges of adapting database optimization techniques to streaming queries due to the specifics of stream processing;
- Section 4 presents the preliminary experiments [TODO]
- In section 5 we discuss the results of the experiments described in section 4 and propose the next steps in implementing support for streaming SQL optimization at the logical level;
- Section 6 presents an overview of the area as well as details recent contributions in query processing optimization.

2 A RUNNING EXAMPLE

In this section we demonstrate the streaming SQL optimization problem on a running example of a query executed on a stream processing engine.

We are using the NEXMark benchmark [5] for our query. The NEXMark benchmark suite, designed for queries over continuous data streams, is an extension of the XMark benchmark [4] 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
```

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 `Bid`, then joining the result with `Auction`, or vice-versa.

Similarly to database SQL queries, streaming queries are parsed into abstract syntax trees from which the logical plan is built, and the logical plan is transformed into a physical plan. The logical and physical plans for our example are as follows:

[there are plans in the comments. they'd be in two columns]

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

[see the comments]

A database optimizer typically selects a plan to be used for execution based upon certain table statistics, such as table cardinality or selectivity of a predicate used in the join operator. A cost function value is computed using these statistics and the plan with the minimum cost function value is selected for execution. A non-optimal order of operators might lead to an increase in the processing time of the query.

While it is possible to apply the same kind of thinking to stream processing optimization, current stream processing engines are not designed to collect and use any statistics for data streams. For example, Apache Beam, which uses Apache Calcite for its SQL processing functionality, passes constant values as data stream statistics to the Calcite query planner. Had it been possible to pass the actual per-window statistics collected and/or predicted during the query execution to the query planner, it would have allowed for a plan optimized specifically for the current data to be used; moreover, if, as it is typical for streaming data, the statistics would change so much at some point that the previous plan would no longer be optimal for the current data, it would be possible to perform adaptive optimization of the graph.

3 CHALLENGES

This section presents the challenges in adapting SQL optimization implementations to data streams.

3.1 Fetching and predicting data statistics

Cost-based query optimization requires knowledge of certain statistics about the data being processed. For example, one such statistic might be the arrival rate of the elements in the stream. For database tables, most enterprise database management systems only use cardinality, and since it is common for DBMS to execute many different queries over *immutable* data, cardinality is immutable throughout the query execution as well and is known beforehand.

This is not the case with data streams. When the elements first start arriving, the SPE has no prior knowledge about their rate or any other statistics. Additionally, since SPEs have to keep running the same query over long periods of time, the statistics are likely to change over the course of query processing with the arrival of new data.

In order to obtain and use statistics for streaming data, we propose collecting statistics for each window, which can be difficult in distributed stream processing systems, as throughput and latency can be affected. However, predicting statistics for the next window could be used for calculating a cost function value to be used for streaming query optimization. The previous window statistics present a strong baseline for this case.

3.2 Statistics integration in query processing

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 [3] or Google Cloud Dataflow optimizer), there are no significant results in logical level optimization yet, and the logical level allows for more complex optimizations than the physical level.

3.3 Execution graph migration in runtime

4 PRELIMINARY EXPERIMENTS

5 DISCUSSION

6 RELATED WORK

7 CONCLUSION

REFERENCES

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142.

[2] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 1757–1772. https://doi.org/10.1145/3299869.3314040

[3] Philipp M. Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2487âĂŞ2503. https://doi.org/10.1145/3318464.3389739

[4] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J Carey, Ioana Manolescu, and Ralph Busse. 2002. XMark: A benchmark for XML data management. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 974–985.

[5] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark–A Benchmark for Queries over Data Streams (DRAFT)*. Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers.