ProgressiveDB – Progressive Data Analytics as a Middleware

Lukas Berg
TU Darmstadt
Germany
lukasjulian.berg@stud.tu-darmstadt.de

Tobias Ziegler
TU Darmstadt
Germany
tobias.ziegler@cs.tu-darmstadt.de

Carsten Binnig
TU Darmstadt
Germany
carsten.binnig@cs.tu-darmstadt.de

Uwe Röhm
University of Sydney
Australia
uwe.roehm@sydney.edu.au

ABSTRACT

ProgressiveDB transforms any standard SQL database into a progressive database capable of continuous, approximate query processing. It introduces a few small extensions to the SQL query language that allow clients to express progressive analytical queries. These extensions are processed in the *ProgressiveDB* middleware that sits between a database application and the underlying database providing interactive query processing as well as query steering capabilities to the user. In our demo, we show how this system allows a database application with a graphical user interface to interact with different backends, while providing the user with immediate feedback during exploratory data exploration of an on-time flight database. ProgressiveDB also supports efficient query steering by providing a new technique, called progressive views, which allows the intermediate results of one progressive query to be shared and reused by multiple concurrent progressive queries with refined scope.

PVLDB Reference Format:
Lukas Berg, Tobias Ziegler, Carsten Binnig and Uwe Röhm. ProgressiveDB – Progressive Data Analytics as a Middleware. *PVLDB*, 12(12): 1814-1817, 2019.
DOI: https://doi.org/10.14778/3352063.3352073

1. INTRODUCTION

Motivation. Interactive visualisations are arguably the most important tool to explore, understand and convey facts about data. For example, as part of data exploration visualisations are used to quickly skim through the data and look for patterns along various dimensions of the data. This requires to generate a sequence of visualisations and allow the user to interact with them. A recent study [6] has shown that visual delays of 500ms tend to decrease both end-user activity and data set coverage, due to the reduction in rates of user interaction that is crucial for overall observation, generalization and hypothesis.

Unfortunately, existing database systems are ill-suited for these types of interactive workloads that result from visual exploration tools for different reasons: (1) Unfortunately, when the data sets are larger, computing results for even a single visualisation can take seconds or even minutes, creating a significant barrier to interactive data analysis. One promising route to compute continuously refining results that was already published in the late 1990's was online aggregation [4]. Different from offline-sampling, which is used in BlinkDB [1] or VerdictDB [8], online aggregation can provide interactive response times for arbitrary aggregate group-by queries over single tables and more recent extensions also support queries including joins [7, 5]. However, online aggregation has not yet made its way into any of the commercial DBMSs available today. (2) In interactive data exploration, users often want to quickly refine a query. For example, in some customer analysis a user might quickly try out different age ranges as filter criteria. In today's DBMS, every user interaction results in a new SQL query putting a high overhead on the performance of the system.

Contribution. In this paper, we propose *ProgressiveDB*, a middleware that provides progressive execution capabilities on top of non-progressive databases such as PostgreSQL, MonetDB, etc. That way, *ProgressiveDB* transforms any standard SQL database into a progressive DBMS.

In order to provide progressive query execution to applications, *ProgressiveDB* introduces a few but important extensions to the SQL query language (called progressive extensions) that allow clients to express progressive analytical queries and consume continuously refining results. Furthermore, the SQL extensions of *ProgressiveDB* additionally support query steering that can be used by applications to change running queries without the need to start a separate query for each user interaction.

For executing the progressive SQL extensions, *ProgressiveDB* rewrites the incoming queries into a set of smaller queries that are continuously executed against the underlying database. *ProgressiveDB* is that way capable of contin-

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352073

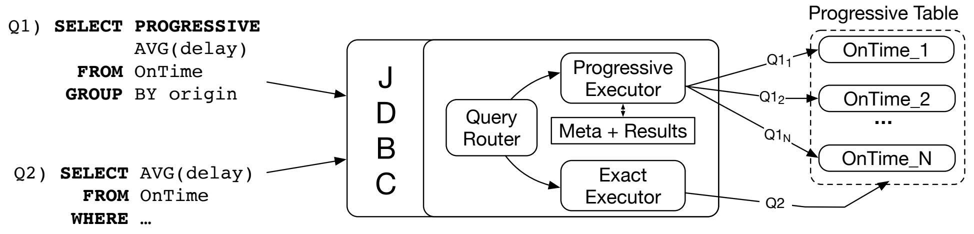


Figure 1: System Architecture of *ProgressiveDB*

uous query processing to answer analytical queries at interactive speeds and refine results in a progressive manner.

The contributions of this demo paper are as follows:

- We define several extensions of SQL that allow users to express progressively executed queries with additional query steering capabilities.
- We present the design of *ProgressiveDB*, a middleware engine that transforms any SQL database system into a progressive DBMS and discuss progressive query processing algorithms of *ProgressiveDB* that are implemented in our middleware which is based on Apache Calcite.
- We discuss our demonstration scenario, where an application on top of *ProgressiveDB* can be used to explore and analyse a flight delay database [3].

Outline. In the remainder of this paper, we first introduce the system architecture of *ProgressiveDB* (Section 2), before we define our progressive extensions of SQL (Section 3) that *ProgressiveDB* supports. Section 4 describes the demo application that allows attendees to interact and explore *ProgressiveDB* with the flight database.

2. SYSTEM OVERVIEW

The goal of *ProgressiveDB* is to enable progressive query processing and steering of queries without changing the underlying DBMS. Consequently, we utilize a middleware-based approach that provides progressive query capabilities to applications via a standard JDBC interface. This middleware handles all query routing, intermediate result processing, and manages the internal state of the progressive query execution as depicted in Figure 1. To support progressive query execution, a table has to be specifically prepared and made available to Progressive-SQL as a chunked *Progressive Table* (depicted on the right side of Figure 1). What is important is that chunking can be typically implemented by simply using the partition functions of an underlying database. See Section 3.1 for details.

As a query interface, *ProgressiveDB* supports both — standard SQL queries, like $Q2$, which are directly executed on the underlying DBMS on the whole table, and progressive queries as shown for $Q1$ formulated in our Progressive-SQL language (which is a SQL extension). Both are handled by the *Progressive Executor* of the middleware. For executing a query formulated using our progressive extensions for SQL like $Q1$, the Progressive Executor translates the incoming progressive queries into a series of sub-queries ($Q1_1$ - $Q1_N$) on the prepared Progressive Table. Each sub-query is executed on the underlying DBMS on a per chunk basis – typically a single partition of the table (cf. Section. 3.1). The Progressive Executor progressively combines the partial results over individual chunks with the current state of the query execution. Every time a new sub-query has finished the approximate overall result is progressively updated and sent to the application. Consequently, the more of the table is processed (i.e., the more sub-queries have been finished) the lower the relative error. See Section 3.2 for details.

Furthermore, the middleware is also responsible to provide efficient execution strategies for query steering that are also part of our progressive SQL extensions. For example, part of our SQL extensions are so called progressive views that allow an application to refine an already running query with additional filtering predicates or to drill-down its grouping categories. In this case, the *ProgressiveDB* middleware manages all necessary intermediate state to do so without having to re-run any already completed (sub-)query. See Section 3.3 for details.

Our implementation of *ProgressiveDB* is based on the *Apache Calcite* data management framework [2]. For *ProgressiveDB*, we extended Calcite with the Progressive-SQL syntax, and added the progressive executor components.

3. PROGRESSIVE SQL

Applications using *ProgressiveDB* can express analytical queries in *Progressive SQL* which allows them to execute queries progressively with interactive latency on the underlying database, and also with the possibility to further refine an already running progressive query (i.e., query steering).

3.1 Database Preparation

In order to be used by Progressive SQL, a table has to be prepared for progressive queries first. *ProgressiveDB* partitions tables into chunks holding data such that each partition is small enough to be queried by an analytical query within a given query latency. An important aspect of this preparation step is that tuples are assigned randomly to each chunk so that the individual sub-queries sent by *ProgressiveDB* access indeed a random sample of the table.

Note that chunking can be implemented without the need to copy the data of a table in the underlying database system. Instead, if a database system natively supports partitioning (e.g., most commercial DBMS or PostgreSQL) *ProgressiveDB* makes use of this capability: the table will be partitioned into n randomized 'chunks' such that the average response time of a simple aggregation query is below the configured threshold latency. Note that for most systems partitioning is natively supported. For systems without partitioning support, a chunking column and an index on that column are used by *ProgressiveDB*.

3.2 Progressive Queries

With progressive query execution, clients continuously receive approximate query results which are progressively getting more accurate approaching the final result. Clients initiate a progressive query by adding the keyword PROGRESSIVE to an aggregation query:

```
  SELECT PROGRESSIVE aggregation_list
    FROM relation
   WHERE condition
GROUP BY group_attrs
```

ProgressiveDB translates such a progressive aggregation query Q internally into a series of sub-queries q_i such that each sub-query queries only a single chunk (partition) of the prepared progressive table. The results of each sub-query q_i are then combined in the middleware with the current state of the progressive query execution to produce an approximate overall aggregation result at stage i.

ProgressiveDB computes the $aggregation\_approximation_i$ based on the combined result of all previous sub-queries q_j ($j < i$) and the additional partial result of sub-query q_i. This is possible for all decomposable aggregation functions, such as SUM, COUNT and AVG, where the result can be computed by aggregating over sub- or auxiliary-aggregates for subsets of the data. In case of SUM and COUNT, the sub-aggregate values are simply added; in case of AVG, the final aggregate value is computed from the queried partial SUM and COUNT auxiliary-aggregate values. These intermediate aggregation values are further scaled based on the query progress, and the number and size of the chunks of the progressive table.

Clients receive a continuous refining query result that consists of the progression of approximate aggregation results after each sub-query. Each result row is tagged with the corresponding sub-queries sequence number and a confidence value for the approximation (note that a general grouping query could have multiple result rows, one per group, which would all be tagged with the same sequence number):

```
0, group_attr, aggregation_approximation_0, confidence_0
1, group_attr, aggregation_approximation_1, confidence_1
2, group_attr, aggregation_approximation_2, confidence_2
...
```

This way, from the viewpoint of a client, the aggregation results are progressively getting closer to the exact result. It is the responsibility of the client application to use these partial approximate query results to progressively update their user interface. Basically, query results with sequence number i replace query results with sequence number $i-1$.

The current implementation is limited to single table queries. In future, we plan to support joins in *ProgressiveDB* following the ideas of [5], but adapted to use the table chunks available in *ProgressiveDB*.

3.3 Query Steering

During interactive data exploration, users often want to quickly refine a query, for example to further filter a result or drill-down into a result. While the progressive queries as introduced above give fast (low latency) and progressive feedback to users, they do not allow the user to steer the query at runtime. With our SQL syntax so far, refining a query would require for each step a new query which has to be processed by the underlying database from scratch – resulting in both execution overhead and a delay in achieving a result accuracy comparable to the already running query.

To facilitate the steering of progressive queries, Progressive SQL introduces the concept of a PROGRESSIVE VIEW for which some FUTURE grouping attributes and filter conditions can be specified:

```
CREATE PROGRESSIVE VIEW name AS
    SELECT aggregation_list
      FROM relation
     WHERE condition (AND condition_i FUTURE)*
  GROUP BY grouping_list (, group_i FUTURE)*
```

The general semantic of a PROGRESSIVE VIEW is the same as with a progressive query: *ProgressiveDB* executes a series of sub-queries q_i such that each sub-query queries only a single chunk table of the prepared database schema, and progressively combines these partial results with the current query execution state to new result approximations. The executed sub-queries, however, include additional grouping attributes for each grouping FUTURE clause and necessary attributes for FUTURE conditions from the view definition. The sub-queries also pre-filter the data with their filter conditions constructed such that any FUTURE conditions can still be evaluated on the view's result approximations.

ProgressiveDB caches the processed partial results tagged with sequence number i and confidence values in its internal state under the view name. A client application can query a progressive view with any combination of the grouping and filter attributes, and *ProgressiveDB* will return only the aggregated results for the corresponding groups and matching filter conditions from its internal cache.

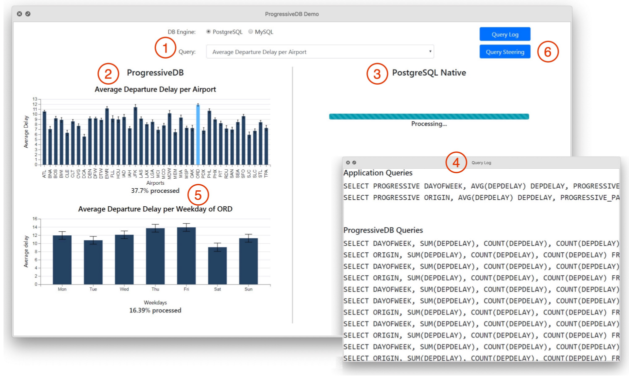
A typical use case for this is that the client initially queries the progressive view for only those groupings and conditions which are *not* marked as FUTURE. The user then interacts with the visualisation of the received progressive query result, for example by drilling down in one of the FUTURE grouping attributes. The application therefore sends a new query to the view which indicates which FUTURE grouping attribute should be used. Since *ProgressiveDB* caches the intermediates of a previous query, it can then immediately return an approximate drill-down result at the confidence level of the state where the initial query was, which it continuous to progressively update while further sub-queries are executed. This avoids the latency and overhead of re-sending all previous sub-queries with a new group-by clause.

4. DEMONSTRATION SCENARIO

To demonstrate the capabilities of *ProgressiveDB*, we use the well-known airline-on-time database [3]. We have built a small data exploration application with an interactive web interface on top of *ProgressiveDB* which allows to analyse flight delay information as shown in Figure 2a. We populated a PostgreSQL and a MySQL database with all flight data for the 40 largest US airports, resulting in a total of ca. 11 GB data on disk. The following two scenarios are also shown in our supplementary demo video.

4.1 Scenario 1: Progressive Queries

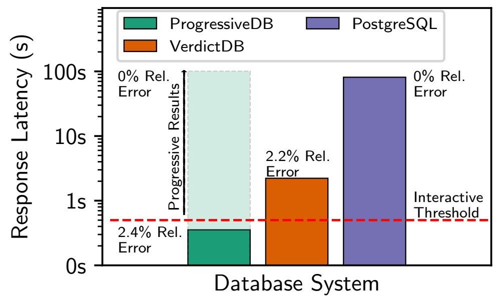
The first use case demonstrates the progressive query execution of *ProgressiveDB*. As shown in Figure 2a, the user first selects one of the pre-defined queries from the dropdown menu ①, which then gets executed concurrently in two modes: The left-side of the window shows the result


ProgressiveDB Demo
DB Engine: [✓ PostgreSQL] MySQL
Query Log
① Query: Average Departure Delay per Airport
Query Steering ⑥
② ProgressiveDB
Average Departure Delay per Airport
Average Delay
Airports
37.7% processed ⑤
Average Departure Delay per Weekday of ORD
Average delay
Mon Tue Wed Thu Fri Sat Sun
Weekdays
16.39% processed
③ PostgreSQL Native
Processing...
④ Query Log

```
Application Queries
SELECT PROGRESSIVE DAYOFWEEK, AVG(DEPDELAY) DEPDELAY, PROGRESSIVE
SELECT PROGRESSIVE ORIGIN, AVG(DEPDELAY) DEPDELAY, PROGRESSIVE_PA

ProgressiveDB Queries
SELECT DAYOFWEEK, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY)
SELECT ORIGIN, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY) FR
SELECT DAYOFWEEK, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY)
SELECT ORIGIN, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY) FR
SELECT DAYOFWEEK, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY)
SELECT DAYOFWEEK, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY)
SELECT ORIGIN, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY) FR
SELECT DAYOFWEEK, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY)
SELECT ORIGIN, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY) FR
SELECT DAYOFWEEK, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY)
SELECT ORIGIN, SUM(DEPDELAY), COUNT(DEPDELAY), COUNT(DEPDELAY) FR
```

(a) Screenshot of Usage Scenario 1



(b) Response Latencies

Figure 2: Demo Screenshot (left) and Latency Comparison of different Systems (right)

of the progressively executed query ②, while the right-side of the window shows the result of running the analytical directly on the underlying database ③. Attendees will see that while they get an immediate feedback from the progressive query, which also constantly updates with the latest flight statistics until the result is exact, they have to wait for a response on the right-hand side.

The native execution on the underlying database only returns a result once the query has completed on the whole dataset, which typically means a delay much worse than the targeted 500ms interactive latency. This can also be seen in the graph of Figure 2b, which compares the first-response latency of *ProgressiveDB* over PostgreSQL with native PostgreSQL for the same query. Just for comparison – not part of the demo – we also included the first response time of VerdictDB over PostgreSQL (which uses a pre-computed sample) on the same dataset. The progressive query is executed by *ProgressiveDB* with a series of sub-queries. These internal sub-queries can be inspected by attendees using the 'Query Log' button of the user interface ④.

While the progressive query is executing, attendees can also click on the result bar of any airport to start a second query which analyses the flight delay information at this airport, but now shown per day of the week ⑤. The progressive result of this second query is shown below the chart of the first query, and it is also compared to a native execution. Attendees can note that the second query, while starting immediately after being selected, does run longer and with an initial less accurate approximation because without query steering a new separate query is initiated which has to analyse the data from scratch. We show the effects of query steering in the second scenario below.

Finally, the user interface also allows attendees to switch between different backend database engines (i.e., PostgreSQL and MySQL) to demonstrate the flexibility of the middleware approach of *ProgressiveDB*.

4.2 Scenario 2: Query Steering

The second use case demonstrates the query steering capabilities of *ProgressiveDB*. While the previous use case showed how *ProgressiveDB* allows interactive response times of single progressive queries, it also demonstrated that refining a query results in an initial loss of accuracy due to the restart of a new query from scratch. However, in many applications the potential refinements of a query are predetermined by the user interface, and hence *ProgressiveDB* introduces the concept of a progressive view which computes an approximate query result suitable for different queries.

Our demo application therefore includes a second view ⑥, which allows query steering using a progressive view (`CREATE VIEW... GROUP BY origin, dayofweek FUTURE`). This progressive view helps to drive a cross-filter visualisation where users can select different airports on the first visualisation, which then instantly affects the output on the second chart which represents the progressive query result of a refined query. The benefit here is that any of the refined queries starts at the same progress and accuracy level as those of the first query, because it can rely on the intermediate results in the progressive view. This also results in less load on the backend database systems as only one sequence of sub-queries has to be executed.

5. SUMMARY

ProgressiveDB provides three main innovations: Firstly, a middleware that transforms any SQL database into a progressive database. Secondly, ProgressiveSQL as a simple to use query interface for interactive applications. Finally, a progressive view mechanism that extends ProgressiveSQL by efficient query steering capabilities.

Acknowledgements

We thank Schloss Dagstuhl and the organizers of Seminar 18411 for the opportunity and motivation to pursue the work reported in this paper.

6. REFERENCES

[1] S. Agarwal et al. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[2] E. Begoli et al. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *ACM SIGMOD*, 2018.

[3] Bureau of Transportation Statistics. ASA 2009 data expo. `http://stat-computing.org/dataexpo/2009/`.

[4] J. M. Hellerstein et al. Online aggregation. In *ACM SIGMOD*, pages 171–182, 1997.

[5] F. Li et al. Wander join: Online aggregation via random walks. In *ACM SIGMOD*, pages 615–629, 2016.

[6] Z. Liu et al. The effects of interactive latency on exploratory visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2122–2131, 2014.

[7] G. Luo et al. A scalable hash ripple join algorithm. In *ACM SIGMOD*, pages 252–262, 2002.

[8] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *ACM SIGMOD*, 2018.