



Hewlett Packard  
**Labs**

## Fast Analytics in Real-time Operations Management

Alkis Simitsis, Kevin Wilkinson, Olga Poppe

Hewlett Packard Labs  
HPE-2016-93

### **Keyword(s):**

Hybrid database; analytics; streaming; updates; column-store

### **Abstract:**

Real-time operations management is a challenging application domain for a general-purpose database management system. Principally, it requires continuously monitoring a high-speed event stream for patterns of interest. It also needs to support complex, analytics queries over new and historical data for problem diagnosis and prediction. Additionally, it also requires support for efficient update since the monitored environment undergoes frequent change. Thus, it has the seemingly conflicting demands of a streaming engine, an analytics engine, and a transaction processing engine. This paper advocates that a single engine cannot satisfy these conflicting demands and presents a hybrid system to address them. Events are ingested by a main-memory database engine that processes requests over recent data. Events are then forwarded to an analytics database engine that processes queries over historical data. A federation layer provides a single interface for applications, parses a request for its data needs and redirects it to one engine or both. We present our system architecture, how specific challenges are addressed using two design optimizations, and performance results showing the feasibility of our approach.

External Posting Date: October 25, 2016 [Fulltext]  
Internal Posting Date: October 25, 2016 [Fulltext]

# Fast Analytics in Real-time Operations Management

Alkis Simitsis  
Hewlett Packard Labs  
Palo Alto, CA, USA  
alkis@hpe.com

Kevin Wilkinson  
Hewlett Packard Labs  
Palo Alto, CA, USA  
kevin.wilkinson@hpe.com

Olga Poppe  
Worcester Polytechnic Institute  
Worcester, MA, USA  
opoppe@wpi.edu

**Abstract**—Real-time operations management is a challenging application domain for a general-purpose database management system. Principally, it requires continuously monitoring a high-speed event stream for patterns of interest. It also needs to support complex, analytics queries over new and historical data for problem diagnosis and prediction. Additionally, it also requires support for efficient update since the monitored environment undergoes frequent change. Thus, it has the seemingly conflicting demands of a streaming engine, an analytics engine, and a transaction processing engine. This paper advocates that a single engine cannot satisfy these conflicting demands and presents a hybrid system to address them. Events are ingested by a main-memory database engine that processes requests over recent data. Events are then forwarded to an analytics database engine that processes queries over historical data. A federation layer provides a single interface for applications, parses a request for its data needs and redirects it to one engine or both. We present our system architecture, how specific challenges are addressed using two design optimizations, and performance results showing the feasibility of our approach.

## I. INTRODUCTION

Real-time operations management involves event streams, transaction processing, and analytics. As an example, consider the oil and gas industry which collects massive amounts of real-time data from a large deployment of sensors in oil wells and related infrastructure (see Figure 1). These sensors emit physical quantities such as temperature, pressure, and flow rate, which are monitored by continuous queries that track the system and issue alerts when problems arise [11]. Typically, these are sliding window queries that compute correlations and trends over recent data and optionally compare the results against known patterns of behavior based on historical data. One solution to this problem is to buffer and process event data in main-memory, e.g., using a stream engine. Then, recent data may be queried in real-time. Older data is eventually discarded or shipped to a separate, disk-based engine for analysis.

But consider what happens when a problem is detected; e.g., a pressure reading from a valve exceeds some threshold. To investigate this, an operations manager might issue various ad-hoc queries, e.g., previous pressure readings for the same valve (trends), other measures from the same valve or from up-stream or down-stream devices (correlations). But these ad-hoc queries may interfere with ingest and processing of new, incoming sensor data. In addition, data of interest may not be available on the engine that detected the threshold violation; e.g., relevant data may have already been evicted. Or, up-stream and down-stream sensor data may be processed by a

different stream or CEP engine. So, the manager may also have to query other engines. Hence, such a support for real-time monitoring would require limiting the scope for real-time analysis of any problems that may be detected.

Alternatively, we could use an analytics engine for real-time, concurrent processing of multiple, often complex, analytics queries, while ingesting events at a high rate. Although an analytics engine can support complex, ad-hoc querying over historical data (a feature typically missing in stream engines), the event data must first be loaded to the engine. Broadly speaking, analytics engines support trickle loading for immediate inserts or updates and bulk data loading for batch inserts. Trickle loading, intended for infrequent data changes, has high overhead and so cannot be used in this context. Bulk data loading is efficient for inserting large amounts of data, but it incurs significant delay because events are assembled into batches prior to load. This extra latency means the event is not immediately queryable in the analytics engine and thus, near real-time analysis is hard to guarantee.

Another problem is the impact of database updates on query processing. In some streaming applications, the modeled application environment is static or slowly changing; e.g., trades in equities markets. However, in real-time operations management, the underlying physical system may be frequently evolving. Sensors are added or removed, some of them may be faulty, devices come on-line or are shut-down, new infrastructure is introduced, objects are physically relocated, and so on. In addition, as sensor data sent over a wireless connection and often from moving objects, the incoming event data has an error rate and once detected recorded data may need to change or be removed. Thus, the event stream is more than just a time series of metrics from individual sensors. It also includes updates and occasional deletes. And these may also affect the continuous queries monitoring the system. In a large physical system in the context of the Internet-of-Things, the update rate could be quite high and must be managed.

We believe a single system, either a stream or a database engine, provides limited support for the conflicting demands of real-time operations management. Instead, we propose to leverage and combine the advantages of each system and present a hybrid database architecture comprising a main-memory database engine coupled to an analytics database engine. The main-memory engine ingests events and processes requests over recent data. It forwards older events to the analytics engine for historical queries. A federation layer

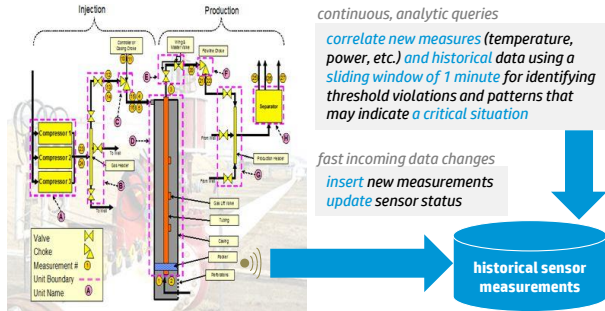


Fig. 1. Example application

simplifies operations managers’ job by providing them with a single interface to the underlying engines. It analyzes incoming queries to determine the time-range of requested data and forwards the query to the the appropriate engine(s). If the time-range includes both recent and historical data, the query is split into fragments for both engines and the results are merged. For example, a continuous query (e.g., monitoring) that presumably needs only recent data, is processed by the main-memory engine. Such a query may compute “the maximum and a moving average for temperature and power per valve per well per minute” over a sliding window of one minute (Figure 1). An analytics query that studies an event from last month (e.g., prediction) would be processed by the analytics engine. A real-time query involving data over the last hour (e.g., diagnosis) would presumably be processed by both engines. However, in all cases the application is unaware of the data location.

There are several challenges in designing and building such a hybrid system. Applications must have a consistent view of the data, despite storing the data across two engines. Data must be efficiently moved from the main-memory engine to the analytics engine with low latency and low impact on existing queries. Database updates should not degrade query performance. In this paper, we present *CuteDB*, a system for real-time operations management. We identify update and delete operations as a key performance challenge and describe two optimizations to handle them. We present performance results showing the feasibility of our solution.

In what follows, Section II presents a system overview. Section III describes the federation layer’s functionality. Section IV describes techniques to synchronize the two engines. Section V presents experiments and finally, the last two sections discuss the related work and conclude the paper.

## II. SYSTEM OVERVIEW

We first present the problem statement. Then, we sketch our approach for handling database updates. Next, we provide an overview of the system architecture.

### A. Problem Statement

We monitor a series of events emitted by a physical system. In database terms, this event stream is captured by inserting each event as a row in a table that includes a column with the time of the event. Thus, we assume a workload comprising a stream of insert operations (i.e., new data events), queries, as well as update and delete requests. All are expressed in

SQL-like form (see Section III). A query may be discrete or continuous. We consider requests over a data stream and, without loss of generality, we assume query and data change requests specify a window of relevant data; e.g., a window can be added to a query by specifying a time range. The window may be time based or count based. A request may reference non-streaming tables (e.g., meta-data about infrastructure) as well as stream data but, to simplify the discussion, we assume a request references at most a single data stream.

Time window semantics defined over a time range (similarly, for count windows) are transformed to predicates; e.g., ‘Window Time Interval  $tstamp_1$   $tstamp_2$ ’ can be rewritten as ‘ $tstamp_1 > ts$  AND  $ts < tstamp_2$ ’, where  $ts$  is a Timestamp. The queries and data changes are ordered by their arrival time. Figure 2 illustrates data arrival. Data that arrived between now,  $t_n$ , and  $t_b$  are kept in the main-memory engine (hereafter, *stream buffer* or *buffer*) and data older than  $t_b$  are stored in the analytics engine (hereafter, *database*), where the oldest data has timestamp  $t_h$ .  $t_b$  is the timestamp of the oldest value in the buffer. As we move data from the buffer to the database,  $t_b$  is updated accordingly. Buffer flush may be policy-based (e.g., based on dynamic workload analysis or on resource utilization) rather than a strict, inflexible design choice (e.g., memory size) as in a typical streaming engine.

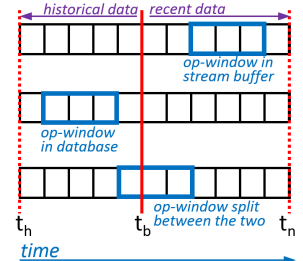


Fig. 2. Operation windows

A request, either a query or a data change, is analyzed based on its window. If a request relates only to recent or only to historical data is sent to the corresponding engine (see the left and right blue boxes in Figure 2). Inserts are sent to the stream buffer, since they involve recent data. Updates, deletes, and queries are analyzed and may be split for recent and historical data. In such cases, we rewrite the request by splitting its window into two windows: one applied to the stream buffer and one to the database. We explain this in Section III.

Our performance study in Section V reveals that both the main-memory engine and the analytics engine provide good performance for queries and insert requests. However, update and delete requests are the scourge of performance for both engines as they force requests to be serialized. Even a small number of update operations have a drastic impact on throughput and response time and thus, they require a special care.

### B. Processing Updates

Update requests arrive as part of a stream workload and may affect recent data, historical data or both. Additionally, an update may conflict with already executing continuous queries. To reduce the performance impact of updates, we employ the following two optimizations.

*Optimization 1 - Defer operations.* We prioritize requests according to their arrival time and dependencies to each other. A key observation is that if a certain data change affects data not related to a known query, then it can be deferred until

either a new query depends on it or the engines have free cycles to process it. Until then, we postpone this requests and all subsequent requests arrived after it, if they depend on it.

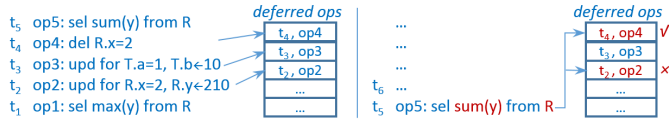


Fig. 3. Deferring operations example

For example, assume the series of operations shown in Figure 3 (in pseudo-SQL). At time  $t_1$ , we execute  $op_1$ . Then, all data changes that follow,  $op_2$ ,  $op_3$ , and  $op_4$ , can be deferred. For that, we keep them in a deferred-operations buffer (DO). When a new operation enters DO, we check for possible conflicts. At time  $t_4$ , we identify that  $op_2$  and  $op_4$  intersect and mark  $op_2$  as invalid as  $op_4$  supersedes it. At time  $t_5$ , a new query  $op_5$  arrives, and before executing it, we need to guarantee that it sees a consistent state. Thus, we visit DO and identify the operations relevant to  $op_5$ , which here are  $op_2$  (invalid) and  $op_4$  (active). Nothing else in DO can be relevant; if it were, then it would have been executed just before  $op_1$ . So, we run first  $op_4$  and then  $op_5$ . Periodically, a data move method discards invalid entries from DO and flushes all or part of its content to the stream buffer or the database. Technically, the data in both the stream buffer and the database are not consistent at all times, but our analysis guarantees that all queries see a consistent state; thus, in our case, consistency comes on-demand.

*Optimization 2 - Convert updates/deletes to inserts.* Analytics engines support very efficient bulk insert, but being read-optimized, they do not excel at efficiently processing updates and deletes. Updates and deletes cannot be processed in bulk. They must be processed as individual statements, which has high overhead. Also, they lock data, which blocks other requests from proceeding and so degrades performance. However, if we are willing to modify the physical database schema, we can convert update operations to insert operations and avoid this performance penalty. This is done by converting the table to be updated from update-in-place to a versioned table by adding a timestamp column (the time of the update) and a delete flag (set to true when the row is deleted). To fetch the current attribute values for an object, we find all rows that match the object’s primary key and return the row with the largest timestamp. If that row’s delete flag is true, null is returned. With this technique, updates and deletes can be encoded as insert operations so that the efficient bulk loader can be used. Note that a view can be defined over the versioned table to make it appear to applications as a normal (update-in-place) table. The conversion of update and delete to insert operations is performed by the Data Mover (explained shortly).

### C. Hybrid Database System Architecture

We implemented a prototype system, *CuteDB*. In our current implementation, we use a commercial, transaction-processing, in-memory database, as the stream buffer and a commercial

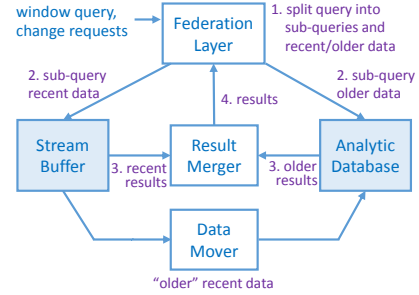


Fig. 4. CuteDB architecture

parallel column-store database, as the analytics engine. Figure 4 illustrates a high-level overview of *CuteDB*. There are two inputs to the system: window queries and change requests, i.e., insert, update, and delete requests. These come in the form of SQL-like statements (i.e., regular SQL syntax augmented by window semantics). We assume that *CuteDB* is the single entry point for all statements, either queries or change requests; i.e., it controls the workload on both databases. The Federation Layer (see Section III) parses a statement, determines whether it involves recent or historical data, and decides when and how to execute it. After query execution, the Result Merger collects the query results and does the final merge (see Section III-C). Periodically, the Data Mover moves data from the buffer to the database, by batching rows from tables in the buffer and loading them to tables in the database using fast bulk loading (see Section IV). Next, we describe these modules in detail.

## III. FEDERATION LAYER

When a statement, i.e., a new window query or a new change request, arrives at the Federation Layer, then *CuteDB* performs two tasks (see Figure 5): it *parses* the statement (Parser) and *splits* it into two fragments involving recent and historical data, respectively (Splitter). A scheduler module (Scheduler) queues statements ready for execution; e.g., it provides a ready queue for queries. As an optimization, the scheduler also decides whether or not a request should be *deferred*. Note, that all requests are logically executed in their arrival order; i.e., even if a request is deferred, it is given a view of the data as of its arrival time. At the end of query execution, the Result Merger propagates the results to the federation layer, where a simple reporting tool forms and returns them to the application. We explain these tasks next.

### A. Statement Parsing and Splitting

Input statements are expressed in a SQL-like form. This is standard SQL extended by clauses representing window semantics for queries and change requests. For ease of presentation, in this section we consider single-table requests and simple, atomic predicates [2]. For an extension to more general requests and a detailed description of the language details, we refer the interested reader to [12].

*CuteDB* parses an input statement and identifies its semantics, e.g., the relations, attributes, predicates, and operations involved. Statement parsing is a relatively straightforward process. Since we currently use SQL interface and databases

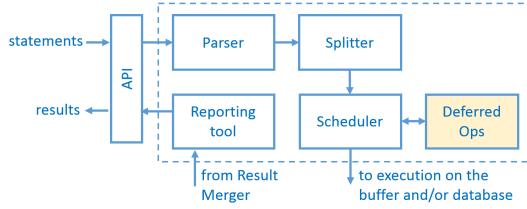


Fig. 5. Federation layer tasks

TABLE I  
EXPRESSING WINDOWS IN SQL

Window	Update
Time-based interval $\{t_1, t_2\}$	UPDATE $R_u$ SET $S$ WHERE $P_u$ AND $t_1 \circ \text{stamp}$ AND $\text{stamp} \circ t_2$
Count-based interval $[c_1, c_2]$	UPDATE $R_u$ SET $S$ WHERE $P_u$ AND $K$ IN (SELECT $K$ FROM $R_u$ WHERE $\text{stamp} \leq \text{ustamp}$ ORDER BY $\text{stamp}$ DESC LIMIT $c_2 - c_1 + 1$ OFFSET $c_1 - 1$ )

Legend:  $R_u$ : a relation;  $P_u$ : a set of predicates;  $S$ : a set of assignments ‘attribute = constant’;  $\text{stamp}$ : a row timestamp;  $\text{ustamp}$ : the time point when the update was issued;  $K$  is the set of primary key attributes;  $t_1, t_2$ : timestamps,  $t_1 \leq t_2$ ; and  $\circ = <$  or  $\leq$  for open, closed windows, respectively.

do not lend themselves nicely to window semantics, the parser translates the operation windows to standard SQL by converting the windows into predicates and clauses. Table I shows an example translation of an update with window semantics (e.g., ‘Window Time Interval  $\text{stamp}_1 \text{stamp}_2$ ’) to standard SQL. We work similarly for other statements too.

An operation window can be defined over a range of values (e.g., ‘Range 1 hour’) or using finer-grained predicates (e.g., ‘time < now() and time > now() - 2 hours’). Table II shows possible time windows. *CuteDB* analyzes an operation window and decides whether a statement needs to be split. In doing so, *CuteDB* takes under consideration the current  $t_b$  just before an operation’s execution and directs the statement or the statement fragments as needed (see also Figure 2).

*CuteDB* converts continuous queries to a series of very frequent discrete queries. In a sense, a continuous query is itself a kind of deferred operation as repeated invocations are deferred to the future. Continuous queries are registered to the scheduler, which resubmits them at a specified frequency, which is a tunable parameter.

### B. Deferring Operations

The benefit in *deferring operations* is more evident as the incoming data rate increases; then, postponing less urgent requests increases system throughput. This is like load shedding in stream engines (e.g., [14]) in that a resource-intensive request is modified to use fewer resources. Except here, the request and data are fully processed; we drop nothing.

Next, we describe when we may defer deletes and updates without sacrificing query correctness or violating primary key constraint. (The formal definitions can be found in [12].) We call *urgent operations* the delete and update operations that cannot be postponed and need to be executed. However,

TABLE II  
WINDOW CLASSIFICATION

Type	Syntax	Contents
Discrete recent time-based	Range $d$	Events within most recent time interval of length $d$
Continuous recent time-based	Range $d$ Slide $s$	Events within recent time interval of length $d$ , window slides when new events within the time interval of length $s$ become available
Discrete time-based	Time-based interval $\{t_1, t_2\}$	Events with time stamps between $t_1$ and $t_2$

Legend:  $d, s$ : time durations;  $t_1, t_2$ : timestamps,  $t_1 \leq t_2$ ;

urgent operations may depend on other, previously deferred operations. Hence, we analyze operation interdependencies to ensure completeness of the operation set that can no longer be postponed. Note, that we defer only update and delete operations, and not inserts, as delete and update are time-hog operations, whilst inserts are very efficient with bulk loading.

As an implementation detail, we keep the deferred operations in an in-memory hash structure. As new operations arrive, we compare them to existing ones (explained shortly). When a deferred operation is ready for execution, we do not parse it again; we simply dispatch it to the appropriate engines.

1) *Query correctness despite deferred operations*: A deferred operation has no effect on a query result set if:

- 1) The deferred operation does not access the same data as the query (Section III-B2).
- 2) All data relevant for the query was deleted or became irrelevant and, thus, we can conclude that the query has no results without evaluating it (Section III-B3).
- 3) The query can be modified such that it returns the same results as if the deferred operation had been performed prior to query execution (Section III-B4).

All other deletes and updates cannot be deferred. They are applied to the data within the query window before the query is evaluated (Section III-B5).

2) *Irrelevant deletes and updates*: If a delete/update does not access the same data as a query, then it is irrelevant for it.

**Irrelevant deletes.** A delete is irrelevant for a query if: (a) they access different relations; (b) their windows do not overlap; or (c) their predicates contradict. Figure 6 illustrates the latter case. The delete and the query access the same relation and the query window (not shown in the SQL code) contains the delete window. But, their predicates are mutually exclusive. The query computes the average temperature value among all temperature measurements exceeding 100 degrees per sensor. The delete removes all temperature measurements which are less than 10 degrees. Thus, they never access the same tuples.

**Irrelevant updates.** An update is irrelevant for a query if: (a) they access different relations; (b) their windows do not overlap; (c) their predicates contradict and no updated tuple satisfies the query predicates; or (d) the update sets no attributes accessed by the query. Figure 7 shows the last two cases. The left update modifies tuples irrelevant for the query before the update (since the update and query predicates are mutually exclusive) and which remain irrelevant for it after the update (as no tuple with  $\text{area}=B$  matches the query predicate

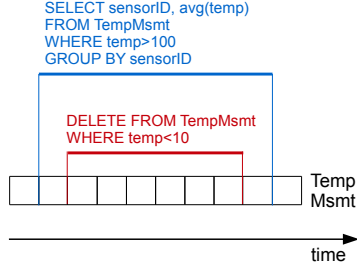


Fig. 6. Irrelevant delete

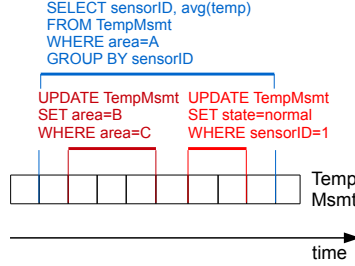


Fig. 7. Irrelevant updates

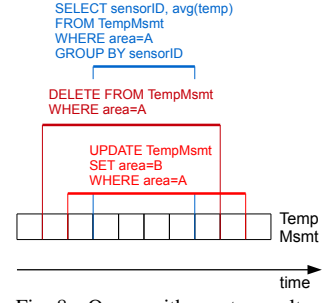


Fig. 8. Query with empty result set

$area=A$ ). The effect of the right update is irrelevant for the query since the query does not access the attribute  $state$ .

3) *Queries with empty result sets*: A query would return no results if all relevant tuples have been either deleted or become irrelevant. Such a query can be omitted and the respective delete or update operation can be deferred.

**Deletion of relevant tuples.** A delete deletes all tuples relevant for a query if the delete window contains the query window and the delete predicates subsume the query predicates.

**Update of relevant tuples.** An update makes all tuples relevant for a query irrelevant for it if the update window contains the query window, the update predicates subsume the query predicates and no updated tuple satisfies the query predicates.

Figure 8 shows two examples of a query with an empty result set. The delete removes all tuples relevant for the query within the query window. The update makes all tuples relevant for the query irrelevant for it within the query window.

4) *Query modification*: We describe now under what conditions a query can be modified to return the same results as if a deferred operation was performed prior to query execution.

**Query window modification.** Tuples relevant for a query are deleted or become irrelevant for it within *part* of the query window if: (a) the query and delete windows overlap (but the delete window does not contain the query window) and the delete predicates subsume the query predicates; or (b) the query and update windows overlap (but the update window does not contain the query window), the update predicates subsume the query predicates and no updated tuple satisfies the query predicates. In this case, the query window  $W_q$  is modified to exclude the delete or update window  $W_{d/u}$ . That is,  $(W_q - W_{d/u})$  is the new query window.

In Figure 9, the delete would remove all tuples relevant to the query within part of the query window. Thus, the query window is modified to disregard the deleted tuples and to consider only the relevant ones (shown as striped boxes).

**Query predicates modification.** Tuples relevant for a query are deleted or become irrelevant for it within the *entire* query window if: (a) the delete window contains the query window and the delete and query predicates do not contradict each other (but the delete predicates do not subsume the query predicates); or (b) the update window contains the query window, the update and query predicates do not contradict each other (but the update predicates do not subsume the query predicates) and no updated tuple satisfies the query predicates. In this case, the query predicates  $P_q$  are extended

by the opposite of the delete or update predicates  $P_{d/u}$ . That is,  $(P_q \wedge \neg P_{d/u})$  are the new query predicates.

In Figure 10, the delete deletes those tuples relevant for the query for which  $temp > 1000$  holds within the entire query window. Thus, the query predicates are modified (shown in the figure with green color) to disregard those deleted tuples.

**Query window and predicates modification.** This case combines the above two cases. Tuples relevant for a query are deleted or become irrelevant for it within part of the query window if: (a) the query and delete windows overlap (but the delete window does not contain the query window) and the delete and query predicates do not contradict to each other (but the delete predicates do not subsume the query predicates); or (b) the query and update windows overlap (but the update window does not contain the query window), the update and query predicates do not contradict to each other (but the update predicates do not subsume the query predicates) and no updated tuple satisfies the query predicates.

In this case, the original query is replaced by two new queries. The first query is evaluated on the part of the original query window  $(W_q - W_{d/u})$  that is not affected by the delete or update operation. Thus, this query maintains the original query predicates  $P_q$ . The second query is evaluated on the part of the original query window  $(W_q \wedge W_{d/u})$  that is affected by the delete or update. Hence, this query has predicates  $(P_q \wedge \neg P_{d/u})$ , which disregard deleted or updated tuples. (The results of these queries need to be combined as we see shortly.)

In Figure 11 the delete would remove the relevant to the query tuples for which  $temp > 1000$  holds within part of the query window. Thus, the original query is replaced by two queries, whose windows cover the original query window and their predicates depend on whether their respective windows are affected or not by the delete. The query on the left has the window (vertically striped) that is affected by the delete. So, its predicates are modified (in green color). The query on the right has the window (horizontally striped) that is not affected by the delete. So, this query maintains the original predicates.

5) *Update window modification*: Summarizing, a delete can be *indefinitely postponed* with respect to a query, as the conditions under which a delete can be deferred cover all possibilities. On the other hand, an update can be postponed with respect to a query if it is either irrelevant for the query or it has the same effect as deletion of relevant tuples for the query. All other updates should be executed prior to query execution. Still, such updates need to be applied only within

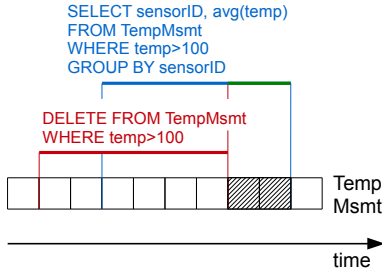


Fig. 9. Window modification

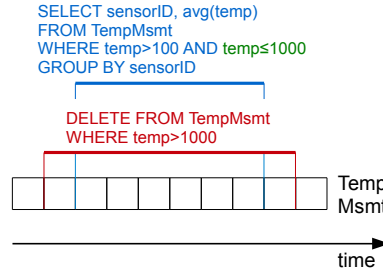


Fig. 10. Predicates modification

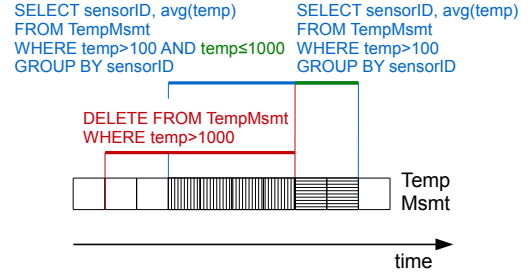


Fig. 11. Window and predicates modification

the query window. To reduce the number of operations that can no longer be deferred because an urgent update depends on them (Section III-B7), only an update with window ( $W_q \wedge W_u$ ) should be executed before the query, while an update with window ( $W_q - W_u$ ) may be postponed.

For example, Figure 12 illustrates the case of update window modification. Data in the striped window gets updated prior to query execution. Data outside the striped window is irrelevant for the query and, thus, can be updated later.

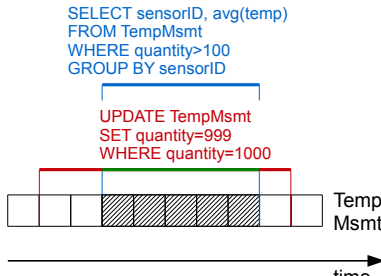


Fig. 12. Update window modification

#### 6) Valid primary key constraint despite deferred operations:

**Overview.** Next, we examine when a delete or an update operation can be delayed with respect to inserts and urgent updates without violating any primary key constraint. A deferred operation has no effect on a primary key constraint if:

- 1) It does not update any primary key attribute value, does not access the same relation as an insert or an urgent update, and does not delete tuples with the same primary key values as the inserted and updated tuples.
- 2) The deferred delete invalidates the insert; i.e., it deletes all tuples inserted by the insert.
- 3) The insert operation can be modified such that it has the same effect as if the deferred delete was applied to the data prior to the insert.

All other deletes and updates may affect primary key constraints and thus, they cannot be deferred.

Let us now examine the three cases in more detail.

**Irrelevant deletes and updates.** An update has no effect on primary key constraint if it does not set any primary key attribute values. Assume that  $msmtID$  is the primary key of a relation  $TempMsmt$ . The first update below cannot be deferred, whilst the second could if the first one (or any other urgent operation) were not depended on it (Section III-B7).

```
UPDATE TempMsmt SET msmtID=5 WHERE quantity<10
UPDATE TempMsmt SET quantity=5 WHERE msmtID<10
```

A delete can be postponed with respect to an insert or an urgent update without affecting primary key constraint if: (a) they access different relations; or (b) the delete specifies mandatory constraints on primary key attribute values and all inserted or updated tuples violate these constraints (before and after the update). Constraints on primary key attribute values are mandatory, since otherwise the delete might remove tuples whose primary key values coincide with those of newly inserted or updated tuples. Such delete cannot be deferred.

For example, the delete below does not remove tuples with the same primary key value as the inserted and updated tuples.

```
INSERT INTO TempMsmt (msmtID, quantity, tstamp)
VALUES (11, 2, '2014-09-01 10:00:00.00'),
      (14, 5, '2014-09-01 11:00:00.00')
UPDATE TempMsmt SET msmtID=5 WHERE msmtID<10
DELETE FROM TempMsmt WHERE msmtID>20
```

**Invalidated insert.** A delete invalidates an insert if the insert was issued not later than the delete, all inserted tuples are within the delete window and satisfy the delete predicates. The delete below removes all inserted tuples by the above insert. So, the insert is omitted and the delete is postponed.

```
DELETE FROM TempMsmt WHERE quantity<10
WINDOW Time-based interval ['2014-09-01
9:00:00.00', '2014-09-01 12:00:00.00']
```

**Insert modification.** A delete invalidates an inserted tuple if the insert was issued not later than the delete, the delete specifies mandatory constraints on primary key attribute values, the tuple is within the delete window and satisfies the delete predicates. Such a tuple may be dropped from the insert operation. Constraints on primary key attribute values are mandatory since otherwise the delete might remove tuples the primary key values of which coincide with those of newly inserted tuples. Such a delete cannot be deferred.

The delete below removes the first inserted tuple by the above insert. So, this tuple is dropped from the insert request.

```
DELETE FROM TempMsmt WHERE msmtID<13
WINDOW Time-based interval ['2014-09-01
9:00:00.00', '2014-09-01 12:00:00.00']
```

We work similarly to ensure validity with respect to other constraint types.

7) *Dependencies of urgent deletes and updates:* Urgent deletes and updates may depend on other operations, which then can no longer be deferred. The analysis for such interdependencies is similar to the analysis made for queries. So, a

deferred delete/update has no effect on an urgent operation if:

- 1) The deferred operation does not access the same data as the urgent operation (Section III-B2).
- 2) The urgent operation is invalidated by a deferred operation that deletes or makes irrelevant all data relevant for the urgent operation (Section III-B3).
- 3) The urgent operation can be modified such that it would have the same effect as if the deferred operation had been applied to the data (Section III-B4).

All other deletes/updates cannot be deferred. These should be applied to the data within the window of the urgent operation before the operation is executed (Section III-B5).

8) *Correctness and performance issues:* For ease of presentation, we have omitted here a formal description of the deferring operations technique. For formal proofs on correctness and termination, we refer the interested reader to [12], where we show that: (a) operation modification is correct; (b) query results are correct despite postponed operations; and (c) primary key constraint holds despite postponed operations. We also provide upper bounds on the number of deferred operations and on the number of comparisons provoked by a new, incoming operation. Empirically, however, the average total cost of deferring operations (e.g., adding, retrieving, comparing deferred operations) were negligible to the average total time spent for a single statement, as most time goes to statement batching, dispatching, and execution.

### C. Merging Query Results

After query execution, Result Merger combines the query results and returns them to the application through the federation layer (see Figure 5). When a query is sent only to the buffer or the database, it is evaluated there and the result is sent to the application. When a query splits to the buffer and the database, the two results are merged. In our current implementation, a query performing simple aggregate calculations (like max, min, count, sum), is partially evaluated on both and then the partial results are collected and merged by the Result Merger. For more complicated computations, we collect the corresponding, partial rowset from the stream buffer and send it via Data Mover to the database, where first, the recent data is merged with the historical data and then the query is evaluated.

As we explain in the experiments, the buffer is a better fit for shorter queries targeting a smaller range of data values. These queries can be an order of magnitude slower when running in the database. On the other hand, more complex, analytics queries run much faster in the database, which is tuned for such queries. Based on this observation, we tune how frequently the Data Mover runs, as we discuss next.

## IV. DATA MOVER

The Data Mover (DM) synchronizes the database at regular, periodic intervals or on-demand, when needed. Assume its previous invocation occurred at time  $t_b$ . Let the current time be  $t_n$ . Then, when DM is next invoked, the buffer contains newly inserted data from time  $t_b$  to time  $t_n$ . Similarly, there may be deferred data change requests that arrived between those

times. The first task of DM is to determine how much data to move. Specifically, it selects a time  $t'_b$  between  $t_b$  and  $t_n$  and moves all data and deferred requests up to that time to the database (Figure 14). When finished, the database is updated and consistent as of time  $t'_b$  and that time becomes the new value of  $t_b$ . Note, we mean *transactionally consistent*, not that the database is consistent with the buffer. The buffer is also transactionally consistent, but as of a future time.

DM transfers new stream data from the buffer to the database using the efficient bulk load command of the analytics engine. To process deferred data change requests, DM has a choice. It may use the update and delete commands of the database. Or it may convert the updates and deletes to inserts. Next, we explain how this is done and also how DM ensures database consistency for both the buffer and the database. Finally, we discuss techniques to select a new move time,  $t'_b$ .

### A. Converting Updates and Deletes to Inserts

Let  $T$  be a table in the database that may be updated. If updates/deletes of  $T$  are infrequent, we may choose not to convert them and process them as individual update/delete commands. If they are frequent, they degrade performance. We can reduce this performance degradation as follows. We require that  $T$  has a primary key that uniquely identifies each row. We store table  $T$  as two physical tables,  $T_c$  and  $T_v$ . Table  $T_c$  is a consistent snapshot of table  $T$  as of some time  $t_c$ . Table  $T_v$  stores (versions of) changes to table  $T$  that occur after time  $t_c$ . To track the changes,  $T_v$  includes two additional attributes, a timestamp and a Boolean delete flag.

When DM processes an update for table  $T$ , it generates an insert for  $T_v$  with the timestamp set to the current time and the delete flag set to false. For a delete on  $T$ , DM generates an insert for  $T_v$  with the delete flag set to true. To retrieve the most recent values for  $T$  in the database, we first group  $T_v$  on its primary key and select the most recent row for each key,

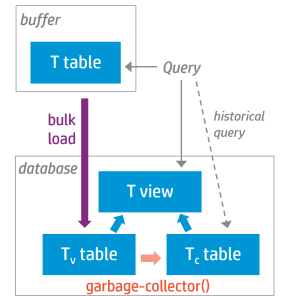


Fig. 13. Data move operation

then we (outer) join the result with  $T_c$  and drop any rows where the delete flag is true. This join can be hidden from application programs by defining table  $T$  as a view over  $T_c$  and  $T_v$ . The join of  $T_c$  and  $T_v$  adds overhead when querying  $T$ . So, applications that do not need the freshest data may choose to query  $T_c$ , which is a consistent snapshot at a previous time. Over time, table  $T_v$  will grow as it accumulates old versions of rows. So, periodically, DM runs a garbage collector process to create a new snapshot table  $T_c$  and truncate table  $T_v$ .

From this basic approach, we employ various optimizations to reduce the time and storage overhead and improve performance. For space considerations, we omit a detailed description. Briefly, note that the deferred updates and deletes may apply to data stream tables or to metadata tables. For deferred updates to the stream tables, it makes sense to include those changes as part of the bulk load for the data stream.

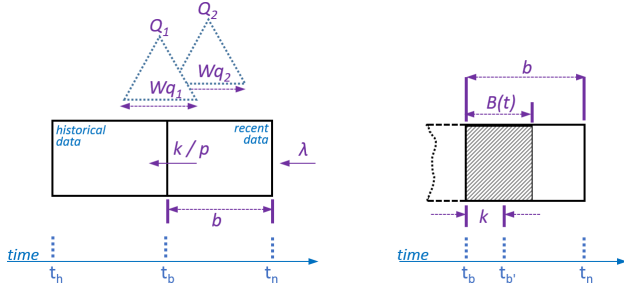


Fig. 14. Data Mover

For deferred updates to metadata tables, DM has a choice of generating insert commands or using a bulk load command for table  $T_v$ . Note that, in either case, the inserted rows require values for all attributes of table  $T$ , not just those updated. But, the original update or delete operation may not specify these values. If DM uses an insert command, it may retrieve the values as part of the command. If DM uses bulk load, in the worst case, it may have to read the values from the database. As another optimization, when the metadata is not large, we cache the current values in the buffer.

### B. Correctness

DM leaves the buffer and the database in a consistent state after it executes. And it ensures that applications see a consistent view of both the buffer and the database while it is executing. It is not necessary to quiesce all activity while DM runs. Consider a split query that is already running when DM begins. DM cannot delete any data from the buffer until the query finishes. However, on the database, DM may be inserting rows to a table,  $T_v$ , that is being read by the query and these new rows should not be visible to the query. To ensure this, the read of  $T_v$  includes a filter to drop any rows with a timestamp greater than  $t_b$ . The situation is the same for a query that starts while DM is executing. Once DM has inserted the new data and processed all deferred operations, it updates the value for  $t_b$ . But, it cannot immediately free the buffer space until all existing requests that may need those rows have terminated. There is a similar delete protocol for the database to garbage collect the version tables,  $T_v$ .

DM operates transactionally. If a failure occurs during a move operation, it may be retried without violating consistency of either the buffer or the database. Details are beyond the scope of this paper.

### C. Invocation

In one extreme, we could leave the recent data in the stream buffer indefinitely (or at least as long as our memory budget allows). But running complex, analytics queries is faster on the analytics engine, while the stream buffer is much faster on simple queries (see Section V). So eventually the data needs to be moved to the database and the question is ‘when’.

Figure 14(left) illustrates the data arrival at the buffer and the data move from the buffer to the database. At present time,  $t_n$ , data arrives at the buffer and get timestamped. Assume that the arrival rate of incoming data is  $\lambda$  and that every  $p$  time units DM moves  $k$  data values from the buffer to the database.

Thus, the data move rate is  $k/p$ . Assume also a buffer size  $b$ . Then, at time  $t \in [t_n, t_b]$ , the buffer contains  $B(t)$  values:

$$B(t) = |(\lambda - \frac{k}{p}) \times t| \leq \theta_b \times b \quad (1)$$

$B(t)$  is bounded by the buffer size weighted by a desired buffer capacity  $\theta_b$ .  $\theta_b$  leaves room in the buffer to accommodate sudden stream bursts in highly dynamic environments and it is a tunable parameter computed as a function of  $\lambda$ . Note that this formula can be straightforwardly adjusted for dynamic arrival rates,  $\lambda(t)$ . For ease of presentation, unless otherwise stated, in our discussion we assume static  $\lambda$  for the period of a single run; but two subsequent runs may have a different  $\lambda$ .

DM starts at time  $t_{DM}$ , which is either at a determined time period (i.e., every  $p$  time units) or at  $t_{full}$  when the buffer has reached the allowed capacity ( $\theta_b \times b$ ), whichever comes first. Hence, DM runs at time:

$$t_{DM} = \min(p, t_{full}), \text{ where } t_{full} = \frac{\theta_b \times b}{|\lambda - \frac{k}{p}|} \quad (2)$$

As an optimization, at time  $t_{DM}$ , we may postpone running DM if  $B(t)$  is below a limit that makes the data move worth it, as each move comes with an overhead cost. This overhead is twofold. It is due to a direct transfer cost for moving data from the buffer to the database. But it is also due to an indirect query execution cost that may arise as we move query windows from recent towards historical data (see also Section IV-A).

Observe Figure 14(left). Assume that there are a number of queries probing the data store as new data arrive at the buffer. Each query  $q_i$  comes with a window,  $W_{q_i}$ , which has a start ( $wq_{i\_start}$ ) and an end ( $wq_{i\_end}$ ). Executing a query in the buffer comes with a cost  $c_{b_i}$  and in the database with a cost  $c_{h_i}$ . Broadly speaking, in real-time operations management there are two classes of queries: those performing simple computations over the incoming data and analytics queries involving historical data too. Based on where the data resides, these queries may be executed in either engine or on both engines. Our experiments (see Section V) show that for simpler queries  $c_{b_i} \ll c_{h_i}$  by at least an order of magnitude, whilst for analytics queries the result is the opposite. This has a direct impact on query response time and freshness. So, based on the query type at hand, we need to tune DM execution frequency to keep more or less data in the stream buffer.

Let us assume the case of the simpler queries, which we would like to run on the stream buffer (we work similarly for analytics queries). *CuteDB* can accommodate all queries with maximum overlapping query window  $W_q$ , as follows:

$$W_q = |\max_i(wq_{i\_start}) - \min_j(wq_{j\_end})| \leq [t_h, t_n] \quad (3)$$

And for executing queries solely in the buffer, their maximum overlapping query window should be in the area  $[t_b, t_n]$ . Thus, for a query  $q_i$  its query window  $W_{q_i}$  should be:  $t_n - t_b \leq W_{q_i}$ , which gives us a lower bound for  $t_b$ , if we need to execute it in the buffer. Deciding on how to execute a multiplicity of queries requiring a combination of recent and historical data as the underlying data stores evolve is essentially a workload management problem. A thorough analysis on this topic is out of the scope of this paper.

Currently, we use a crude estimate for tuning the buffer size taking into account the windows of the queries accessing data in it. Assuming that a bunch of data of size  $\lambda$  arrives at the same time, we estimate an expected  $t'_b$  prior to DM execution as follows. The oldest value in the buffer has timestamp  $t_b$ . When DM runs, the  $t_b$  time gets updated to reflect the new oldest timestamp in the buffer. Observe Figure 14(right). Every  $p$  time units, DM moves  $k$  values, and then, the amount of data in the buffer is  $B(t=p) = |\lambda p - k|$ . Hence, if before a data move the buffer contained  $B(t)$  values whose oldest timestamp were  $t_b$ , after the move completes at time  $t'$ , the buffer contains  $B(t') = B(t) - k$  values and the new  $t'_b$  is the oldest timestamp in the portion of data  $B(t')$  estimated using Equation 4.

$$B(t') = \lambda t - \left(\frac{t}{p} + 1\right)k \quad (4)$$

Using Equation 4 recursively, we estimate the amount of data,  $B(t')$ , in the buffer after a number of data moves, and by looking up the earliest timestamp in the earliest arrived bunch in  $B(t')$ , we get an estimate of  $t'_b$ . With a  $t'_b$  estimate, we regulate the number of queries accessing the buffer (or similarly the database) as a function of DM execution frequency.

In practice, not all  $\lambda$  data of a bunch arrive exactly at the same time, nor all  $k$  data are moved at the same time, due to several contention factors (e.g., system load, network bandwidth, system implementation) that may vary with  $\lambda$  and  $k$  too. Therefore, the above formulas are weighted for contention by adjusting  $\lambda$  and  $k$  as  $\lambda \times c_\lambda(\lambda)$  and  $k \times c_k(k)$ , respectively. For the same reasons, when we use the  $t'_b$  estimate in Equation 4, we also look for the earliest timestamp in the  $B(t')$  bunch.

## V. EXPERIMENTS

### A. Configuration

We used a parallel, column-store database as the analytics engine and a parallel, in-memory transactional database engine as the buffer. *CuteDB*'s components (federation layer, data mover, etc.) were custom code in Java 1.8. The analytics engine was installed on an eight node cluster (each node: 126GB / 40 CPUs, RHEL 6.6). The buffer engine was installed a four node cluster with 10 hosts on each node (each node: 24GB / 16 CPUs, RHEL 6.6) and all unnecessary to our immediate task operations (e.g., logging, replication) were disabled.

Our evaluation is based on a real operations management scenario from the oil and gas industry. Sensor data measuring physical quantities (temperature, pressure, etc.) are streaming in, while they are monitored by 14 continuous queries that issue alerts when problems appear (e.g., threshold violation). At the same time, 12 analytics queries run periodically over the recent and historical data to identify patterns of erroneous behavior and predict potential critical situations. Emitted from sensors and sent over a wireless connection, the incoming event data has an error rate. Of these errors, some are correctable within a certain amount of time (normally a short time period but occasionally longer), in which case the events are updates, and some are not correctable, in which case the events are deleted. Normally the updates would go to events in the buffer. But occasionally, if the delay in updating an event

is long, the event may have migrated to the database. For example, this happens when a sensor is faulty and generates bad data for a time period (e.g., unknown sensor id, out-of-range data value), in which the error might not be immediately obvious. Once the faulty sensor is discovered, it is necessary to go back and delete the bad data or correct it.

The events arrive in CSV format and are converted on-the-fly to SQL insert/update/delete statements. They involve 3-15 numeric attributes. The monitoring queries have a varying selectivity and contain 1-5 aggregate functions (e.g., sum, max, count, avg), a group-by clause, a predicate on 1-5 attributes (plus those needed for the window translation), and some, contain an order clause. The updates have window semantics, change 1-5 attributes, and comprise both infrastructure updates changing metadata tables and also bad reading fixes updating past events. The inserts and deletes have 1-3 predicates and have no subqueries. The analytics queries contain 3-5 joins and 1-4 user-defined functions (UDFs) written in C++.

For simplicity, in the rest and unless otherwise stated, the term updates refers to both update and delete requests.

To evaluate *CuteDB* under various conditions, we generated additional workloads of varying characteristics based on this use case. For example, in the oil and gas scenario the update requests are between 1% and 12%. Started from that, we also generated workloads with a different composition comprising a larger number of updates. And we also varied the event arrival rate and the time windows on queries and change requests.

We compared *CuteDB* performance against running the workload solely on the analytics database engine, which would have been the default engine in such an operation management scenario involving a mix of simple, monitoring queries and complex, analytics queries. Using a stream or a CEP engine, or the in-memory transactional database as a single competitor was not an option, as none of them could provide by itself the entire infrastructure needed in this use case (e.g., lack of support for UDFs, inferior performance for analytics, etc.).

In particular, we compared two methods for processing workloads. The first resembles common practice and is used as a baseline method. The second is *CuteDB*'s operation.

*Baseline (BSC)*. As trickle load all requests is very expensive, BSC uses bulk loading whenever is possible. Hence, inserts are bulk loaded when: (a) a query or a change request arrived, or (b) a time threshold has been met; whichever comes first. Updates and deletes are trickle loaded as they arrive.

*CuteDB Hybrid (HB)*. This is *CuteDB* employing query split, deferring operations, converting updates/deletes to inserts, etc. When no data change requests are present, HB treats event data similarly to BSC (bulk loads it to the database).

### B. Results

1) *HB vs. BSC*: We compared the two methods in terms of throughput (requests per sec), ability to run continuous queries, and response time for monitoring and analytics queries. We tested the execution of a streaming workload running for 5 minutes (after a warmup period of 2 minutes) and report the averages of 10 runs for each experiment. The workload

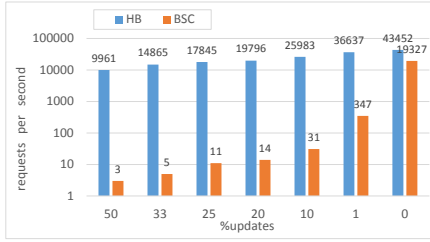
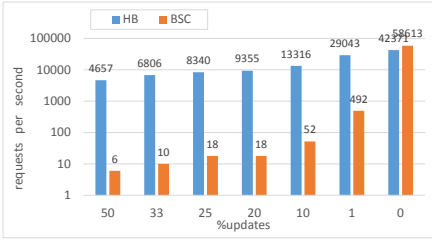


Fig. 15. %upd vs. throughput at a varying query frequency: 1qry/1s (left), 1qry/0.1s (right)

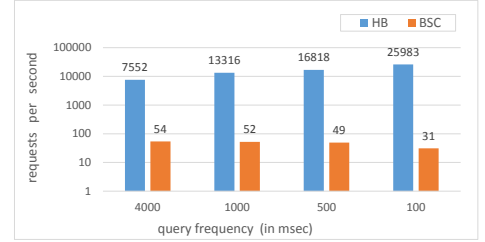


Fig. 16. Query frequency vs. throughput

consists of new and updated events that are streaming in at a fixed arrival rate, while they are monitored by 14 continuous queries looking for trends and threshold violations (these queries may be sent to either engine or to both engines). We run these queries as a series of discrete queries sent at a varying frequency. Higher frequency results in a fresher query result. At the same time, multiple instances of the 12 analytics queries run on the database over historical data (these queries run on the database and 5 of them combine results from the monitoring queries for pattern detection and prediction).

**Throughput.** We tested workloads with varying percentage of updates from 0% (only inserts), to 1%, 10%, 20%, 25%, 33%, and 50%. For *CuteDB* we used a typical configuration for the application at hand, where the buffer size was set at 2.5M rows and the time window for the continuous queries was 1 second. (But the trends observed with this configuration were similar for other settings we also tried.) At the beginning, the continuous queries were first directed to the buffer (to recent data). After 2.5M inserts, DM moved data to the database. Then, for a short period ( $\leq 1$  sec), the monitoring queries were directed to the database too. And the process goes on.

Figure 15 shows in log-scale how the throughput (requests processed per second) is affected when varying the number of updates arriving in the system. As the percentage of updates increases, HB’s throughput is up to *three orders of magnitude* higher than BSC. Interestingly, even for a very small number of updates (1%) and although, BSC gets faster, there is still a significantly large difference with HB. When only new events arrive (0%), both methods perform bulk inserts in similar way. Then, BCS is faster than HB when the DM overhead is larger than the benefit of running the queries in the buffer, instead of the database. For example, for query frequency set to 1 query per second (left plot), in the duration of the experiment (5 minutes), with the buffer size at 2.5M, and average throughput around 42K/sec, DM executed 5 times.

We increased query frequency to 1 query per 100 msec. The right plot in Figure 15 shows that HB is still superior and in fact, as query frequency increases, HB’s throughput increases too, whilst BSC’s throughput does not change much. In addition, note that for higher query frequency BSC’s throughput decreases in the presence of updates. But interestingly, BSC falls behind even when no updates are involved (0%). Then, HB is 2.26 times faster than BSC. The reason being that then BSC deals with a very large number of very short queries (similar to point or short-range queries in some cases due to the window predicates), which is not the strength of a column-store, analytics engine, but it is for the in-memory database

we use as the buffer. However, this also mean that BSC is not a good candidate to run continuous queries too.

**Continuous queries.** To study this further, we focused on a continuous query checking for pressure threshold violation, ran it as a series of discrete queries, and varied query frequency. In this experiment, we considered a workload with 10% updates, buffer size fixed at 250K rows, and query time window at 1 sec. We split the continuous query into 75 to 3000 discrete queries, by running each discrete query every 4sec, 1sec, 500msec, and 100msec. Figure 16 shows in log-scale our average findings. As before, HB’s throughput increases with query frequency, while at the same time it falls behind for BSC. Thus, continuous queries, converted to a series of very frequent discrete queries, are handled well in *CuteDB*.

**Query response time.** We also studied query response time for continuous queries. Here, we had fixed buffer size and a varying percentage of updates (10%, 25%, and 33%). We considered a single continuous query split in discrete queries and were sending a discrete query every 600 msec. We measured the delay before getting a query answer back with BSC and HB. Figure 17 shows in log-scale the average results for 10 runs. HB runs the discrete queries with no much latency: a) 6.2% avg delay for 33% UPD, b) 2.3% avg delay for 25% UPD, and c) 1.1% avg delay for 10% UPD. This is a very little overhead showing that HB may run continuous queries in near real-time. On the other hand, BSC is two orders of magnitude slower, in all cases, unable to provide fresh results.

**Analytics query latency.** Next, we investigated how loading new events or data changes affects the 14 analytics queries running in the database. We first measured execution time for each analytics query without any data ingestion, at steady state. Figure 18 shows the average difference (latency) in query execution times between the steady state and when data are streaming in as well. We tested three scenarios involving a varying number of updates (1%, 10%, 25%). BSC adds a significant latency to the analytics queries as these wait according to their query windows for the data to be ingested and also, because trickle loading of updates hurts (column-store/read-optimized) database performance. HB incurs very low latency. We present two HB variants, one with the deferring operations optimization disabled (HB-noDef) and the full-fledge HB. In HB-noDef, the large difference from BSC is attributed to the conversion of updates/deletes to inserts. The extra gain in HB is due to the deferring operations technique. This technique largely depends on the distribution of change requests in the workload and it varies with the application and the workload. Figure 18 shows a representative example of its potential.

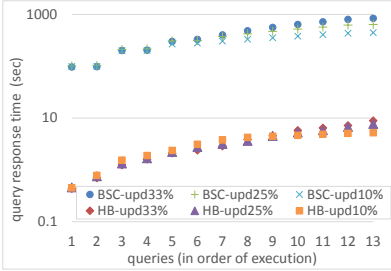


Fig. 17. Continuous query response time

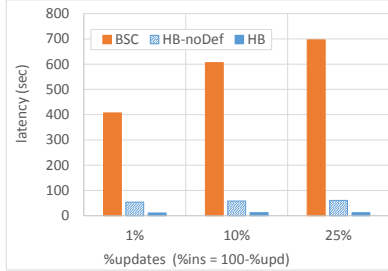


Fig. 18. Analytics query latency

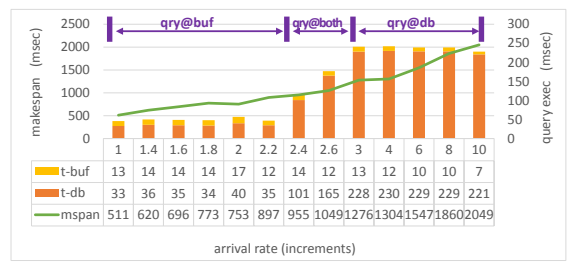


Fig. 19. Arrival rate vs. Query execution and Makespan

2) *CuteDB operation*: Next, we delve into *CuteDB* internals and investigate its behavior for various configurations.

*Arrival rate*. In this experiment, the new events and change requests arrived at various times with a varying arrival speed. The events were monitored by continuous queries with fixed windows and the buffer size remained fixed. Figure 19 shows average results for 10 runs of this experiment. The plot has two y-axes. The left one, shows makespan (i.e., total run time for workload) and corresponds to the solid, green line in the plot. The right one, shows average query runtime in msec and corresponds to the stacked-bar chart in the plot. Each bar shows query execution in the buffer (top bar fragment) and in the database (bottom bar fragment). Below the plot, we show the average data values. We started with an arrival rate at 30K requests per second. As the arrival rate increases, we observe the following: (a) more data get transferred from the buffer to the database; (b) query windows move from the recent data towards the historical data; (c) Data Move frequency increases too, and (d) thus, the makespan increases too. As the plot shows, for slower arrival rates *CuteDB* keeps up and most of the queries are executed in the buffer (the small portion shown as execution time in the database stands for a checking operation we do). As we doubled and tripled the initial arrival rate, data was moved to the database faster (the buffer sized was fixed) and thus, a larger number of queries were split between the buffer and the database. For even larger arrival rates, most of the query windows were targeting historical data.

*Data move*. DM adds an overhead to *CuteDB*'s operation. Figure 20 illustrates this with a partial snapshot of an example execution. This snapshots shows what happens at DM's operation. At steady operation, in this experiment, HB's throughput was close to 35K requests per second. When DM starts, throughput gets a sudden fall of about 25%-30%, as the query windows (and some update windows too) targeting recent data are moved toward the database. As time passes by and new requests are coming in, the windows of the operations getting processed are split to both the buffer and the database. Then, at some point, the windows target more recent data that is currently in the buffer until the next DM operation starts. This behavior of DM affects throughput negatively, but it also harms the response time of continuous queries (not shown here) as at periods, they may have to process historical data too.

We also tested various conditions for when to run DM, as described in Section IV. Figure 21 illustrates an example streaming workload with a varying arrival rate. The snapshot depicted here shows how throughput and average query re-

sponse time were affected by DM execution. The two plots show four measurements. First, each DM initiation time is illustrated on the x-axis with an orange triangle; e.g., dm(-)9 means that a DM process started at time 9. The solid light blue area shows system throughput. Its borderline shown as an orange dashed line, shows the average arrival rate per second; it starts at 45K/s, drops at 10K/s, then 45K/s, 20K/s, and 45K/s. The throughput variation is due to: (a) there is a fluctuation in the actual arrival rate (see  $c_\lambda(\lambda)$  in Section IV); and (b) as DM executes, the actual throughput drops as we saw in Figure 20. Finally, the two plots also show the average query response time for that workload as a solid purple line (it is plotted on the right y-axis in msec). Query response times vary due to: (a) queries at DM execution may split to both the buffer and the database as we show before; (b) as more data are collected in the buffer, queries targeting the buffer get slower, see for example that between times 20 and 60 in the right plot, query response times increase.

The two plots in Figure 21 illustrate two example policies for DM. The left plot shows what happens when we run DM at specific periods (here, every 10 sec). As DM executes frequently, query response time is considerably affected.

However, we can do better. As the arrival rate changes, *CuteDB* may decide dynamically when DM should run and thus, it might improve query response time at the steady state as shown in the right plot. In this case, DM follows the variation of the arrival rate and runs in a way that enables monitoring queries to run in the buffer and analytics queries in the database. When data arrives at slower rates, throughput is lower and the period between two subsequent DM executions is larger. At higher rates, throughput is higher and DM may kick in more frequently. Also, *CuteDB* analyzes the windows of the most recent queries as described in Section IV-C and lets DM move only data that are older than those needed by the most recent queries. In doing so, the most recent data needed by these queries are kept in the buffer. If a query needs older data then it is directed to the database. This operation is regulated by using the buffer size estimate of Equation 4 (see Section IV-C) in combination with the analysis of the query windows. Our experiments show that this case works fairly well when a large fraction of the queries (or large groups of consecutive queries) targets either recent data or historical data. Of course for different workloads, other policies or a different configuration might work better. We consider a more elaborate analysis of the operation of DM as a function of query windows as an interesting future work.

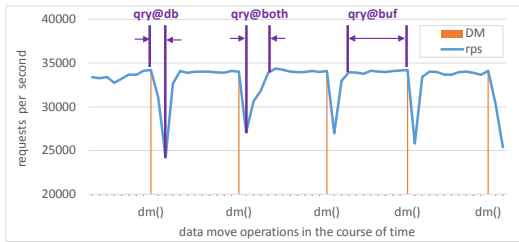


Fig. 20. DM exec frequency vs. throughput

*Discussion.* For ease of presentation, most of the experiments we show here were run for 5 minutes. We also repeated these experiments for a period of 45, 60, and 90 minutes to examine the robustness of our method. The overall trends of the longer experiments were as those described in this section.

## VI. RELATED WORK

There are not many solutions for connecting fast incoming data streams to a column-store, read-optimized database. Existing connectors lack the extra optimizations we perform in DM and the ability to query both recent and historical data at the same time. MonetDB has an extension to streams and combines streaming and persistent data [6]. It maintains an event buffer, with static size. Continuous queries are processed against the buffer and persistent tables are accessed with discrete queries. This enables most DBMS algorithms and optimizations to be reused. But an event is dropped from the buffer once it has been seen by all queries. In addition, in *CuteDB* streaming data may *modify* persistent data stores.

A previous work describes how to support continuous queries in a database, using ideas similar to incremental view maintenance [13]. The results of the queries are stored either to a queue or to a table. Users are expected to asynchronously de-queue the results from the queue or to retrieve them from the table, and also to manage the destination (table/queue). Also, the focus is on OLTP-like transactions. *CuteDB* supports analytics queries and automates the entire process (e.g., destination choice, buffer flush). Past work on evaluating queries over streaming and persistent data consider streams and relations *independently* and focus on individual, binary (or n-ary) join operations (e.g., [9], [15], [16]). Most approaches on stream query processing (e.g., aggregate computation [3], [5], [7]) do not implicate data stored in persistent data stores.

Deferred updates have been used in the context of database recovery. During a transaction, the data changes are recorded in a log and when the transaction commits, the data actually get updated. It's also being used in database replication [10] and in flash storage [1]. In a different context, deferring updates are used in the log-structured merge (LSM) trees [8]. There, index changes are cascading from a memory-based component through a disk component as in merge sort. This method was designed to reduce disk arm movements compared to a traditional access methods such as B-trees. But we are not aware of any other work using this technique to improve data change performance in column-store, analytics engines.

The problem of slow updates in read-optimized stores has

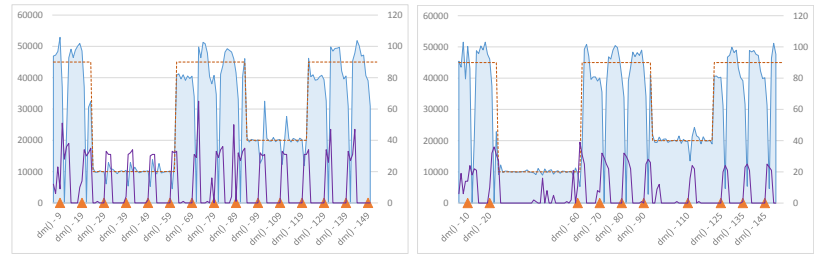


Fig. 21. DM execution periodically (left) and with the arrival rate (right)

been studied before [4]. This work proposed the Positional Delta Tree (PDT) for maintaining positional updates and showed results based on the TPC-H benchmark. *CuteDB* is designed to deal with high rates of incoming streaming data and works on top of the query engines. Working with the internals of an analytics query engine seems to be a promising area for future work.

## VII. CONCLUSIONS

Real-time operations management comes with three challenges, it needs to (a) provide continuous monitoring of a high-speed event stream for patterns of interest; (b) support complex, analytics queries for problem diagnosis and prediction; and (c) perform data change requests efficiently, since the monitored environment undergoes frequent change.

Our hybrid architecture resolves these challenges by employing an in-memory transactional database for fast processing and an analytics engine for complex, analytics queries. Our experiments show that *CuteDB* is a promising architecture and system. Future work includes studying more elaborate scenarios for engine synchronization based on the query workload.

## REFERENCES

- [1] B. K. Debnath, M. F. Mokbel, D. J. Lilja, and D. H. C. Du. Deferred updates for flash-based storage. In *IEEE MSST*, pages 1–6, 2010.
- [2] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.
- [3] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. *SIGMOD Rec.*, 30(2):13–24, 2001.
- [4] S. Héman et al. Positional update handling in column stores. In *SIGMOD*, pages 543–554, 2010.
- [5] J. Li et al. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.
- [6] E. Liarou, S. Idreos, S. Manegold, and M. Kersten. Enhanced stream processing in a dbms kernel. In *EDBT*, pages 501–512, 2013.
- [7] J. Mondal and A. Deshpande. EAGr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. *CoRR abs/1404.6570* 2014.
- [8] P. E. O’Neil, E. Cheng, D. Gawlick, and E. J. O’Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.
- [9] N. Polyzotis et al. Meshing streaming updates with persistent data in an active data warehouse. *IEEE TKDE*, 20(7):976–991, 2008.
- [10] R. Schmidt and F. Pedone. A formal analysis of the deferred update technique. In *DISC*, pages 499–500, 2007.
- [11] A. Simitsis, C. Gupta, K. Wilkinson, and U. Dayal. Optimizing flows for real time operations management. In *SSDBM*, pages 607–612, 2012.
- [12] A. Simitsis et al. *CuteDB: A Hybrid Engine that Enables Analytics for Real-time Operations Management*. Technical report, HP Labs, 2015.
- [13] S. Subramanian et al. Continuous Queries in Oracle. In *VLDB*, pages 1173–1184, 2007.
- [14] N. Tatbul et al. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
- [15] W. H. Tok and S. Bressan. Efficient and adaptive processing of multiple continuous queries. In *EDBT*, pages 215–232, 2002.
- [16] K. Towne et al. Window query processing for joining data streams with relations. In *CASCON*, pages 188–202, 2007.