IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 15, 2020, accepted May 30, 2020, date of publication June 3, 2020, date of current version June 16, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2999572

*Streaming*Cube: Seamless Integration of Stream Processing and OLAP Analysis

SALMAN AHMED SHAIKH[1] AND HIROYUKI KITAGAWA[2], (Member, IEEE)
[1]Artificial Intelligence Research Center (AIRC), National Institute of Advanced Industrial Science and Technology (AIST), Tokyo 135-0064, Japan
[2]Center for Computational Sciences, University of Tsukuba, Tsukuba 305-0006, Japan

Corresponding author: Salman Ahmed Shaikh (shaikh.salman@aist.go.jp)

This work was supported in part by JSPS KAKENHI under Grant JP20K19806 and Grant JP19H04114, and in part by the Project Commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

ABSTRACT Many organizations require real-time analysis of their business data streams to make instantaneous decisions. Data streams are often multidimensional and at the lowest level of abstraction, yet analysts are interested in their multilevel interactive analyses across several dimensions. Online analytical processing (OLAP) is a technique whose utility has been proven for such an analysis. Traditionally, OLAP over streams has been achieved by coupling a stream processing engine (SPE) with an OLAP engine. However, for many organizations, this is not an effective solution as it results in lower performance, resource wastage, and increased complexity and maintenance costs. Hence, we present *Streaming*Cube, which is a unified framework for stream processing and OLAP analysis. The idea of a unified framework is supported by the observation that the incremental computation in stream processing and the maintenance of the materialized view of OLAP have many similarities. To seamlessly integrate the SPE and OLAP, as well as to maintain the OLAP lattice vertices incrementally, a *cubify* operator is introduced. *Streaming*Cube supports two types of queries, namely, continuous queries (CQs) and OLAP queries. To demonstrate the effectiveness of the proposed framework, a detailed experimental evaluation is presented herein.

INDEX TERMS Continuous queries, cubify operator, incremental computing, materialized views, OLAP analysis, stream processing, unified framework.

I. INTRODUCTION

In the contemporary global economy, organizations cannot survive without analyzing their business data, and different analytic tools and techniques are being used. Online analytical processing (OLAP) is one technique used to perform the multidimensional analysis of data, although it was initially utilized for static data. With the recent increase in the amount of stream data, the demand for their analysis and processing is on the rise. Many modern applications are a mixture of streaming and analytical workloads. For instance, 1) Multinational wholesale and retail chains (e.g., Walmart, Costco, etc.) are interested in analyzing their sales stream across several dimensions (e.g., product, store, customer, etc.) in real time to make timely decisions; 2) Telecommunication companies receive the call records and data usage information of their subscribers in the form of data streams and are interested in analyzing bandwidth usage trends across area, day, and time dimensions to improve service provision for their customers. In these examples, several multidimensional stream data are at the lowest level of abstraction; however, data analysts are interested in analyzing data at different levels of abstraction and across different sets of dimensions. OLAP is a state-of-the-art technique for analyzing static data, and it has also been recently employed for data streams.

Although independent solutions for processing data streams exist [1]–[9], as well as their OLAP analysis (drill-down, roll-up, slice, and dice) [10], [11], few solutions are viable for both applications. Traditionally, OLAP analysis over data streams is achieved by coupling a stream processing engine (SPE) with an OLAP engine. For instance, [12] coupled μCosminexus SPE [13] with a locally developed OLAP engine to process and perform OLAP analysis over data streams. Similarly, SnappyData [14] integrated Apache Spark [15] as a computational engine and Apache GemFire as an in-memory transactional store to support transaction processing and approximate analytical queries on high speed data streams. However, coupling of multiple systems results

The associate editor coordinating the review of this manuscript and approving it for publication was Wen Chen.

in lower performance and resource waste. For instance, the data needed to be transferred from one system to another, which may also require transformation, causing performance degradation as well as the need for the data to be replicated for each coupled system, resulting in wastage of storage resources. Furthermore, coupling increases complexity and maintenance cost of the overall system. Thus, this work proposes an integrated framework, *Streaming*Cube, to enable stream processing and OLAP analysis on a single platform with the help of a novel *cubify* operator. The idea of a unified framework is supported by the observation that the incremental computation in stream processing and OLAP's materialized view/vertex (hereafter, view and vertex are used interchangeably) maintenance have many similarities. *Streaming*Cube is built on an incremental data processing framework and supports all the essential stream processing operators. To support OLAP operations, a *cubify* operator is introduced, which incrementally maintains materialized vertices within a dedicated cubify-window (c-window). An end user needs to specify which vertices to materialize with the help of a configuration file. *Streaming*Cube supports two classes of queries over data streams: 1) Continuous queries (CQs) and 2) OLAP queries. CQs in our framework do not only maintain OLAP materialized views but also generate continuous aggregates corresponding to OLAP lattice vertices.

Use case scenario (Retail Chain OLAP Analysis) – A retail chain collects sales data of its stores at the granularity of the individual product, store location, supplier, and promotion (under which the product is sold) dimensions. The data arrive continuously as an infinite data stream. The end user is interested in executing a continuous query on the stream and directing the output to some application program. Furthermore, (s)he is interested in its real-time interactive analysis across different dimensions. Traditionally, this is achieved by coupling an SPE with an OLAP engine. Since the business streams of retail chains and similar organizations are not so velocious (Walmart, one of the largest retail chains, receives 17,000 online orders per minute, which is approximately 283 orders per second [16]), they do not need complex distributed solutions capable of handling millions of tuples per second using dedicated SPEs and OLAP engines. Although such solutions can handle huge data traffic, their deployment and maintenance costs are too high and are useless to organizations with low to medium data processing and analytical requirements. Hence, for such organizations, a unified framework is more desirable.

This work is an extended version of our *Streaming*Cube demo paper [17]. In contrast to the demo paper, this paper presents the technical details of *Streaming*Cube and its data structures. Furthermore, an extensive experimental study is presented to evaluate the proposed framework. The main contributions of this work are summarized below as follows:

- An integrated framework *Streaming*Cube, which supports the following:
  1) All essential stream processing operators, including windowing, select, join, etc., wherein the operators' synopses are maintained incrementally.
  2) Interactive OLAP analysis with the help of the introduced *cubify* operator.
- A prototype system corresponding to the proposed unified framework.
- A detailed experimental evaluation to demonstrate the effectiveness of the proposed framework and the *cubify* operator.

The rest of the paper is organized as follows: Related works are discussed in Section II, while some essential concepts are presented in Section III. In Section IV, the *Streaming*Cube data processing framework is presented. Details of the proposed *cubify* operator are given in Section V, while in Section VI, the query specification of the proposed framework is presented. Incremental processing to maintain synopses and materialized views in the proposed framework is presented in Section VII. The *Streaming*Cube architecture is discussed in Section VIII. The detailed experimental evaluation is presented in Section IX, while Section X concludes this work and discusses some future directions.

II. RELATED WORK

Since the focus of this work is multifold, i.e., stream processing, OLAP analysis, and the maintenance of materialized views, we divide this section into two.

A. STREAM OLAP

Stream processing has recently gained significant attention given the exponential growth in data stream sources. Several SPEs have been proposed and developed by academic researchers as well as industries whose primary objective is to process high velocity data streams effectively. Apache Storm [1], Apache Flink [2], STREAM [3], Aurora [4], Borealis [5], Apache Samza [6], MillWheel [7], S4 [18], etc., are some well-known and commonly used SPEs, most of which employ incremental computation for processing data streams. However, these systems do not support OLAP analysis over data streams, which is an essential requirement of many organizations nowadays.

However, OLAP over static relational data and the maintenance of materialized views have been intensively studied by database researchers for decades. [19] investigated the issue of nodes materialization when it was expensive to materialize all the OLAP lattice nodes. They presented a greedy algorithm that determined a good set of nodes to be materialized. Aouche *et al.* in [20] proposed to couple materialized views and index selection to consider view-index interaction and achieve efficient storage space sharing in OLAP. [21] proposed a systematic study of the OLAP view and index-selection problem. C. Joslyn *et al.* [22] made used of statistical techniques for effective view maintenance. However, the focus of these works was OLAP analysis over static data rather than data streams. Hence, they are unsuitable for data streams.

One of the earliest studies on stream OLAP was conducted by J. Han *et al.* [10]. They proposed an architecture to facilitate OLAP for streams. To reduce the query response time and the storage cost, their system keeps distant data and very new data at coarser and finer granularities, respectively, and precomputes some OLAP queries at coarser, intermediate, and finer granularity levels. E. Lo *et al.* in [23] presented an OLAP solution for sequential data, i.e., for data streams that exhibit logical ordering among their data items. However, their solution is a good fit for sequence data rather than general data streams. A framework to support the OLAP-style interactive exploration of Twitter data by the hierarchical spatio-temporal hashtag clustering of Twitter streams was proposed by W. Feng *et al.* [24]. However, their solution is extremely specific and targets clusters of textual data (hashtags) based on the time and locations of tweets, and thus, cannot be used for general OLAP analysis over data streams. These frameworks are beneficial in the maintenance of materialized views over data streams; however, they all require a dedicated SPE to process data streams, i.e., they do not support general stream processing capability as our unified framework does. Similarly, high performance analytical systems, such as MemSQL [25] and Apache Cassandra [26] both require an external stream engine for OLAP analysis over data streams.

In addition to the above-mentioned solutions, which may be used for stream OLAP by either integrating them with an SPE or an OLAP engine, there exists a few solutions that couples multiple engines to achieve stream OLAP over evolving data streams. For instance, [12] coupled an SPE with an OLAP engine to perform OLAP analysis over data streams. Similarly, SnappyData [14] integrated Apache Spark [15] as a computational engine and Apache GemFire as an in-memory transactional store to support transaction processing and approximate analytical queries on high-speed data streams. However, it does not intrinsically support OLAP operations, for instance, drill down, roll up, etc., and therefore, does not maintain materialized views at multiple granularity levels as ours does herein. Besides, these hybrid solutions require deployment, integration, and maintenance of multiple independent engines. Although these hybrid/distributed solutions are useful to organizations with heavy data loads, our proposed solution is more desirable for organizations with low to medium workloads.

Besides the above-stated works, there exists a few solutions that deal with the integration of SPE and OLAP engines. For instance, M. Sadoghi *et al.* in [27] presented a lineage-based data store that combined real-time transactional and analytical processing within an engine with the help of lineage-based storage. However, their focus entailed the improvement of general analytics through the use of lineage-based storage rather than core OLAP, i.e., their work did not discuss the materialized view maintenance for OLAP queries as we do herein. Interactive analysis over data streams can also be achieved by utilizing Python or Scala over Spark shell [28]. However, such approaches do not support incremental view maintenance for interactive OLAP analysis. In contrast to the aforementioned works, an integrated framework for stream processing and OLAP analysis for organizations with low to medium data processing and analytical workloads is presented herein.

B. MATERIALIZED VIEW MAINTENANCE

Materialized view maintenance, which has been studied by many researchers, is a mature database area. Several works have discussed techniques for efficient materialized view maintenance for databases with high update rates and data streams. For instance, [29] presented a view maintenance approach *viewlet transforms* for databases that change at exceptionally high rates. *Viewlet transforms* materializes a query and a set of its higher-order deltas (views), which support each other's incremental maintenance, thereby resulting in reduced overall view maintenance costs by trading space. Phantoms are intermediate materialized queries (views) to accelerate user queries. Zhang *et al.* [30] proposed the use of phantoms to reduce the overall cost (query answering and data transfer costs) within the extremely limited memory of a network interface card. M. Nikolic *et al.* [31] presented an incremental view maintenance technique called *domain extraction*, which enabled an incremental maintenance of views based on complex Structured Query Languages (SQL) queries with nested aggregates for batch updates. They studied the trade-offs between single-tuple and batched incremental processing and identified the cases where batching could improve the performance of the incremental view maintenance. G. Luo *et al.* in [32] proposed a content-based method for detecting irrelevant updates to the base relations of a materialized view. Their proposed approach can reduce the view maintenance processing cost at the expense of memory space. Fegaras [33] presented an incremental stream processing approach of nested-relational queries capable of converting the latter to the former programs automatically. Authors in [34] presented the View Delta Function (ViewDF), a framework for the incremental maintenance of materialized views over streaming data. The main component of the proposed framework is the ViewDF, which declaratively specifies how to update a materialized view when a new batch of data arrives. Although the works of [33] and [34] help incremental view maintenance in streaming workloads, their solutions do not support general stream processing as ours does.

Data warehouses use materialized views intensively to answer user queries. DataDepot [35], Everest [36], Moirae [37], latte [38], and Truviso [39] are some of the earliest stream warehouse systems. DataDepot [35] is a streaming warehouse designed to automate the ingestion of streaming data from a wide variety of sources and to maintain materialized views over these sources. Furthermore, the authors in [35] discussed the issue of an out-of-order streaming tuple. They extended their work in [40] and presented a mechanism for managing and exploiting the multilevel consistency of materialized views in a stream warehouse. The consistency problem in their work entailed the synchronization delays caused by the late and out-of-order arrival of stream

tuples (temporal consistency) rather than transactional consistency. Their proposed consistency model focuses on query and update consistencies. Everest [36] is a relational data warehouse engine on commodity hardware, which utilizes a column storage system and supports distributed query processing for effective analytical query processing. In contrast to DataDepot and Everest, which maintain streaming and transactional warehouses in underlying relational databases, respectively, for efficient analytical query response, our proposed framework supports OLAP analysis in addition to traditional stream processing by maintaining materialized views on the primary memory.

Moirae [37] integrates continuous monitoring with the querying of a historical log to enable history-enabled monitoring. The Moirae architecture is based on hierarchical log partitioning and query execution, where the recent past is stored at a higher cost but can be queried faster than older data. Latte [38] employs the Niagara internet query system [41] to answer analytical queries that combine live data streams with data archives. Truviso [39] proposes a query-processing approach that runs SQL continuously and incrementally over relational tables, streams, and combinations of the two before the data are stored in the database.

In contrast to the above-mentioned works on the effective maintenance of materialized views, an integrated framework for stream processing and incremental materialized view maintenance is presented herein.

III. PRELIMINARIES

A continuous query in *Streaming*Cube follows the Continuous Query Language (CQL) specification [42]. In this section, we briefly present the CQL model and its incremental computation, which is essential to understand the proposed framework. For details on CQL, please refer to [42].

A. CQL MODEL

The CQL model is based on streams and relations. Each stream and relation has a fixed schema consisting of a set of named attributes. Let Γ be a discrete, ordered time domain, then a time instant is any value from Γ. A *stream* S is an unbounded multiset of tuples $[\langle e\rangle, t]$, where e is a tuple belonging to the schema of S and $t \in \Gamma$ is the timestamp assigned to e by the system on its arrival, while a *relation* R is a time-varying (updatable) mapping from Γ to a finite but unbounded multiset of tuples belonging to the schema of R. A relation R at a time instant $t \in \Gamma$ is denoted by $R(t)$ and is called an *instantaneous relation*. CQL semantics is based on three classes of operators over streams and relations. 1) The *relation-to-relation* operator takes one or more relations as the input and produces a relation as the output. 2) The *stream-to-relation* operator takes a stream as the input and produces a relation as the output. 3) The *relation-to-stream* operator takes a relation as the input and produces a stream as the output.

CQL is defined by instantiating the operators of the three classes. For the *relation-to-relation* operators, CQL uses existing SQL constructs. All the *stream-to-relation* operators in the CQL, which are specified using the window specification language derived from SQL-99, are based on sliding window over stream. A window at any point in time holds a historical snapshot of a finite portion of the stream. Herein, two classes of sliding window operators are used, i.e., tuple-based and time-based. 1) The *Tuple-based window* operator on a stream S is specified using an integer n. At any time t, it returns a relation R of n most recent tuples from stream S. 2) The *Time-based window* takes a parameter τ and at any time t returns a relation R containing tuples with timestamps between $t - \tau$ and t from a stream S. The CQL has three *relation-to-stream* operators, which are also adopted in our framework: insert (i)-stream, delete (d)-stream, and relation (r)-stream. At a time t, the i-stream applied to a relation R results in a stream tuple $[\langle e\rangle, t]$ whenever tuple e is in $R(t) - R(t-1)$. The d-stream returns a stream tuple $[\langle e\rangle, t]$ from R whenever tuple e is in $R(t-1) - R(t)$, while the r-stream applied to R, results in a stream tuple $[\langle e\rangle, t]$ whenever tuple e is in $R(t)$. A sample CQL query is shown in Query 1, which performs binary join between two streams S1 and S2.

```
Select S1.A, S2.B
From S1[Rows n], S2[Range τ]
Where S1.C = S2.C
```

Query 1. A Simple CQL query.

B. CQL IMPLEMENTATION AND INCREMENTAL COMPUTATION

STREAM [3] is a well-known implementation of the CQL model. In STREAM, once a CQL query is registered, it is compiled into a query plan and is executed continuously. Query plans are composed of operators, queues, and synopses. *Operators* perform the actual processing on data streams. The data arrive at an operator as a sequence of elements, and each operator reads from one or more input queues, processes the input, and writes any output to the output queue. A *Queue*, which buffers the elements as they move between operators, connects its input operator to its output operator. *Synopses* store the intermediate state needed by continuous query plans. A synopsis is owned by a single operator O, and the state contained by the synopsis is needed to evaluate O in future. In STREAM SPEs, query operator synopses are maintained incrementally. To achieve incremental computing, STREAM uses $+/-$ flags embedded in elements. When a stream tuple $\langle e\rangle$ enters the STREAM SPE at timestamp t, it is assigned a "+" lag by its respective window operator. This tuple-timestamp-flag triple $[\langle e\rangle, t, +]$, also called element, is sent to the query downstream operators. The query operators use this element to update their synopses. Similarly, on the expiration of an element due to window size, a corresponding "−" flagged element, $[\langle e\rangle, t, -]$, is sent to the query downstream operators. On receiving "−" flagged elements, the query operators update their synopses by removing the corresponding "+" elements.

More precisely, inside STREAM, streams and relations are represented uniformly as sequences of elements. A stream is represented as a sequence of timestamped-element ''insertions'' (a timestamped-element with a ''+'' flag). A relation is also represented as a sequence of timestamped-elements, except that a relation has both ''insertion'' and ''deletion'' (a timestamped-element with a ''−'' flag) elements to capture the changing state of the relation.

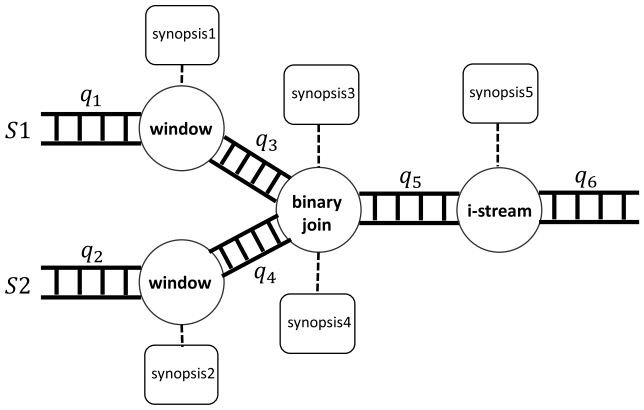


FIGURE 1. Query 1 query plan.

A query plan for Query 1 is shown in Fig. 1. The query plan consists of four operators: two instances of window operators, a binary join operator, and an i-stream operator. Note that the projection is performed as part of the binary join, so no separate projection operator is needed. Queues q_1 and q_2 hold the elements of the input streams $S1$ and $S2$, respectively. Queues q_3 and q_4 hold elements corresponding to the relations $S1$[Rows n] and $S2$[Range τ Seconds], respectively. Queue $q5$ holds output elements from the binary join operator, while queue q_6 holds stream elements as generated by the i-stream operator. The query plan has five synopses, *synopsis1*∼ *synopsis5*. Each window operator has a synopsis to maintain its current state. The binary join operator has two synopses to materialize its relational inputs, whereas the i-stream operator has a synopsis to generate the output stream.

IV. OLAP AND MATERIALIZED VIEWS MAINTENANCE IN *STREAMING*CUBE

OLAP is a technique for interactive analysis of multidimensional data. It makes use of a partially-normalized star schema (Appendix B) for efficient analysis. The data in a star schema are represented as a data cube consisting of several dimension tables and a fact table. Dimension tables contain descriptive attributes, while fact tables contain business facts, also called measures, and foreign keys referring to the primary keys in the dimension tables. Some dimension attributes are hierarchically connected, and they comprise a cube lattice where each lattice vertex corresponds to different combinations of attributes at different hierarchy levels and an edge between two vertices represents a subsumption relation between them. Hence, vertices in a lattice are combinations of dimension attributes and represent OLAP queries. In practice, only a few lattice vertices are materialized, while queries related to non-materialized vertices are computed from the materialized vertices on an ad-hoc basis. Typical OLAP operations, including roll-up, drill-down, slice, and dice, are implemented using one of the aggregation operations, i.e., sum, min, max, etc., and are defined as follows:

- **Roll-up:** This is also known as aggregation, and it can be performed by reducing the dimensions or moving up in the dimension hierarchy.
- **Drill-down:** This is the reverse operation of roll-up. It is performed by either moving down in the dimension hierarchy or by introducing a new dimension.
- **Slice:** This operation selects one particular dimension from a given cube and provides a new sub-cube.
- **Dice:** This operation creates a sub-cube by performing a selection on two or more dimensions.

OLAP over data streams allows fact tables to be streams. This is different from static data as the stream data arrive continuously and at high velocities, thereby requiring the materialized vertices to be updated continuously. Hence, OLAP materialized vertices are usually maintained on a primary memory and are always updated incrementally. *Streaming*Cube proposes a novel operator *cubify* to generate an OLAP lattice structure and maintain its materialized views incrementally. Herein, it is assumed that only the fact table is the data stream, while the dimension tables are static.

A database materialized view is a derived relation defined on top of one or more finite-base relations. Since data streams are infinite, the maintenance of materialized views over them requires a mechanism to bound data stream tuples. Conventional SPEs utilize windows to bound stream tuples for blocking operators. A window is used to execute queries only over the recent contents of data streams [43]. One could use windows for bounding the contents of the stream-based materialized views. In this case, the duration of OLAP queries is limited by the window size, which usually contains very recent data only. To support OLAP analysis over larger time durations, this work presents a cubify-window (c-window), which is a dedicated window for the lattice-materialized vertices and is maintained by the *cubify* operator.

Definition 1 (Cubify-window (c-window)): The c-window is a time-based window with size Δ, whereof the materialized vertices are maintained. This corresponds to the finest lattice vertex and is maintained at the lowest time granularity level, which enables the *cubify* operator to answer the OLAP queries at any time granularity.

In *Streaming*Cube, unlike the ordinary window operator, the c-window size is supplied as a parameter in a configuration file (Appendix A). The size of a c-window is usually kept larger than the ordinary window to enable analysis over historical data as well. However, the user is free to choose any size for it.

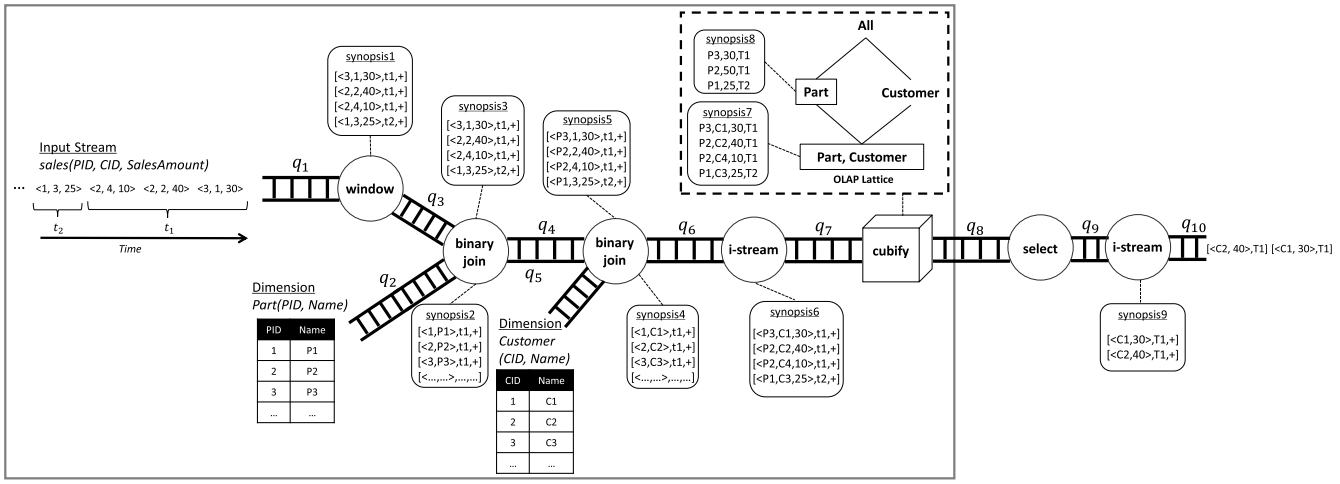


FIGURE 2. Query plan tree of a Query with two dimension tables and a *cubify* operator.

V. THE CUBIFY OPERATOR

This section presents the *cubify* operator, which is the core of *Streaming*Cube. In the following, we present its logical and physical representations, along with an example.

A. LOGICAL REPRESENTATION

Cubify is a *stream-to-relation* operator. It takes a stream S as the input, updates one or more OLAP materialized vertices $v_1, \ldots, v_n$ ($1 \leq n \leq l$, where l is the number of OLAP lattice vertices), and produces zero or more relations $R_0, \ldots, R_m$ ($0 \leq m \leq l$) as the output. The granularities and schemas of R_i can be different from S and depend on the lattice vertices selected for output generation. At any time instant t, v_i contains aggregated tuples within $[t - \Delta, t]$.

On its initial execution, the *cubify* operator materializes n ($1 \leq n \leq l$) lattice vertices, where each materialized vertex is associated with a synopsis. The schema of each materialized vertex v_i depends on the grouping of dimensional attributes and the time granularity (δ_i), where $\psi \leq \delta_i \leq \Delta$. ψ is the time granularity wherein the *cubify* operator receives stream tuples. Please note that δ_i depends on v_i and δ_0 denotes the lowest time granularity whereof the finest lattice vertex is maintained. The user must specify the corresponding lattice vertices to get relational outputs from the *cubify* operator. The output can be processed by the query downstream operators or sent to the user via the i-stream, r-stream, or d-stream operator. It is not necessary for a lattice vertex to be materialized to generate output. A relational output corresponding to a non-materialized vertex is generated using the nearest materialized vertex on an ad-hoc basis.

Definition 2 (Nearest Vertex of v_q ($NV(v_q)$)): Let V_n be the set of materialized vertices, which can respond to the non-materialized vertex query v_q (i.e., contain all the attributes of v_q). Assuming that $|v_i|$ denotes the number of rows in vertex v_i, then the nearest vertex of v_q is a $v_i \in V_n$, such that $|v_i|$ is minimum among all the vertices of V_n.

In addition to continuous output generation, the *cubify* operator can answer ad-hoc OLAP queries corresponding to any lattice vertex. If the queried vertex (v_q) is not materialized, the *cubify* operator utilizes $NV(v_q)$ to answer it. Note that the *cubify* operator always materializes the finest lattice vertex to answer all possible OLAP queries. Algorithm 1 presents how a v_q can be answered from a materialized or non-materialized vertex. The *cubify* operator treats time as part of all the lattice vertices as it is assigned to all incoming stream tuples by the system. Hence, we can always aggregate a vertex with respect to time, i.e., s, min, h, etc.

Algorithm 1 OLAP Queries Computation

Input: Queried vertex v_q, Time granularity τ
Output: v_τ: Queried vertex aggregated w.r.t. τ

```
V: Set of OLAP lattice' vertices
V_m: Set of materialized vertices
V_n: V − V_m
if v_q ∈ V_m then
    v_τ ← v_q aggregated w.r.t. τ
else
    Find the nearest vertex v' ∈ V_n which can answer v_q
    v'' ← v' aggregated w.r.t. queried vertex dimensions
    v_τ ← v'' aggregated w.r.t. τ
end if
return v_τ
```

B. PHYSICAL REPRESENTATION

Since *cubify* is a *stream-to-relation* operator, it can either directly accept stream inputs or follow any *relation-to-stream* CQL operator in a CQL query plan, as shown in Fig. 2. Queries with the *cubify* operator require a configuration file specifying the following: 1) Dimension attributes along with their hierarchies to create lattice structure, 2) Lattice vertices to materialize along with their time granularities, δ_i, 3) Lowest time granularity, δ_0, wherein the finest lattice vertex is

maintained, 4) Size of the c-window, Δ, and 5) *cubify* output vertices ($R_0, \ldots, R_m$ ($0 \leq m \leq l$), where l is the number of OLAP lattice vertices). Appendix A shows a sample configuration file.

The *cubify* operator, like other CQL operators, reads from an input queue, processes the input by updating its associated synopses (materialized vertices), and writes any output to the output queue. In contrast to other CQL operators, it maintains a variable number of synopses depending on the number of materialized lattice vertices. Although there exists an array of research on how to select the best set of vertices to materialize for static OLAP [20], [44]–[46], only a few exist for stream OLAP [10], [12], [34]. While some are interested in minimizing the OLAP-querying cost, others focus on materializing the maximum number of vertices within a given memory. In [12], we presented an optimization algorithm to select the best set of vertices to materialize, which can minimize the OLAP-querying cost within a given memory. *Streaming*Cube supports the optimization algorithm proposed in [12]. In addition, the user can explicitly specify which vertices to materialize through the configuration file (Appendix A).

1) DATA STRUCTURE

The *cubify* operator's materialized vertices are based on the c-window, which corresponds to the finest lattice vertex of size Δ and slide step δ_0, as shown in Fig. 3. Other materialized vertices are maintained at coarser granularity levels. Each materialized vertex maintains a dedicated relational synopsis, m-synopsis. In contrast to the ordinary synopsis, the elements in the m-synopsis do not contain +/− flag; hence, we call them rows. A row contains dimensional keys, one or more facts, and a timestamp corresponding to its schema. In contrast to the ordinary synopses, the rows in the m-synopsis may be aggregated.

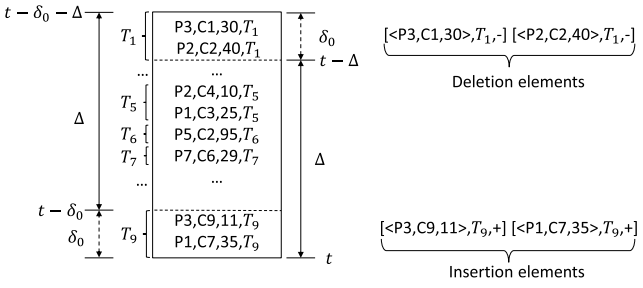


FIGURE 3. The cubify-window (c-window) synopsis.

The m-synopsis is implemented as a doubly linked list in which the rows are ordered by timestamp. The rows in the m-synopsis are partitioned into timestamps $T_1, T_2, \ldots, T_m$ and are maintained at a δ_i time granularity level. Hence, the stream elements, which arrive at the *cubify* operator during the j^{th} timestamp are assigned to the T_j timestamp partition, as shown in Fig. 3. At any time instant, the m-synopsis consists of m timestamps. A timestamp partition is called *active* if stream elements can still be assigned to it. For instance, let δ_i = min, then a timestamp partition $T_j \in v_i$ remains active and can accept stream elements for a span of 60 s. After that partition, T_{j+1} becomes active. This work assumes that the *cubify* operator receives elements in a timestamp order. Hence, each arriving element either updates an existing row or is appended as a new row in the m-synopsis active partition. Namely, if the dimensional keys of the incoming element ε matches the keys of an existing row ρ in the v_i active timestamp partition T_a, ρ is updated with ε. Otherwise, ε is added as a new row in v_i with T_a. Hence, only the rows in the active partition are checked for updates rather than the entire m-synopsis.

Example 1: Assuming that a *cubify* operator receives stream elements at a ψ time granularity with timestamps $t_1, t_2, \ldots$. The finest m-synopsis maintains rows at the δ_0 time granularity with timestamps $T_1, T_2, \ldots$. Let δ_0 = min, ψ = s, and the active partition be T_9, as shown in Fig. 3. On receiving the stream elements consisting of two dimensional attributes, a fact value, a timestamp, and a flag, [<P3, C9, 21>, t_{951}, +] and [<P8, C9, 18>, t_{952}, +], the *cubify* operator updates the m-synopsis rows in the active timestamp partition T_9. Assuming *sum* as an aggregation operation of the m-synopsis, after the update, the active partition will contain the rows: [<P3, C9, 32>, T_9], [<P1, C7, 35>, T_9], and [<P8, C9, 18>, T_9]. Here, the fact value "32" in the row [<P3, C9, 32>, T_9] is the sum of the fact values "11" and "21" of the existing row [<P3, C9, 11>, T_9] and the newly arrived element [<P3, C9, 21>, t_{951}, +], respectively.

The materialized vertices maintained by the *cubify* operator are based on the c-window, which is checked for expired rows after each δ_0 time unit. After the detection of expired rows, i.e., the rows with timestamps older than $t - \Delta$, they are deleted from the finest lattice vertex v_0. In addition, the expired rows need to be deleted from other materialized vertices. To achieve this in an incremental fashion, the expired rows from v_0 are communicated to all the materialized vertices v_i as deletion "−" elements. On receiving the "−" elements, the materialized vertices update the oldest timestamp partitions in their synopses. Similarly, on the reception of a new stream element by the *cubify* operator, the active partition of v_0 is updated and a corresponding insertion "+" element is communicated to all the materialized vertices to update their synopses. To communicate the effect of these insertions and deletions to the downstream operators, "+/−" elements corresponding to $R_0, \ldots, R_m$ ($0 \leq m \leq l$) are generated and sent downstream as the c-window moves. For instance, consider the movement of the c-window from [$t - \delta_0 - \Delta$, $t - \delta_0$] to [$t - \Delta$, t] in Fig. 3. As the c-window moves, "+" elements ([<P3, C9, 11>, T_9, +] and [<P1, C7, 35>, T_9, +]) are generated and sent to the downstream operators corresponding to the rows in timestamp partition T_9. Similarly, "−" elements ([<P3, C1, 30>, T_1, -] and [<P2, C2, 40>, T_1, -]) corresponding to the elements in the timestamp partition T_1 are sent to the downstream operators.

Note that the effect of the *cubify* operator cannot be achieved by a continuous query with multiple aggregation

operators because in addition to aggregation, the *cubify* operator supports the following: 1) OLAP queries, 2) generation of aggregates related to non-materialized vertices, and 3) the use of a dedicated window (c-window) for analysis.

2) COMPUTATIONAL COMPLEXITY

Consider a materialized vertex v_i with a time granularity δ_i. Let n_i and m_i denote the size and the total number of timestamps maintained in v_i synopsis, respectively. An insertion operation in the m-synopsis checks only the rows in the active partition; hence, the worst and average cases of insertion complexities are $O(n_i/m_i)$ and $O(n_i/2m_i)$, respectively. Similarly, a deletion operation in the m-synopsis checks only the rows in its oldest timestamp partition; thus, the worst and average cases deletion complexities are $O(n_i/m_i)$ and $O(n_i/2m_i)$, respectively. The worst case and average case time complexities of insertion for the finest vertex v_0 with time granularity δ_0 are same as those of v_i. However, the deletion complexity of v_0 is $O(1)$ because when the c-window slides, all the rows in the oldest partition of the v_0 synopsis are expired and deleted. To support the efficient insertion and deletion at the top and bottom of the m-synopses, respectively, doubly linked lists are used. Querying cost differs for the materialized and non-materialized vertices. The complexity of querying a materialized vertex, i.e., retrieving n_i rows from the m-synopsis of v_i, is $O(n_i)$. Querying a non-materialized vertex involves a small additional hashing cost to aggregate the retrieved rows, keeping the querying complexity at $O(n_i)$.

VI. QUERY SPECIFICATION

*Streaming*Cube supports two types of queries: 1) CQs and 2) OLAP queries. CQs can be further subdivided into two types i.e., 1.1) Ordinary CQs, and 1.2) Cubify CQs. *Ordinary CQs* are supported by most SPEs. A user can register *Ordinary CQs* to perform typical stream processing operations, including select, filter, aggregate, join, group-by, etc. *Cubify CQs* are introduced herein to support OLAP operations over data streams. *Cubify CQs* utilize the *cubify* operator in addition to regular CQL operators. Since the *cubify* is a *stream-to-relation* operator, within the *Cubify CQ*, it can appear at the beginning of the query (accepting stream input directly) or can follow any *relation-to-stream* CQL operator.

A sample *Cubify CQ* written in Jaql [47] is shown in Query 2. Since *Streaming*Cube supports JavaScript Object Notation (JSON) data [48], Jaql is an evident query format for it. Notably, *Streaming*Cube can process any JSON streams internally. However, in this paper, we focus on the relational streams represented in the JSON format for simplicity; however, its discussion is outside the scope of this work. The query reads a stream source *LINEORDER* and four dimension tables *CUSTOMER, PART, SUPPLIER*, and *STORE*, corresponding to TPC-H schema (Appendix B), and performs join among them. Finally, the query makes use of an i-stream operator followed by a *cubify* operator, which employs

```
stream1 = readFromWrapper("LINEORDER");
dim1 = read("CUSTOMER");
dim2 = read("PART");
dim3 = read("SUPPLIER");
dim4 = read("STORE");
tmp1 = stream1 -> window[rows n];
j1 = join f in tmp1, d1 in dim1
     where f.CustomerID == d1.CustomerID
     into {d1.CName, f.PartID, f.SupplierID,
           f.StoreID, f.ExtendedPrice};
j2 = join r in j1, d2 in dim2
     where r.PartID == d2.PartID
     into {r.CName, d2.PName, r.SupplierID,
           r.StoreID, r.ExtendedPrice};
j3 = join s in j2, d3 in dim3
     where s.SupplierID == d3.SupplierID
     into {s.CName, s.PName, d3.SuppName, s.
           StoreID, s.ExtendedPrice};
j4 = join t in j3, d4 in dim4
     where t.StoreID == d4.StoreID
     into {t.CName, t.PName, t.SuppName, d4.
           SArea, t.ExtendedPrice};
j4 -> istream -> cubify(config.conf);
```

Query 2. Continuous query over synthetic data.

a configuration file as a parameter. The configuration file contents are discussed in Sec. V. The initial execution of the *cubify* operator results in the formation of a lattice structure, while its latter execution updates the materialized lattice vertices with the incoming elements and generates output. Once a *Cubify CQ* is registered and is under execution, the user can submit OLAP queries to *Streaming*Cube. OLAP queries involve different OLAP operations, including the request of a particular lattice vertex, drill down, roll up, slice, and dice. At the time of writing this paper, *Streaming*Cube can accept OLAP querying with the following aggregate functions: 1) Sum, 2) Count, 3) Maximum, and 4) Minimum.

VII. INCREMENTAL STREAM PROCESSING AND VIEW MAINTENANCE

Computation, which updates the output incrementally rather than re-computing everything from scratch for successive runs of a job with input changes, is called incremental. Similarly, algorithms that compute changes to a view in response to changes to the base relations are called *incremental view maintenance* algorithms [49]. In *Streaming*Cube, incremental computation is used to update the synopses and materialized vertices of the query operators. A continuous query (CQ) in *Streaming*Cube is translated into a query plan and is executed continuously. Data streams enter *Streaming*Cube as a sequence of tuples, where each tuple is additionally assigned a timestamp and flagged as either an insertion "+" or deletion "−" to achieve incremental computation, as explained in the previous section.

Consider a *Cubify CQ* similar to Query 2; although, with two dimension tables, PART and CUSTOMER, as well as a fact stream, SALES. Its query plan tree is shown in the rectangular boundary in Fig. 2. The sales stream tuples with part ID (first value), customer ID (second value), and sales amount

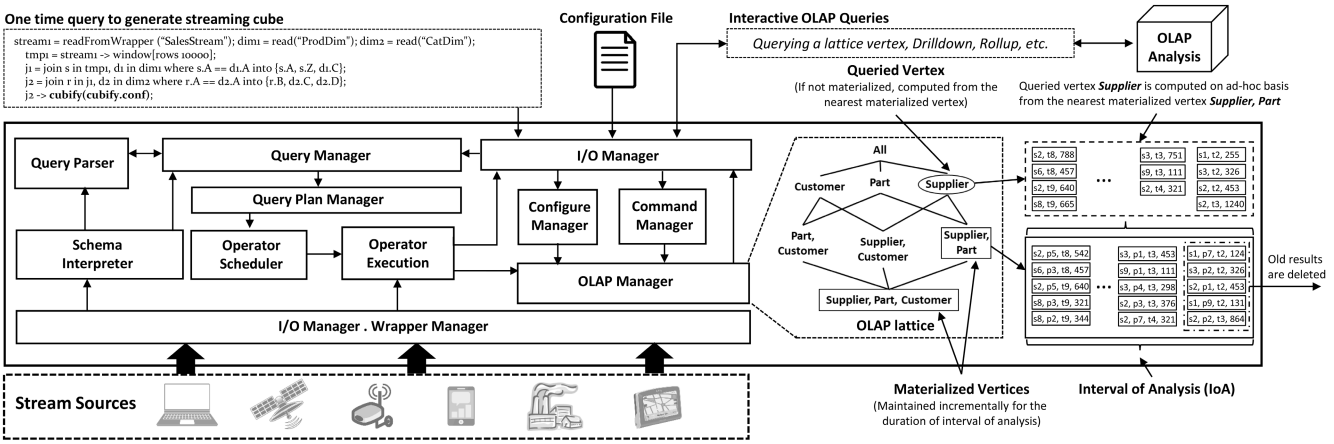


FIGURE 4. ***Streaming*Cube Architecture.**

(last value) ($< 1, 3, 25 >$, $< 2, 4, 10 >$, $< 2, 2, 40 >$, $< 3, 1, 30 >$) are appended with a timestamp and "+" flag on their arrival by *Streaming*Cube, thus, creating elements of the form $[< 1, 3, 25 >, t_2, +]$, $[< 2, 4, 10 >, t_1, +]$, $[< 2, 2, 40 >, t_1, +]$, $[< 3, 1, 30 >, t_1, +]$. The arriving elements are inserted into the window operator's synopsis and are also forwarded to the synopses of the downstream operators. Assuming that the window operator size (n) is 4 rows; hence, the next arrival of tuple causes the expiration of the oldest element, i.e., $[< 3, 1, 30 >, t_1, +]$. The window operator then deletes the oldest element from its synopsis and sends a corresponding "−" element, i.e., $[< 3, 1, 30 >, t_1, -]$, to the downstream operators causing the deletion of the oldest element from the downstream operator synopsis. After the join operations between the SALES stream and the two dimension tables PART and CUSTOMER, the resultant elements consist of PartName (denoted by Px) and CustomerName (denoted by Cx) of the dimensions PART and CUSTOMER, respectively. The join output is forwarded to the *cubify* operator via an i-stream operator. The i-stream operator generates a stream of elements: $[< P1, C3, 25 >, t_2, +]$, $[< P2, C4, 10 >, t_1, +]$, $[< P2, C2, 40 >, t_1, +]$, and $[< P3, C1, 30 >, t_1, +]$, which are received by the *cubify* operator. The elements are maintained as rows in the finest latex vertex synopsis at the δ_0 time granularity and are also used to update other materialized vertices. In the example of Fig. 2, the finest latex vertex is: {Part, Customer}, whereas the other materialized vertex is: {Part}.

To incrementally maintain the Cubify CQ operators' synopses following the *cubify* operator, the c-window is used. The c-window shares its synopsis with the lattice finest vertex. As the c-window moves, "+" and "−" elements corresponding to the rows in the c-window are generated and sent to the downstream operators that utilize these insertion (+) and deletion (−) elements to update their synopses incrementally, just like ordinary CQL operators. The only difference entails that the insertion and deletion elements are now being generated by the c-window operator rather than the ordinary window operator.

VIII. PROTOTYPE SYSTEM ARCHITECTURE

This section presents the prototype system architecture of *Streaming*Cube. Figure 4 shows the main components of the unified framework *Streaming*Cube. The **I/O Manager** is responsible for all the input and output in *Streaming*Cube, including the registration of CQs, OLAP queries, getting streams input, dimension tables, configuration files, sending out query results, etc. On receiving a CQ in the Jaql format, the **I/O Manager** forwards it to the **Query Manager** that gets the query parsed/translated into the query intermediate representation via the **Query Parser**. The **Query Parser** employs the streams and dimension table schema information to parse the CQ. The schema information is available to the **Query Parser** via the **Schema Interpreter**, which is supplied by the user. The schema information, herein referred to as *wrappers*, is also used by the **Wrapper Manager** for reading data streams and dimension tables. Once the query is parsed, it becomes available to the **Query Plan Manager**, which is responsible for the generation of a query execution plan that includes the assignment of synopses to the query operators and the connection of operators through queues. The query execution plan is in the form of a directed acyclic graph (DAG) of query operators and is sent to **Operator Scheduler**. The **Operator Scheduler** executes DAG operators continuously by selecting one operator at a time.

If the registered query is an *Ordinary CQ*, the query result is available to the user or application program via the **I/O Manager**. On the other hand, if the registered query is a *Cubify CQ*, the **OLAP Manager** receives a configuration file through the **Configure Manger** for the construction of the OLAP lattice and the materialization of selected lattice vertices. Furthermore, in *Cubify CQ*, the result of the *cubify* operator execution is available to the **OLAP Manager**, and it is used to incrementally maintain the materialized views.

Once the *Cubify CQ* is under execution, the user can register OLAP queries as HttpRequest, which are handled by the **Command Manager**. Upon receiving the OLAP query, the **OLAP Manager** checks if the queried vertex is materialized or not. The OLAP queries corresponding to the materialized vertices are answered directly, while in the other case they are computed from the nearest materialized vertex. In Figure 4, vertices with rectangular boundaries are materialized. In the figure, the vertex {*Supplier*} is being queried, which is not materialized. Therefore it is computed from the nearest materialized vertex {*Supplier, Part*} and the results are returned to the user via the *I/O Manager*.

IX. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of our proposed unified framework *Streaming*Cube and the *cubify* operator.

A. EXPERIMENTAL SETUP

1) PLATFORM

For experiments, the prototype system explained in Sec. VIII has been developed in C++ and is named as *Streaming*Cube [17]. *Streaming*Cube is available as open source at Github.[1] The experiments are performed on one of the nodes of an HP BladeSystem c7000 server, where each node is equipped with an Intel Xeon 20 core processor (ES-2650 v3 @ 2.3GHz), 6 GB RAM, and 10 Gbps Ethernet networking card. Each node is operated by Ubuntu 14.10 OS.

2) COMPARATIVE METHODS

To prove the effectiveness of *Streaming*Cube, we compared it with the *Streaming*Cube deployed as two separate engines/processes, i.e., SPE and OLAP on a single machine, hereafter called the *SPE+OLAP* framework. Both the SPE and OLAP engines are locally developed. In the SPE+OLAP framework, the SPE processes data streams and sends them to the OLAP engine via sockets. The OLAP engine generates lattice-materialized views and responds to OLAP queries. To keep the comparison fair, equal hardware resources are allocated to both frameworks.

3) DATA STREAM

Since it is extremely difficult to obtain real sales or similar datasets suitable for OLAP analysis due to data ownership and privacy issues, the following two datasets are used for the experiments: 1) Synthetic and 2) TPC-H[2] benchmark. Both datasets are multidimensional, where the dimensions are supplied as static tables and the fact tables are fed as continuous data streams to the two frameworks. The synthetic stream schema consists of six dimension tables, a fact table, and its lattice structure contains 64 vertices. The fact table SALES tuple is of the form $<$ *ProductID, SupplierID, PromotionID, CustomerID, StoreID, SalesPersonID, Timestamp, SalesAmount* $>$, where the IDs are generated randomly to reflect the IDs in the dimension tables PRODUCT, SUPPLIER, PROMOTION, CUSTOMER, STORE, and SALESPERSON. Timestamp (system time), which is part of all the lattice vertices and is not treated as a separate dimension, is appended to all the generated tuples by the respective frameworks.

TPC-H is a well-known benchmark dataset for testing OLAP systems. For the sake of experiments in this work, its schema is modified according to the Star Schema Benchmark (SSB) [50]. Furthermore, we added a store dimension, resulting in a four dimensional star schema with dimensions, CUSTOMER, PART, SUPPLIER, and STORE, as well as a fact table, LINEORDER, as shown in Fig. 15 (Appendix B). The lattice structure of the TPC-H schema consists of 16 vertices. The LINEORDER tuples are of the form $<$ *CustomerID, PartID, SupplierID, StoreID, Timestamp, ExtendedPrice* $>$, which are repeatedly fed to the experimental framework as data streams. Similar to the synthetic data stream, a timestamp (system time), which is part of all the lattice vertices and is not treated as a separate dimension, is appended to all the tuples by the respective experimental framework.

4) CONTINUOUS AND OLAP QUERIES

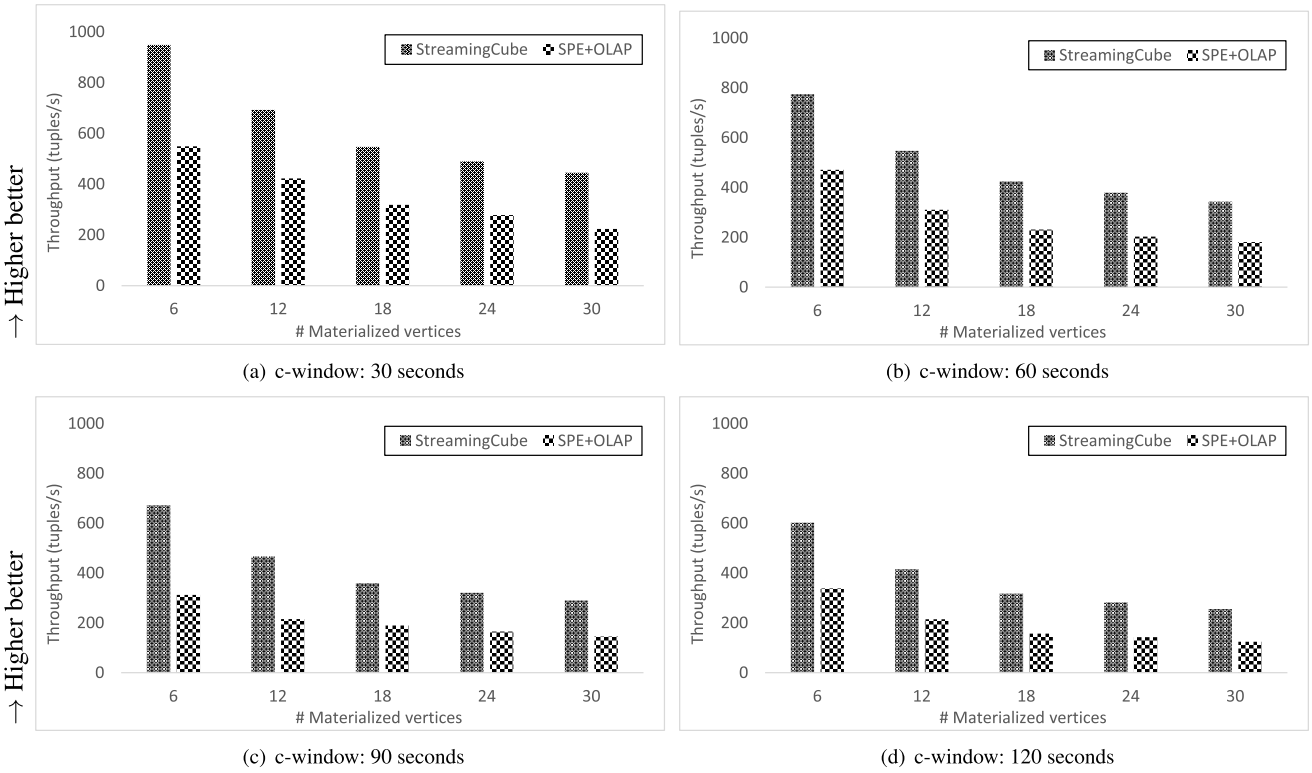
To evaluate the effectiveness of the proposed unified framework, two CQs are used, one for synthetic data stream and the other for the TPC-H data. Both queries are join queries between fact data streams and dimension tables, and they use a *cubify* operator to generate an in-memory OLAP lattice and maintain the selected materialized vertices. The *cubify* operator is also provided with a configuration file as a parameter. Query 2 shows one such queries written in Jaql [47] for TPC-H data.

To evaluate the OLAP querying cost, we issue a query corresponding to each lattice vertex. The average OLAP querying cost is computed by summing up the querying costs to query all the lattice vertices and dividing this sum by the total number of lattice vertices. In general, OLAP queries corresponding to the non-materialized vertices require more computation time than the materialized ones because the former needs to be computed from the latter on an ad-hoc basis.

B. EXPERIMENTAL RESULTS

The experimental evaluation is divided into three sections: The first section compares the performance of the proposed *Streaming*Cube with the SPE+OLAP system, and the second section evaluates the OLAP-querying cost, while the third section evaluates the *cubify* operator. The experimental results are presented by varying different important parameters. Unless stated otherwise, the following default parameter values are used in the experiments: the number of materialized vertices (# Materialized vertices) 12 (synthetic data) and 6 (TPC-H data), window size (n) 10,000 rows, the c-window 60 seconds. For the OLAP queries, aggregate function Sum is used.

[1] https://github.com/salmanahmedshaikh/StreamingCube
[2] TPC-H. http://www.tpc.org/tpch/



(a) c-window: 30 seconds

(b) c-window: 60 seconds

(c) c-window: 90 seconds

(d) c-window: 120 seconds

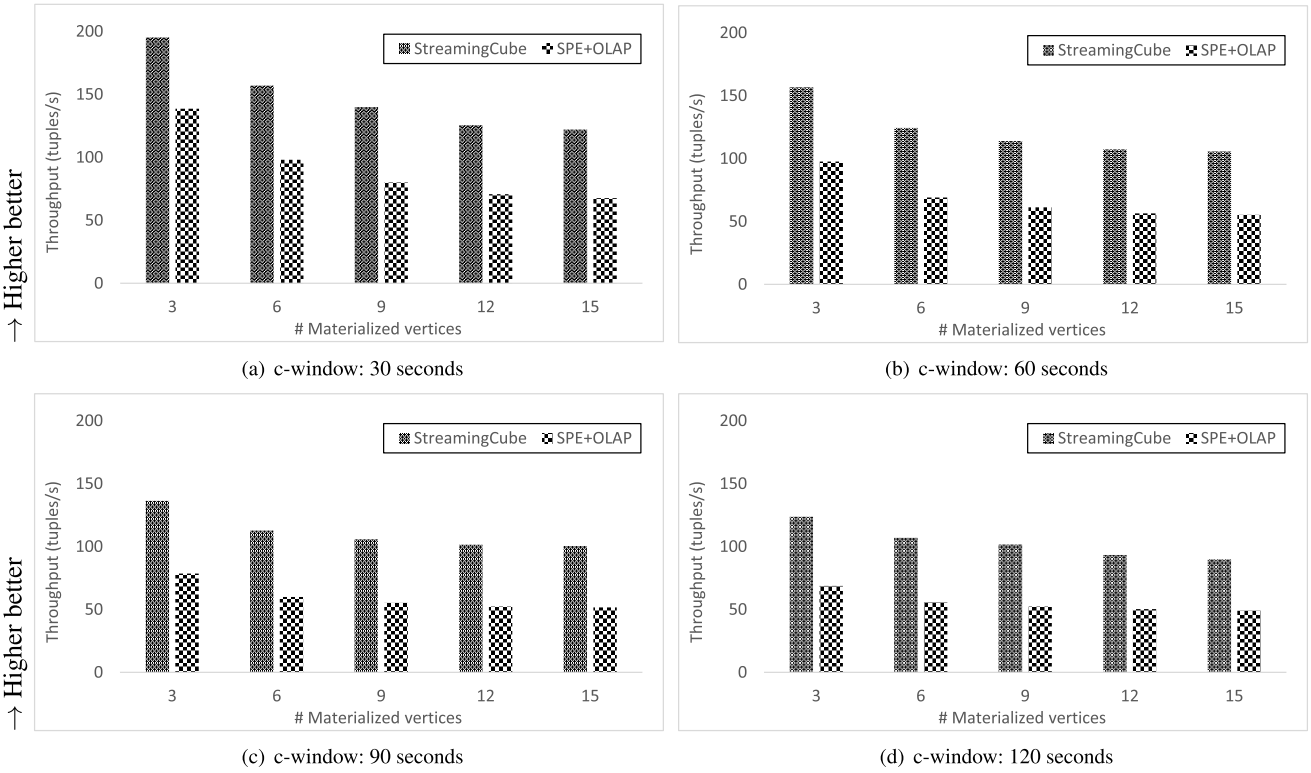
FIGURE 5. Throughput comparison over synthetic data (*Streaming*Cube vs. SPE+OLAP).

1) *STREAMING*CUBE VS. SPE+OLAP

In this section, we compare the throughput and memory consumption of our proposed *Streaming*Cube framework with the SPE+OLAP framework. Figs. 5 and 6 compare the throughput of the two frameworks for the synthetic and the TPC-H data streams, respectively, by varying the number of materialized vertices. Here, the throughput can be defined as the maximum number of tuples that can be processed by the system. For the synthetic data, the number of materialized vertices is varied from six to 30, whereas for the TPC-H data it is varied from three to 15. From the figures, *Streaming*Cube evidently outperforms SPE+OLAP for all the variations of the number of materialized vertices, i.e., the proposed framework can achieve better throughputs than by using separated SPEs and OLAP engines. The trends are quite similar for both data streams, which is because our proposed framework processes the data streams and maintains materialized vertices as a single process. However, in the SPE+OLAP framework, the output data from the SPE need to be sent to the OLAP engine, thereby incurring data transfer (between processes) and data transformation costs. Furthermore, it is observable from the figures that the throughput decreases with an increase in the number of materialized vertices, which is apparent as each materialized vertex maintains a data structure that must be updated on the arrival and departure of tuples.

It is noteworthy that the throughput shown in Figs. 5 and 6 seems lower than some of the general SPEs. This is due to several reasons: 1) Inclusion of the cubify operator, which performs several operations, including maintaining the data structure of the materialized views and responding to OLAP queries. In our experiments, 12 and 6 materialized vertices are maintained for OLAP cubes, corresponding to the synthetic and TPC-H data, respectively. 2) Execution of multiple joins between a fact table and dimension tables (where join is one of the most expensive database operations). For the synthetic and TPC-H data streams, 6 and 4 joins are performed between a fact stream and 6- and 4-dimension tables, respectively, in our experiments. 3) Absence of indices in our implementation since the main focus of this work is the unified framework via the use of the cubify operator. However, the throughput can be improved with the implementation of indices and putting more effort in the implementation, as in the case of state-of-the-art SPEs. Since the main focus of the experiments in this section is to prove the significance of the unified framework by comparing it with the SPE+OLAP framework, we did not concentrate on the implementation of indices.

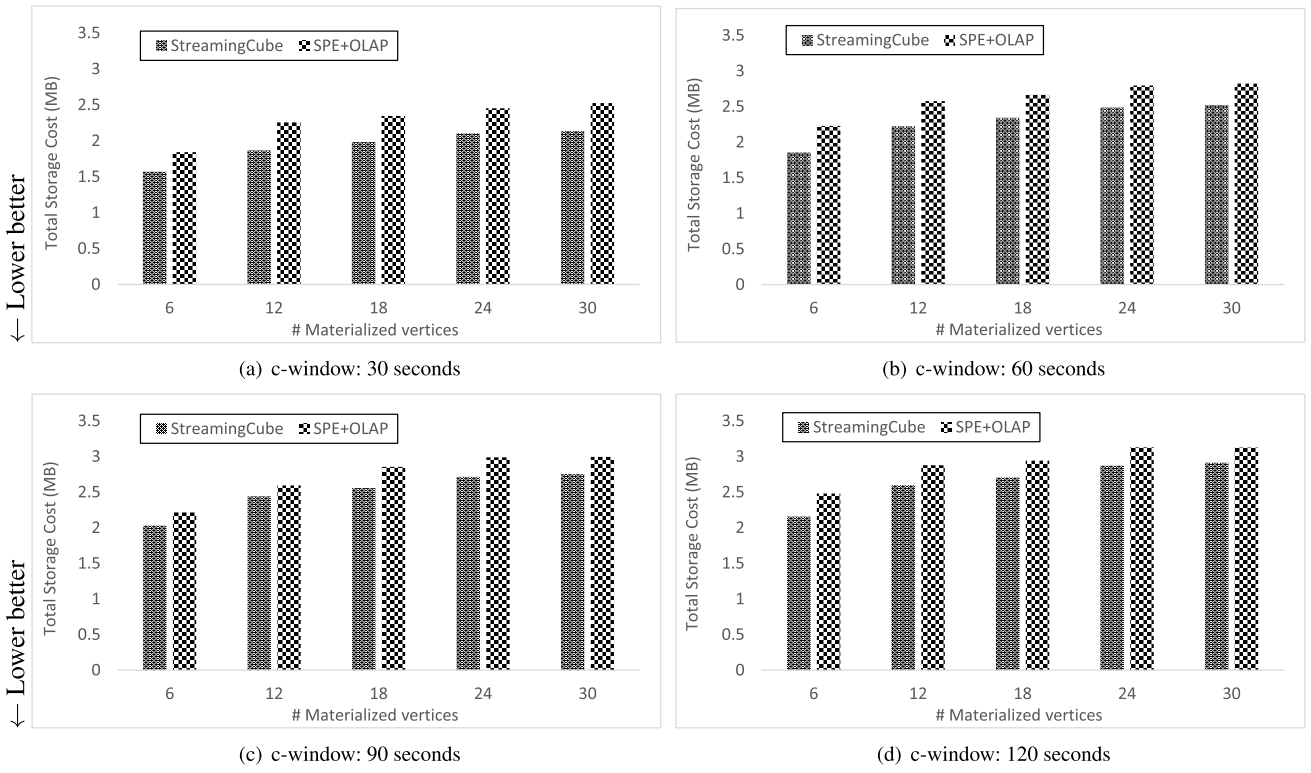
Notably also, in Figs. 5 and 6, the throughput of the TPC-H data is lower than that of the synthetic data although the number of vertices to materialize is lower for the TPC-H data. This is because the join processing between the fact and the dimension tables containing tuples of larger sizes takes a comparatively larger time, thereby ultimately resulting in smaller throughputs in the TPC-H data. Nonetheless, since the focus of this work is the unified framework rather than the



(a) c-window: 30 seconds (b) c-window: 60 seconds

(c) c-window: 90 seconds (d) c-window: 120 seconds

FIGURE 6. Throughput comparison over TPC-H data (*Streaming*Cube vs. SPE+OLAP).

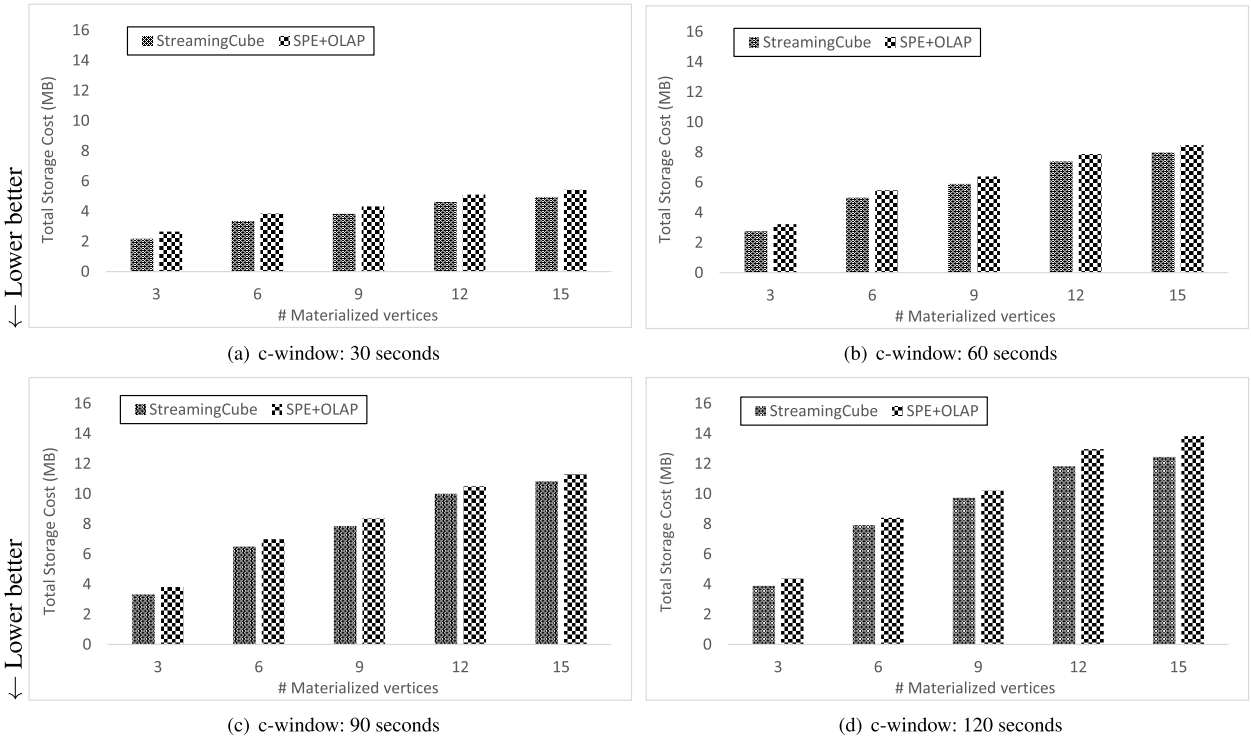


(a) c-window: 30 seconds (b) c-window: 60 seconds

(c) c-window: 90 seconds (d) c-window: 120 seconds

FIGURE 7. Storage cost comparison over synthetic data (*Streaming*Cube vs. SPE+OLAP).

join processing, we have not put much effort in this direction. However, the overall throughput for both data sets could be improved by incorporating state-of-the-art join processing algorithms and appropriate indices in the proposed framework. Furthermore, it is observable from Figs. 5 and 6 that the throughput decreases with an increase in the c-window



(a) c-window: 30 seconds (b) c-window: 60 seconds

(c) c-window: 90 seconds (d) c-window: 120 seconds

FIGURE 8. Storage cost comparison over TPC-H data (*Streaming*Cube vs. SPE+OLAP).

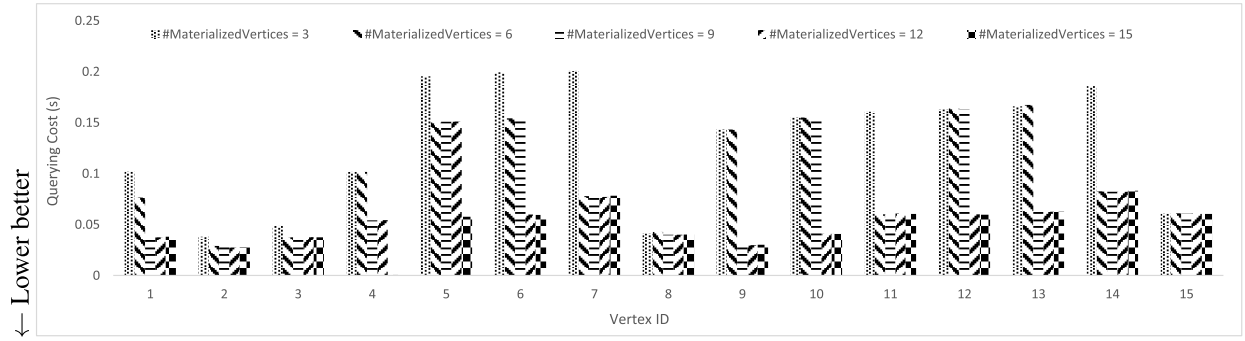
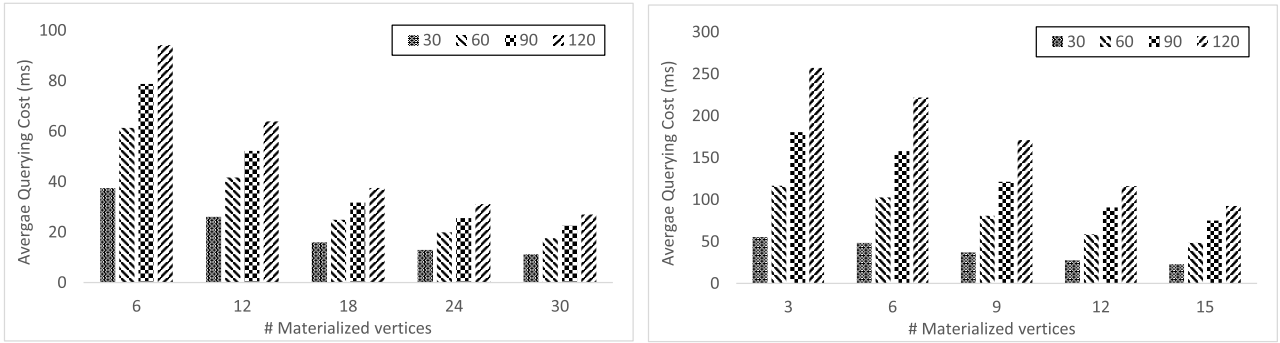


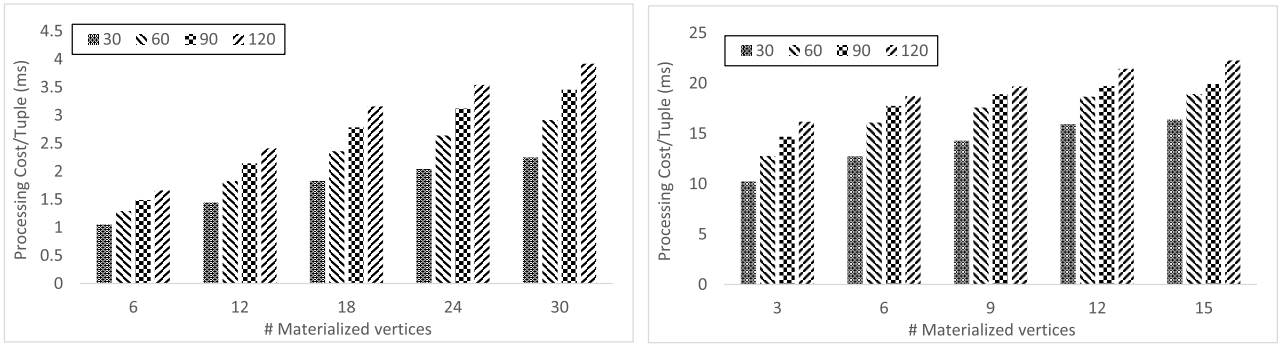
FIGURE 9. OLAP individual querying cost (TPC-H data).



(a) Synthetic data (b) TPC-H data

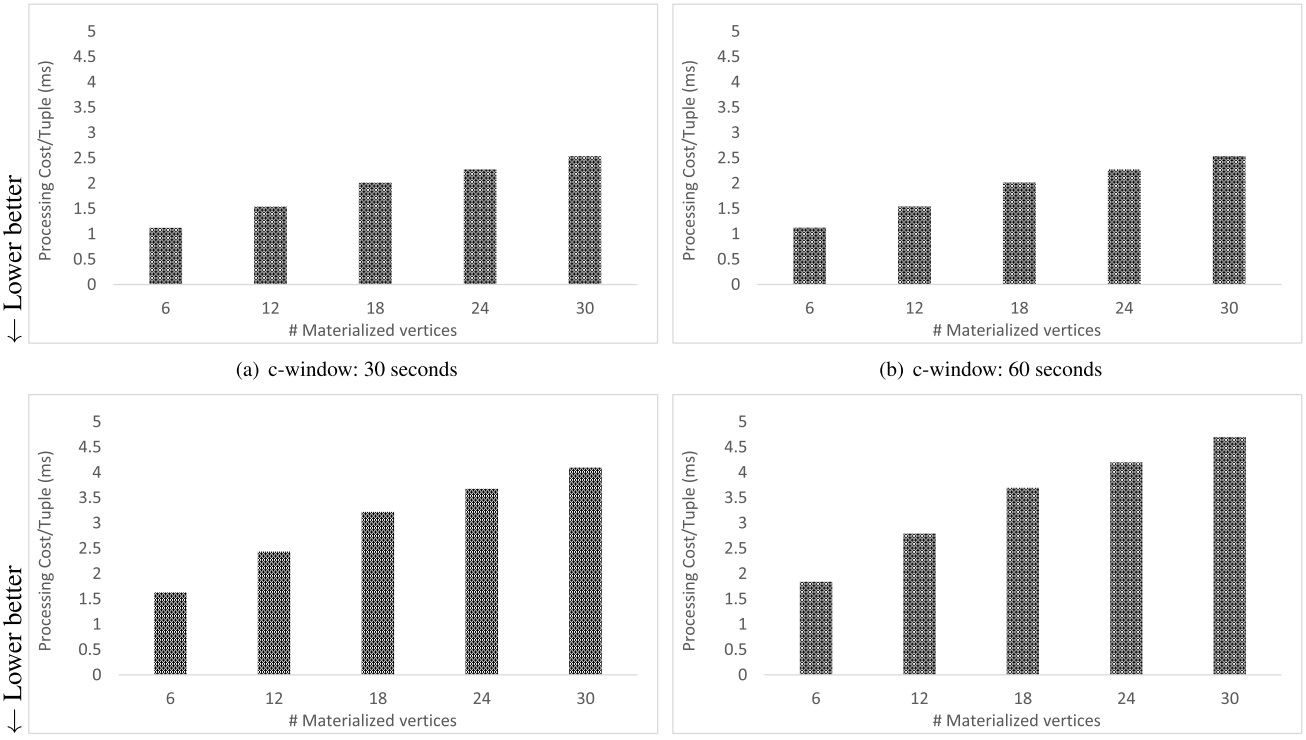
FIGURE 10. Average OLAP querying cost.

size for both data sets because as the size of the c-window increases, the number of tuples in the materialized vertices also increase. Hence, the time to insert and delete data to and from the data structures corresponding to the materialized vertices also increases, resulting in a slight decrease in the throughput.



(a) Synthetic data (b) TPC-H data

FIGURE 11. **Processing cost per tuple.**



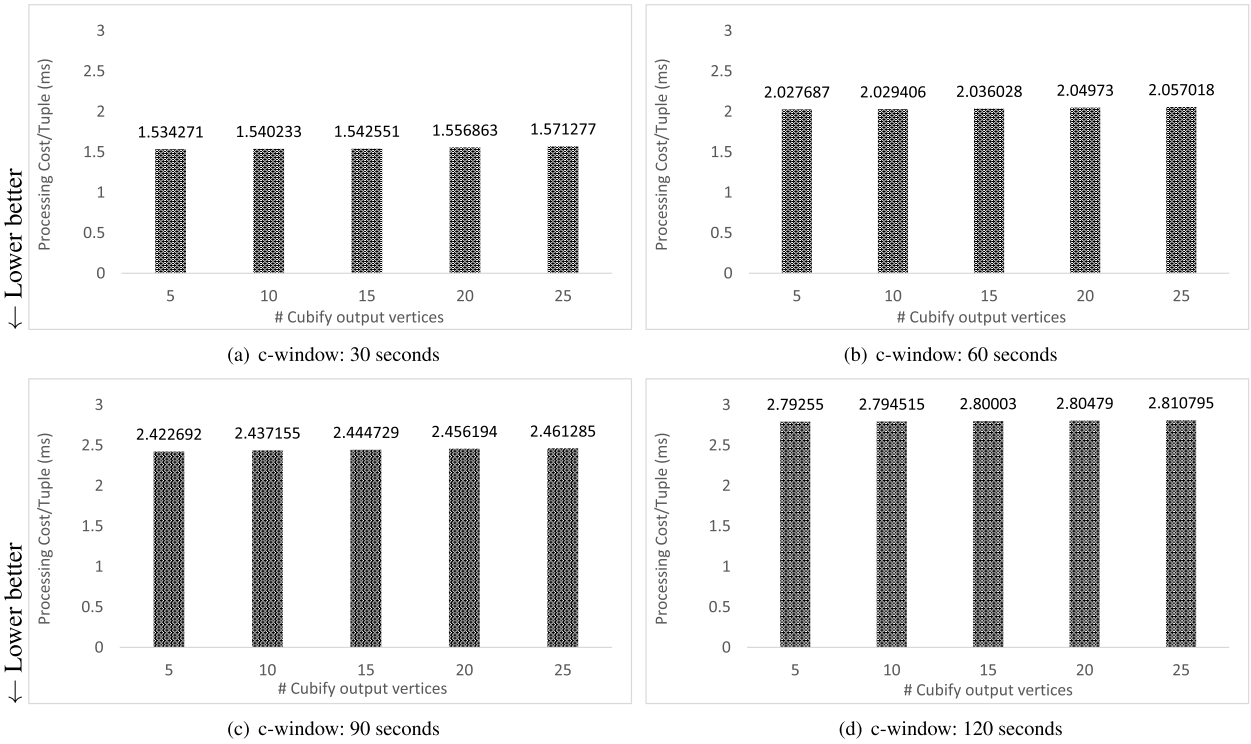
(a) c-window: 30 seconds (b) c-window: 60 seconds

(c) c-window: 90 seconds (d) c-window: 120 seconds

FIGURE 12. **Processing cost per tuple (by varying number of materialized vertices).**

Next, we compare the memory consumed in MB by the two frameworks in Figs. 7 and 8. The memory consumption is calculated by considering the storage utilized by the synopses of the query operators and the OLAP-materialized views. From the figures, it is evident that the proposed unified framework consumes less memory than the SPE+OLAP framework. This is because the SPE in the SPE+OLAP framework utilizes an additional operator, and therefore, an additional synopsis for generating query output. Furthermore, the OLAP engine requires some additional memory space for receiving data from the SPE and transforming them. Additionally, one can observe from Figs. 7 and 8 that memory consumption increases with an increase in the number of materialized vertices, which is evident as a separate data structure is maintained for each materialized vertex. Notably also in Figs. 7 and 8 is that the memory consumed by the TPC-H data is larger than the synthetic data although the number of vertices to materialize is smaller for the TPC-H data. This is because of the larger number and individual sizes of tuples in the TPC-H dimension tables.

Furthermore, we can observe from Figs. 7 and 8 that memory consumption increases with an increase in the c-window size. This is because with the increase in the c-window size, more tuples need to be maintained in the memory for the materialized vertices. Moreover, the memory consumption increment in Fig. 8 is steeper than in Fig. 7, which is due to the larger tuple sizes of the TPC-H dimension tables, as discussed above.



(a) c-window: 30 seconds

(b) c-window: 60 seconds

(c) c-window: 90 seconds

(d) c-window: 120 seconds

FIGURE 13. Processing cost per tuple (by varying the number of cubify output vertices).

2) OLAP QUERYING COST

This section evaluates the OLAP-querying cost and the processing cost per tuple in *Streaming*Cube. Figs. 9 and 10 present the OLAP-querying costs. OLAP queries include requesting a particular lattice vertex, drill down, roll up, slice, and dice. Fig. 9 shows the querying cost of each lattice vertex (including materialized and non-materialized vertices) for a different number of materialized vertices for the TPC-H data. The TPC-H lattice contains 16 vertices (practically 15 vertices as the 16th vertex does not contain any dimension). If a queried vertex is non-materialized, it is computed from one of the nearest materialized vertices by performing aggregation, which is computationally expensive than directly querying the materialized vertex. Fig. 9 shows that the querying cost is higher for most of the queried vertices when the number of materialized vertices = 3. The querying cost decreases gradually with an increase in the number of materialized vertices due to the materialization of either the queried vertex or some nearby vertices.

Figs. 10(a) and 10(b) present the average OLAP-querying costs for the synthetic and TPC-H datasets for different values of the c-window size by varying the number of materialized vertices. The average OLAP-querying cost is computed by summing up the querying costs for all the lattice vertices and dividing this sum by the total number of lattice vertices. In both figures, the average querying cost decreases with an increase in the number of materialized vertices, which is apparent and has been explained earlier. Furthermore, it is noticeable that as the c-window increases, the average querying cost rises, which is due to the increase in the number of tuples in the materialized vertices. Comparing Figs. 10(a) and 10(b), the average querying cost is higher for TPC-H data, because it is a benchmark dataset with near-real dimensional attribute values, i.e., larger tuple sizes. Hence, queries need to fetch and process larger amounts of data, thereby resulting in higher querying costs.

Fig. 11 presents the processing cost per tuple of our proposed framework, which rises with an increase in the number of materialized vertices, as the arrival and departure of each tuple must update all the materialized vertices. Similarly, the processing cost per tuple increases as the c-window size enlarges because the size of the data structure and the materialized vertex are directly proportional, which causes each input tuple to require more processing time.

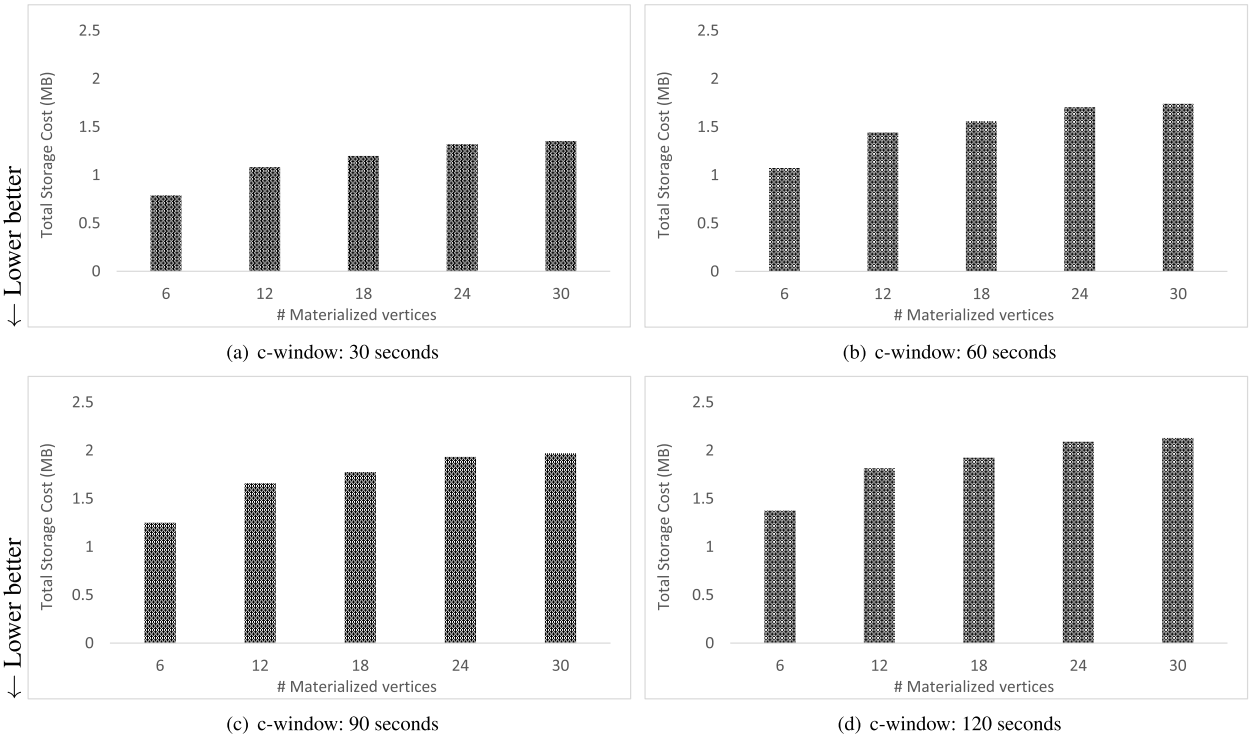
```
stream =readFromWrapper ("PreJoinedStream");
stream -> cubify(config.conf) -> istream;
```

Query 3. CQ with cubify output stream.

3) THE CUBIFY OPERATOR

To evaluate the *cubify* operator, we utilized Query 3. It is a *Cubify CQ* with a *cubify* and an *i-stream* operators, and employs a synthetic data stream. The data stream used in this section is the same one used in earlier experiments. However, we pre-joined it with dimensions to effectively evaluate the *cubify* operator.

Figs. 12 and 13 reveal the processing cost per tuple in the *cubify* operator for the different values of the c-window size by varying the number of materialized vertices and the number of cubify output vertices, respectively. A cubify output



(a) c-window: 30 seconds

(b) c-window: 60 seconds

(c) c-window: 90 seconds

(d) c-window: 120 seconds

FIGURE 14. The cubify operator storage cost.

vertex is an OLAP lattice vertex selected to generate continuous output. The generated output is aggregated with respect to the lattice vertex schema and time unit δ. From Fig. 12, the processing cost per tuple of the cubify operator increases with an increase in the number of materialized vertices, which is due to the increase in the number of synopses that must be maintained. Furthermore, the processing cost increases with the increase in the c-window size, which is due to an increase in the size of each synopsis; hence, requiring larger updating times.

Similarly, the processing cost per tuple rises with an increase in the number of cubify output vertices, as evident in Fig. 13. The cost of output generation from a lattice vertex depends on whether the requested vertex is materialized or otherwise. If the vertex is not materialized, then its aggregated output stream is computed from the nearest materialized vertex. In the experiments, we attempted to select an equal number of materialized and non-materialized cubify output vertices. As the number of the cubify output vertices increases, the number of elements to be generated from these vertices also rises, thereby resulting in higher processing costs per tuple.

Next, we evaluated the *cubify* operator's storage cost in Fig. 14 by varying the number of materialized vertices. Since each materialized vertex requires a dedicated synopsis for its maintenance, the cubify operator's storage cost rises with an increase in the number of materialized vertices. Similar trends can be observed for all the values of the c-window in Fig. 14. Notably also is that the storage cost increases as the c-window sizes expand because larger c-window sizes can accommodate more tuples; therefore, requiring more substantial storage.

X. CONCLUSION AND FUTURE WORK

Herein, we present a unified framework *Streaming*Cube for stream processing and OLAP analysis for applications with a mixture of streaming and analytical workloads. The proposed framework supports *Cubify CQs* in addition to the ordinary CQs. To maintain the OLAP lattice materialized views incrementally, a *cubify* operator is introduced. In addition to CQs, *Streaming*Cube supports OLAP queries, including drill down, roll up, slice, and dice. Furthermore, the *cubify* operator can generate an aggregated output of the selected lattice vertices. Detailed experimental evaluation over synthetic and TPC-H data proves that the unified framework results in higher system throughputs and consumes less memory than the hybrid framework when provided with equal resources. Furthermore, the proposed framework requires less maintenance cost by avoiding the deployment of two dedicated engines, i.e., SPE and OLAP. In addition, the unified *Streaming*Cube saves communication costs that may have otherwise been required to transfer data between the two engines/processes.

In the future, we plan to extend this work in the following directions. In the current framework, the dimension tables are static; however, in many applications dimension tables need to be updated. Although the update frequency of dimension tables may be small, we believe that it is an important issue to address. There are two possible ways to achieve this: 1) New data overwrite the existing data in the dimension table.

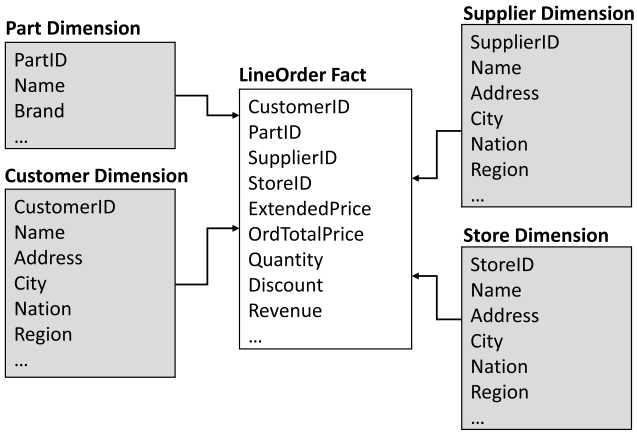


FIGURE 15. Modified TPC-H benchmark schema (star schema).

TABLE 1. Star Schema Benchmark (SSB) lattice vertices.

Level	Vertex ID	Vertex
0	0	all
1	1	Customer
	2	Part
	3	Supplier
	4	Store
2	5	Customer, Part
	6	Customer, Supplier
	7	Customer, Store
	8	Part, Supplier
	9	Part, Store
	10	Supplier, Store
3	11	Customer, Part, Supplier
	12	Customer, Part, Store
	13	Customer, Supplier, Store
	14	Part, Supplier, Store
4	15	Customer, Part, Supplier, Store

Thus, the existing data are lost as they are not backed up anywhere else. Hence, both the historical and the new data refer to the same updated dimension tuples. 2) A new tuple is added for the updated dimension tuple, while the old tuple is retained in the dimension table. Each tuple contains the effective time and expiration time to identify the time period wherein the tuple was active. This enables historical data to refer to the old tuple and new data to the new tuple. Another interesting future direction is the distributed processing of *Streaming*Cube using state-of-the-art stream processing platforms, i.e., Apache Flink or Apache Spark.

APPENDIX A
SAMPLE CONFIGURATION FILE

#Dimension attributes with hierarchy
Dimensions = productID, productName | supplierID, supplierName, supplierAddress | customerID, customerName, customerAddress
#Vertices to materialize
MVertices = productID, Minute | productName, supplierName, Hour
#Finest vertex materialization time granularity
TimeGrain = Minute
#c-window size (The unit is same as that of TimeGrain)
CWindow = 500
#Lattice output vertices
OutputVertices = productName, customerName

APPENDIX B
EXAMPLE OF STAR SCHEMA (MODIFIED TPC-H BENCHMARK SCHEMA)

See Fig. 15.

APPENDIX C
LIST OF VERTICES (STAR SCHEMA BENCHMARK)

See Table 1.

REFERENCES

[1] *The Apache Software Foundation, Apache Storm*. Accessed: Feb. 23, 2017. [Online]. Available: http://storm.apache.org/
[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, Feb. 2015.
[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "STREAM: The Stanford data stream management system," in *Data Stream Management: Processing High-Speed Data Streams*, M. Garofalakis, J. Gehrke, and R. Rastogi, Eds. Berlin, Germany: Springer, 2016, pp. 317–336, doi: 10.1007/978-3-540-28608-0_16.
[4] D. J. Abadi, D. Carney, U. Eetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *VLDB J. Int. J. Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, Aug. 2003, doi: 10.1007/s00778-003-0095-z.
[5] U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J.-H. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik, *The Aurora and Borealis Stream Processing Engines*. Berlin, Germany: Springer, 2016, pp. 337–359.
[6] A. Samza. *The Apache Software Foundation*. Accessed: Aug. 25, 2017. [Online]. Available: http://samza.apache.org/
[7] T. Akidau, A. Balikov, K. Bekiroülu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: Fault-tolerant stream processing at Internet scale," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013.
[8] J. Fang, R. Zhang, X. Wang, and A. Zhou, "Distributed stream join under workload variance," *World Wide Web*, vol. 20, no. 5, pp. 1089–1110, Sep. 2017.
[9] S. A. Shaikh, Y. Watanabe, Y. Wang, and H. Kitagawa, "Smart scheme: An efficient Query execution scheme for event-driven stream processing," *Knowl. Inf. Syst.*, vol. 58, no. 2, pp. 341–370, Feb. 2019.
[10] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai, "Stream cube: An architecture for multi-dimensional analysis of data streams," *Distrib. Parallel Databases*, vol. 18, no. 2, pp. 173–197, Sep. 2005.
[11] A. Cuzzocrea, A. Schuster, G. Vercelli, and M. Nolich, "Privacy-preserving olap-based monitoring of data streams: The PP-OMDS approach," in *Proc. 27th Italian Symp. Adv. Database Syst., Castiglione della Pescaia (Grosseto)*, Rome, Italy, Jun. 2019, pp. 282–292.
[12] K. Nakabasami, T. Amagasa, S. A. Shaikh, F. Gass, and H. Kitagawa, "An architecture for stream OLAP exploiting SPE and OLAP engine," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2015, pp. 319–326.
[13] Hitachi. *μCosminexus Stream Data Platform*. Accessed: Aug. 10, 2017. [Online]. Available: http://www.hitachi.co.jp/Prod/comp/soft1/cosminexus/sdp/
[14] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav, "Snappydata: A unified cluster for streaming, transactions and interactice analytics," in *Proc. CIDR*, 2017, pp. 1–8.
[15] *The Apache Software Foundation, Apache Spark—Lightning-Fast Cluster Computing*. Accessed: Feb. 17, 2017. [Online]. Available: http://spark.apache.org/
[16] M. Use of technology. *The Internet in 60 Seconds*. Accessed: Sep. 5, 2017. [Online]. Available: http://www.makeuseof.com/tag/the-internet-in-60-seconds/

[17] S. A. Shaikh and H. Kitagwa, "Streamingcube: A unified framework for stream processing and OLAP analysis (demo paper)," in *Proc. 26th ACM Int. Conf. Inf. Knowl. Manage.*, 2017, pp. 2527–2530, doi: 10.1145/3132847.3133165.

[18] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. IEEE Int. Conf. Data Mining Workshops*, Dec. 2010, pp. 170–177.

[19] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1996, pp. 205–216.

[20] K. Aouiche and J. Darmont, "Data mining-based materialized view and index selection in data warehouses," *J. Intell. Inf. Syst.*, vol. 33, no. 1, pp. 65–93, Aug. 2009.

[21] Z. A. Talebi, R. Chirkova, Y. Fathi, and M. Stallmann, "Exact and inexact methods for selecting views and indexes for OLAP performance improvement," in *Proc. 11th Int. Conf. Extending Database Technol. Adv. Database Technol. (EDBT)*, 2008, pp. 311–322.

[22] C. Joslyn, J. Burke, T. Critchlow, N. Hengartner, and E. Hogan, "View discovery in OLAP databases through statistical combinatorial optimization," in *Proc. the 21st Int. Conf. Sci. Stat. Database Manage.*, 2009, pp. 37–55.

[23] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung, "OLAP on sequence data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2008, pp. 649–660, doi: 10.1145/1376616.1376682.

[24] W. Feng, C. Zhang, W. Zhang, J. Han, J. Wang, C. Aggarwal, and J. Huang, "STREAMCUBE: Hierarchical spatio-temporal hashtag clustering for event exploration over the Twitter stream," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Apr. 2015, pp. 1561–1572.

[25] *MemSQL: The Fastest In-Memory Database*. Accessed: Feb. 14, 2017. [Online]. Available: http://www.memsql.com/

[26] *The Apache Software Foundation*. Accessed: Feb. 14, 2017. [Online]. Available: http://cassandra.apache.org/

[27] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim, "L-store: A real-time OLTP and OLAP system," in *Proc. CoRR*, 2016, pp. 1–15.

[28] M. Zaharia, P. Wendell, A. Konwinski, and H. Karau, *Learning Spark*. Newton, MA, USA: O'Reilly Media, 2015.

[29] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha, "DBToaster: higher-order delta processing for dynamic, frequently fresh views," *VLDB J.*, vol. 23, no. 2, pp. 253–278, Apr. 2014.

[30] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou, "Streaming multiple aggregations using phantoms," *VLDB J.*, vol. 19, no. 4, pp. 557–583, Aug. 2010.

[31] M. Nikolic, M. Dashti, and C. Koch, "How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 511–526.

[32] G. Luo and P. S. Yu, "Content-based filtering for efficient online materialized view maintenance," in *Proc. 17th ACM Conf. Inf. Knowl. Mining (CIKM)*, 2008, pp. 163–172.

[33] L. Fegaras, "Incremental stream processing of nested-relational queries," in *Database and Expert Systems Applications*, S. Hartmann and H. Ma, Eds. Cham, Switzerland: Springer, 2016, pp. 305–320.

[34] Y. Yang, L. Golab, and M. Tamer Ozsu, "ViewDF: Declarative incremental view maintenance for streaming data," *Inf. Syst.*, vol. 71, pp. 55–67, Nov. 2017.

[35] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk, "Stream warehousing with datadepot," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2009, pp. 847–854, doi: 10.1145/1559845.1559934.

[36] M. Ahuja, C. C. Chen, R. Gottapu, J. Hallmann, W. Hasan, R. Johnson, M. Kozyrczak, R. Pabbati, N. Pandit, S. Pokuri, and K. Uppala, "Peta-scale data warehousing at yahoo!" in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2009, pp. 855–862, doi: 10.1145/1559845.1559935.

[37] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee, "Moirae: History-enhanced monitoring," in *Proc. 3rd CIDR Conf.*, 2007, pp. 213–217.

[38] K. Tufte, J. Li, D. Maier, V. Papadimos, R. L. Bertini, and J. Rucker, "Travel time estimation using niagarast and latte," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2007, pp. 1091–1093, doi: 10.1145/1247480.1247617.

[39] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre, "Continuous analytics: Rethinking Query processing in a network-effect world," in *Proc. CIDR*, 2009, pp. 1–10.

[40] L. Golab and T. Johnson, "Consistency in a stream warehouse," in *Proc. CIDR*, Jan. 2011, pp. 114–122.

[41] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. K. Gupta, and R. Chen, "The niagara Internet Query system," *IEEE Data Eng. Bull.*, vol. 24, pp. 27–33, Oct. 2001.

[42] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, Jun. 2006, doi: 10.1007/s00778-004-0147-z.

[43] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, New York, NY, USA, 2002, pp. 1–16, doi: 10.1145/543613.543615.

[44] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in SQL databases," in *Proc. 26th Int. Conf. Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann, 2000, pp. 496–505. [Online]. Available: http://dl.acm.org/citation.cfm?id=645926.671701

[45] H. Gupta and I. S. Mumick, "Selection of views to materialize in a data warehouse," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 1, pp. 24–43, Jan. 2005, doi: 10.1109/TKDE.2005.16.

[46] C. A. Dhote and M. S. ALi, "Materialized view selection in data warehousing," in *Proc. 4th Int. Conf. Inf. Technol. (ITNG)*, Apr. 2007, pp. 843–847.

[47] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita, "Jaql: A scripting language for large scale semistructured data analysis," in *VLDB J.*, vol. 4, no. 12, pp. 1272–1283, 2011.

[48] *JSON—JavaScript Object Notation*. Accessed: Sep. 30, 2017. [Online]. Available: http://www.json.org/

[49] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1993, pp. 157–166.

[50] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *Proc. 1st TPC Technol. Conf. Perform. Eval. Benchmarking (TPCTC)*. Berlin, Germany: Springer-Verlag, 2009, pp. 237–252, doi: 10.1007/978-3-642-10424-4_17.

SALMAN AHMED SHAIKH received the B.E. degree in computer systems and the M.E. degree in communication systems and networks from the Mehran University of Engineering and Technology, Pakistan, in 2005 and 2008, respectively, and the Ph.D. degree in computer science from the University of Tsukuba, Japan, in 2014. From April, 2014 to March 2018, he worked as a Postdoctoral Researcher with the University of Tsukuba, Japan, on a Big Data Integration and Analysis Project, Research and Development on Real World Big Data Integration and Analysis, funded by Ministry of Education, Culture, Sports, Science, and Technology, Japan. He is currently working as a Research Scientist with the Artificial Intelligent Research Center (AIRC), AIST, Tokyo Waterfront, Japan. His research interests include stream processing, point cloud data processing, spatial data processing, big data manipulation, transaction processing, uncertain data processing, and data mining. He is a member of the Database Society of Japan (DBSJ), the International Association of Computer Science and Information Technology (IACSIT), and Pakistan Engineering Council (PEC).

HIROYUKI KITAGAWA (Member, IEEE) received the B.Sc. degree in physics and the M.Sc. and Dr.Sc. degrees in computer science from the University of Tokyo, in 1978, 1980, and 1987, respectively. He is currently a Full Professor with the Center for Computational Sciences, University of Tsukuba. His research interests include databases, data integration, stream processing, data mining, social media mining, information retrieval, and scientific databases. He is a member of ACM and JSSST. He is an IEICE Fellow, an IPSJ Fellow, and an Associate Member of the Science Council of Japan. He has served as the President of the Database Society of Japan, from 2014 to 2016, the Chairperson of the IEICE Special Interest Group on Data Engineering, from 1999 to 2001, and the Chairperson of ACM SIGMOD Japan Chapter, from 2003 to 2007.