Supporting Views in Data Stream Management Systems

THANAA M. GHANEM
University of St. Thomas
AHMED K. ELMAGARMID
Purdue University
PER-ÅKE LARSON
Microsoft Research
and
WALID G. AREF
Purdue University

In relational database management systems, views supplement basic query constructs to cope with the demand for "higher-level" views of data. Moreover, in traditional query optimization, answering a query using a set of existing materialized views can yield a more efficient query execution plan. Due to their effectiveness, views are attractive to data stream management systems. In order to support views over streams, a data stream management system should employ a closed (or composable) continuous query language. A closed query language is a language in which query inputs and outputs are interpreted in the same way, hence allowing query composition.

This article introduces the Synchronized SQL (or `SyncSQL`) query language that defines a data stream as a sequence of modify operations against a relation. `SyncSQL` enables query composition through the unified interpretation of query inputs and outputs. An important issue in continuous queries over data streams is the frequency by which the answer gets refreshed and the conditions that trigger the refresh. Coarser periodic refresh requirements are typically expressed as sliding windows. In this article, the sliding window approach is generalized by introducing the synchronization principle that empowers `SyncSQL` with a formal mechanism to express queries with arbitrary

T. M. Ghanem and A. K. Elmagarmid's research was partially supported by Lilly Endowment, NSF-ITR 0428168, NSF IIS 9983249, and US DHS PURVAC. W. G. Aref's research was partially supported by NSF Grant Number IIS-0811954.
Authors' addresses: T. M. Ghanem, Department of Computer and Information Sciences, University of St. Thomas-Minnesota, OSS 402, 2115 Summit Avenue, Saint Paul, Minnesota 55105; email: ghan6402@stthomas.edu; A. K. Elmagarmid, Department of Computer Science, Purdue University, Office LWSN 2142A, West Lafayette, IN 47907; email: ake@cs.purdue.edu; P.-Å. Larson, Microsoft Research, One Microsoft Way, Redmond, WA 98052-6399; email: palarson@microsoft.com; W. G. Aref, Department of Computer Science, Purdue University, West Lafayette, IN 47907; email: aref@cs.purdue.edu.

DOI 10.1145/1670243.1670244 http://doi.acm.org/10.1145/1670243.1670244

refresh conditions. After introducing the semantics and syntax, we lay the algebraic foundation for `SyncSQL` and propose a query-matching algorithm for deciding containment of `SyncSQL` expressions. Then, the article introduces the `Nile-SyncSQL` prototype to support `SyncSQL` queries. `Nile-SyncSQL` employs a pipelined incremental evaluation paradigm in which the query pipeline consists of a set of differential operators. A cost model is developed to estimate the cost of `SyncSQL` query execution pipelines and to choose the best execution plan from a set of different plans for the same query. An experimental study is conducted to evaluate the performance of `Nile-SyncSQL`. The experimental results illustrate the effectiveness of `Nile-SyncSQL` and the significant performance gains when views are enabled in data stream management systems.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query language*

General Terms: Language, Experimentation, Performance

Additional Key Words and Phrases: Data streams, query language, expression matching, incremental evaluation, views

ACM Reference Format:

Ghanem, T. M., Elmagarmid, A. K., Larson, P.-Å., and Aref, W. G. 2010. Supporting views in data stream management systems. ACM Trans. Datab. Syst, 35, 1, Article 1 (February 2010), 47 pages. DOI = 10.1145/1670243.1670244 http://doi.acm.org/10.1145/1670243.1670244

1. INTRODUCTION

In relational database management systems, views on database tables provide a basic query construct to cope with the demand for "higher-level" views over the base data. A view defines a function from a set of base tables to a derived table. The derived table (or the view) can be used as input to other functions or queries. Views are needed because the actual schema of the database is usually normalized for various reasons and queries are more intuitive using one or more denormalized relations that better represent the real world. At the same time, the emergence of data streaming applications calls for new data management technologies to cope with the characteristics of *continuous* data streams. Examples of data streaming applications include: environmental and road traffic monitoring through sensors, online data feeds [Chen et al. 2000], and online analysis of network traffic [Cranor et al. 2003]. A data stream is defined as a continuous sequence of tuples. Unlike traditional snap-shot queries over data tables, queries over data streams are continuous. A continuous query is issued once and may remain active for hours or days. The answer to a continuous query is constructed progressively as new input stream tuples arrive. To support views over data streams means the ability to express derived streams as a function of one or more input streams. The derived streams are then used as inputs to other continuous queries.

To support views is an attractive property for data stream management for the following reasons.

(1) *More intuitive query expressions.* Data streams are usually received from a distributed set of data sources (e.g., sensors). A query is more intuitive if expressed using a derived stream (or a view) that better represents the real world. The view can be expressed as a function over one or more input streams.

(2) *Answering multiple (concurrent) continuous queries using views.* Views can be beneficial in streaming environments that are characterized by a large number of concurrent overlapping queries. For a set of overlapping queries, a view can be defined to represent the overlapped part among the queries. Then, the shared execution of the overlapped part can be used in optimizing the query execution cost.
(3) *Data privacy.* An input stream may contain attributes or tuples that should not be seen by a certain group of users. Restricted access to stream attributes can be achieved by defining a view that projects out the private attributes. Then, users are given access only to the view. Multiple views can be defined depending on the privileges of the different user groups.
(4) *Answering ad hoc queries over data streams*. In addition to continuous queries, ad hoc queries over data streams form another important class of queries in Data Stream Management Systems (DSMS). An ad hoc query over data streams is interested in knowing the current status of the underlying streams. A view can be continuously maintained and is used in answering an ad hoc query that is frequently issued.

In order to support views over streams, a DSMS should employ a *closed* (or composable) continuous query language. A closed query language is a language in which query inputs and outputs are interpreted in the same way, hence allowing query composition. Query composition means the ability to express a query in terms of one or more subqueries (or views). In this article, we propose the `Synchronized SQL` query language (`SyncSQL` for short), a *closed* query language that enables supporting views over streams. We introduce the `Nile-SyncSQL` prototype server that supports `SyncSQL`, and hence supports views over data streams. We evaluate the performance of `Nile-SyncSQL` via an extensive set of experiments. The experimental results illustrate that views over streams have a tremendous effect on the performance of a DSMS.

1.1 New Challenges to Continuous Query Languages

A closed continuous query language is needed in order to support query composition, and hence to support views over streams. Query composition is a fundamental property of query languages (e.g., SQL), and it requires that query inputs and outputs be interpreted in the same way. The current approaches for continuous query languages (e.g., Arasu et al. [2006], Abadi et al. [2003], Chandrasekaran et al. [2003], Cranor et al. [2003]) suffer from the following: (1) Semantics of query composition is complex and it is difficult to understand the exact meaning of the composed queries, and (2) the set of queries that can be composed is limited. In this article, we propose the `SyncSQL` query language that gives a general and clear semantics for continuous query composition, and hence gives a framework for supporting views over streams. Basically, we address the following challenges.

Challenge 1: *Using streams to represent the output of continuous queries that produce nonappend-only output.* A continuous query may not be able to produce an append-only output relation even when the input streams represent append-only relations. For example, consider an application that monitors a parking lot where two sensors continuously monitor the lot's entrance and exit. The sensors generate two streams of identifiers, say S_1 and S_2, for vehicles entering and exiting the lot, respectively. A reasonable query in this environment is P_1:*"Continuously keep track of the identifiers of all vehicles inside the parking lot"*. The answer to P_1 is a *view* that at any time point, say T, contains the identifiers of vehicles that are inside the parking lot. S_1 can be modeled as a stream that inserts tuples into an append-only relation, say $\Re(S_1)$, and similarly, S_2 inserts tuples into the append-only relation $\Re(S_2)$. Then, P_1 can be regarded as a *materialized view* that is defined by the set-difference between the two relations $\Re(S_1)$ and $\Re(S_2)$. As tuples arrive into S_1 and S_2, the corresponding relations are modified, and the relation representing the result of P_1 is updated to reflect the changes in the inputs. The result of P_1 is updated by *inserting* identifiers of vehicles entering the lot and *deleting* identifiers of vehicles exiting the lot. Notice that although the input relations in P_1 change by only inserting tuples (i.e., are append only), the output of P_1 changes by both insertions and deletions. In order to represent P_1's output as a single stream, we should be able to represent two different types of stream tuples (one type of stream tuples to represent the *insertions* in the output and the other type of stream tuples to represent the *deletions*).

Challenge 2: *Similar interpretation of query inputs and output.* To enable query composition, query inputs and output should be interpreted in the same way so that the output of one query can be used as input to another query. Consider the following query from the same application P_2: *"Group the vehicles inside the parking lot by type (e.g., trucks, cars, or buses). Continuously keep track of the number of vehicles in each group"*. By analyzing the two queries, P_1 and P_2, it is obvious that P_2 is an aggregate query over P_1's output. This observation motivates the idea of defining P_1 as a view, say V_1 and then expressing both P_1 and P_2 in terms of V_1. Notice that the output of P_1 is a stream that represents the changes (i.e., insertions and deletions) in the parking-lot state. P_1's incremental output is interpreted in the same way as inputs, namely, as a stream that represents modifications to an underlying relation. However, P_1's incremental output stream consists of two different types of tuples.

Challenge 3: *Expressing ad hoc queries over data streams.* An ad hoc query is a transient query that, once launched, computes and promptly returns the query result. An ad hoc query will not come alive again until the query is again launched by the user. Ad hoc queries form another important class of queries over data streams. For example, consider the following query from the parking-lot monitoring application, P_3:*"Is Joe Doe's car in the parking lot right now*?". Answering P_3 requires knowing the whole history of S_1 and S_2. However, maintaining the total history of the infinite S_1 and S_2 is impossible. Hence, P_3 can be answered using the view V_1. Basically, a view can be created and maintained for each ad hoc query that is frequently issued, for example, the current time's view, that is, the NOW view, with some update or refresh time granularity (see Challenge 4 that follows).

Challenge 4: *Expressing general refresh conditions (other than time- or tuple-based refresh conditions).* Another important issue in data stream query languages is the frequency by which a query answer gets refreshed as well as the conditions that trigger the refresh. In streaming applications with high tuple arrival rates, an issuer of continuous queries may not be interested in refreshing the answer in response to every tuple arrival. Instead, coarser refresh periods may be desired. For example, instead of reporting the count of vehicles with every change in the parking lot, P_2 may be interested in updating the count of vehicles in each group *every four minutes*. This refresh condition is temporal. However, a powerful language should allow a user to express more general refresh conditions based on time, tuple arrival, events, relation state, etc. For example, P_2 may be interested in updating the count of vehicles in each group whenever a police car enters the parking lot. In this case, the refresh condition is event based where the event is defined as "the entrance of a police car".

Challenge 5: *Expressing queries over streams that do not represent append-only relations.* Streams from different domains may be interpreted differently by different applications [Maier et al. 2005]. For example, one sequence of tuples can represent an infinite append-only relation (e.g., S_1 in P_1). On the other hand, another sequence of tuples may represent an update stream in which an input tuple is an update over the previous tuple with the same key value. For example, consider a temperature-monitoring application in which sensors are distributed in rooms and each sensor continuously reports the room temperature. A reasonable query in this environment is T_1 : *"Continuously keep track of the rooms that have a temperature greater than 80"*. Neither the input nor the output streams in T_1 represent append-only relations. The input in T_1 is an update stream in which a room identifier is considered a key and an input tuple is an update over the previous tuple with the same key value. Notice that although an update stream is also represented as a sequence of tuples, the interpretation of an update stream is different from the interpretation of an append-only stream. The output tuples from T_1 represent an incremental answer that includes insertions and deletions for rooms that switch between satisfying and not satisfying the query predicate.

1.2 Illustrative Example

In this section, we give an example to illustrate that the semantics of query composition, and hence views, is difficult to express by a language that restricts the stream definition to the append-only model. Consider the following query from the same application as that of P_1 in Section 1.1. P_4: "*Continuously keep track of the identifiers of all vehicles inside the parking lot, report changes in the answer every 2 minutes*". In the following, we use CQL [Arasu et al. 2006] as a representative for the class of languages that use the append-only model. CQL uses sliding windows to express the coarser refresh periods where a sliding window is defined by two parameters, namely range and slide. Assume that the schema of the input streams consists of two attributes, VID that represents the vehicle identifier, and VType that represents the vehicle type (i.e, car, bus, or truck). CQL can express P_4 in four different ways as follows.

Case 1. *Relational output*:

```
SELECT R1.VID, R1.VType
FROM S1[range ∞ slide 2] R1 − S2[range ∞ slide 2] R2
```

In this case, the output of P_4 is a relation (not a stream). The output relation gives the complete query answer and is refreshed every 2 minutes. The output is not incremental, which means that every 2 minutes, the query issuer sees all identifiers of vehicles inside the lot.

Case 2. *Streamed relational output*:

```
SELECT RStream(R1.VID, R1.VType)
FROM S1[range ∞ slide 2] R1 − S2[range ∞ slide 2] R2
```

The output in this case is a stream that represents the concatenations of Case 1's output relation. Basically, whenever the output relation is modified (i.e., every 2 minutes), the whole output relation is streamed out (or pushed) to the query issuer. Notice that the output representation is different than Case 1 in which the output relation is stored and the query issuer needs to pull the modified query answer from the stored relation. Notice also that the output stream, say S_o, is interpreted differently from the input streams. An input tuple in any of the input streams (i.e., S_1 or S_2) represents an insertion into the corresponding relations. However, a tuple in S_o may represent a repetition for a previous S_o tuple. For example, vehicles that are inside the lot for more than 2 minutes are reported several times in S_o.

Case 3. *Stream of insertions to the output relation*:

```
SELECT IStream(R1.VID, R1.VType)
FROM S1[range ∞ slide 2] R1 − S2[range ∞ slide 2] R2
```

The IStream (or insert stream) operation produces a tuple in the output stream whenever a tuple is inserted in the output relation (i.e., whenever a vehicles enters the lot). Notice that because of the slide parameter of length 2, the inserted tuples are accumulated and are produced in the output stream every 2 minutes. Although IStreams's output stream is incremental, it gives only a *partial* answer for P_4 because it does not include any information about vehicles exiting the lot.

Case 4. *Stream of deletions from the output relation*:

```
SELECT DStream(R1.VID, R1.VType)
FROM S1[range ∞ slide 2] R1 − S2[range ∞ slide 2] R2
```

The DStream (or delete stream) operation produces a tuple in the output stream whenever a tuple is deleted from the relation (i.e., whenever a vehicles exits the lot). Notice that because of the slide parameter of length 2, the deleted tuples are accumulated and are produced in the output stream every 2 minutes. DStream's output is an incremental but *partial* answer for P_4 because it does not include information about vehicles entering the lot.

Notice that outputs in both Case 1 and Case 2 give nonincremental answers for P_4. On the other hand, for Case 3 and Case 4, the outputs give incremental but partial answers for P_4. However, CQL cannot produce a single stream that represents the whole incremental answer to P_4 that includes both insertions into and deletions from the parking lot state. P_4's output incremental stream should include two different types of tuples to distinguish between the insertions and deletions.

Consider another query P_5 that is similar to Query P_2 however P_5's answer need to be refreshed every 4 minutes. Basically, P_5 is as follows:*"Group the vehicles inside the parking lot by type (e.g., trucks, cars, or buses). Continuously keep track of the number of vehicles in each group, report the changes in the answer every 4 minutes"*. Careful analysis of P_4 and P_5 shows that: (1) P_5 is an aggregate over the output of P_4, and (2) P_5's refresh time points form a subset of P_4's refresh points. As a result, in a powerful language, P_5 should be easily expressed over the output of P_4. However, none of the four CQL's outputs for P_4 (i.e., Cases 1 to 4) can be used as input to express P_5 for the following reasons.

—Case 1's output is a relation (not a stream) and windows (of range and slide) cannot be expressed over relations. As a result, P_5's sliding window (that slides every 4 minutes) cannot be expressed over Case 1's output relation.
—Case 2's output stream is not incremental and does not represent an append-only relation. However, sliding window semantics are defined for streams that represent append-only relations. As a result, P_5's window cannot be expressed over Case 2's output stream.
—Both Case 3's and Case 4's output streams represent *partial* answers for P_4. As a result, expressing P_5 over a Case 3 (or Case 4) output stream does not give the correct answer for P_5.

1.3 Nile-SyncSQL: Supporting Views over Data Streams

This article presents the Nile-SyncSQL prototype server, an engine to support views over data streams. Nile-SyncSQL is based on the Synchronized SQL query language, a *closed* language to express composable queries over data streams. The contributions of this article are as follows.

—We motivate the need for views over streams and discuss challenges that need to be addressed by a query language in order to support views over streams.
—We propose the SyncSQL query language, a *closed* stream query language that enables views over streams. We define concise semantics, syntax, data types, operators, algebra, and transformation rules for SyncSQL. (Sections 4, 5, and 6).
—Based on SyncSQL's algebraic foundation, we propose a query-matching algorithm to deduce containment relationships among SyncSQL expressions. The algorithm is then used to answer queries using views (Section 7).

—We give an analytical cost model to estimate the cost of a given `SyncSQL` execution pipeline. The cost model can be used to choose the best execution plan from a set of possible execution pipelines for a given query (Section 9).

—We design and implement the `Nile-SyncSQL` prototype to support `SyncSQL` queries. We conduct an experimental study to evaluate the performance of `Nile-SyncSQL`. The experimental results are twofold: (1) show the effectiveness of `Nile-SyncSQL` to support continuous queries over data streams; and (2) show significant performance gains when views are enabled in DSMSs (Sections 8 and 10).

2. RELATED WORK

2.1 Continuous Query Semantics and Languages

The unique characteristics of data streams and continuous queries impose new requirements on query languages. Many research efforts have developed semantics and query languages for continuous queries over data streams, for example, Arasu et al. [2006], Bonnet et al. [2001], Abadi et al. [2003], Chandrasekaran et al. [2003], Cranor et al. [2003], and Zaniolo et al. [2002]. The existing continuous query languages define a stream as a representation of an append-only relation. The append-only stream definition limits the set of queries that can produce streams as output. This is because, even if the input streams represent append-only relations, a continuous query may produce nonappend-only output. Different languages follow different approaches in order to handle the nonappend-only output as follows.

Continuous Query Language-CQL [Arasu et al. 2006]. CQL is the query language that is used by the STREAM DSMS. The nonappend-only query output is either: (1) divided into two streams using the IStream and DStream operators, or (2) represents concatenation of time-varying versions of the output using the RStream operator. RStream's output cannot be used as input stream to another continuous query.

Expressive Stream Language (ESL) [Zaniolo et al. 2002]. ESL is used by the `ATLaS` DSMS [Zaniolo et al. 2002]. In order to avoid the nonappend-only output streams, ESL limits the set of operators that can be used to produce output streams to include only unary operators (e.g., selection and projection). Since a window function produces a nonappend-only output, window queries produce concrete views as output. A concrete view is stored and is continuously modified as the input changes. A query issuer, or an ad hoc query pulls the current complete answer from the stored view. Join is defined between streams and concrete views, but the modifications in the view affect only the future join outputs and do not affect the already-produced join output.

GSQL [Cranor et al. 2003]. GSQL is used in the Gigascope stream database that is used for network monitoring. GSQL put some restrictions over SQL to guarantee that a query cannot produce a nonappend-only output.

StreaQuel [Chandrasekaran et al. 2003]. StreaQuel is used in the TelegraphCQ stream database system. A StreaQuel query is expressed in SQL syntax and is followed by a for-loop construct to express windows over input streams. The output of a StreaQuel query is a sequence of time-stamped sets where each set corresponds to the answer of the query at the time that is indicated by the attached timestamp (similar to CQL'S RStream operator).

StreamSQL[http://www.StreamSQL.org]. StreamSQL is a query language that has been developed by computer science and data management experts from various universities in conjunction with StreamBase Systems (http://www.streambase.com). StreamSQL extends SQL by adding new operations in order to manipulate streams. The output stream from a StreamSQL query is append only and does not include delete or update tuples. However, the StreamSQL's language specifications do not address how nonappend-only query output (e.g., output from an aggregate query or a sliding window query) is interpreted for query composition purposes. For example, the output from a sliding window query will not reflect the tuples that expire when the window slides.

2.2 Views in Database Management Systems

Views have been widely used in database management systems. Once defined, the view can be used as input to other queries or views. Views are needed because usually the actual schema of the database is normalized for implementation reasons and the queries are more intuitive using one or more denormalized relations that represent the real world [Gupta and Mumick 1999]. A *materialized view* is a view that is materialized by storing the tuples of the view in the database. Materialized views provide fast access to data since the view is computed once and is stored. Then, any query can use the stored results without recomputing the view. Materialized views have been widely used in query optimization, since answering queries using an existing view yields more efficient query execution plans.

A materialized view becomes out of date when the underlying base relations are modified. Hence, *view maintenance* is the process of updating the view in response to changes in the underlying relations. In most cases, it is wasteful to maintain a view by recomputing it from scratch [Gupta and Mumick 1999]. Thus, it is usually less expensive to compute only changes in the view to update its materialization. Algorithms that compute changes to a view are called *incremental view maintenance* algorithms.

View exploitation is the process of making efficient use of materialized views to speed up query processing [Goldstein and Larson 2001]. Given a query expression, an optimizer uses a view-matching algorithm to see which one of the existing views can be used to rewrite the given expression. The query optimizer then chooses the rewriting that gives the most efficient execution plan. In `Nile-SyncSQL`, we investigate how to apply the various materialized concepts (e.g., incremental maintenance and view matching) over data streams.

2.3 Processing Continuous Queries over Data Streams

The emergence of data streaming applications calls for new query processing techniques to cope with the high rate and the unbounded nature of data streams. A sliding window query is one of the most popular types of queries over append-only streams [Babcock et al. 2002; Golab and Ozsu 2003]. A sliding window query is a continuous query over n input data streams where each input data stream S_j is assigned a window of size w_j. At time T, the current window for stream S_i contains the tuples arriving between times $T - w_i$ and T. Two approaches have been conducted to support sliding window queries, namely, query *reevaluation* [Abadi et al. 2003; Ryvkina et al. 2006] and *incremental evaluation* [Arasu et al. 2006; Ghanem et al. 2007]. In the query reevaluation approach, the query is reevaluated over each window independent from all other windows. Basically, buffers are opened to collect tuples belonging to the various windows. Once all the tuples in the window are received, the completed window buffer is processed to produce the complete window answer [Abadi et al. 2003]. On the other hand, in the incremental evaluation approach, when the window slides, only the changes in the window are processed to produce the answer of the next window. As the window slides, the changes in the window are represented by two sets of inserted and deleted tuples. Incremental operators are used in the pipeline to process both the inserted and deleted tuples and to produce the incremental changes to the query answer [Arasu et al. 2006; Ghanem et al. 2007].

Notice that streams of insert and delete tuples are frequently used when addressing continuous query processing [Ryvkina et al. 2006; Babu et al. 2005; Ghanem et al. 2007]. However, query languages do not consider expressing queries over these modify streams. This conflict between the language and internal streams is the main obstacle in achieving continuous query composition.

2.4 Shared Execution of Continuous Queries

A typical streaming environment has a large number of concurrent overlapping continuous queries. Sharing the query execution is a primary task for query optimizers to address scalability. The current efforts for shared query execution focus on sharing execution at the operator level. For example, shared aggregates are addressed in Arasu and Widom [2004] where an aggregate operator is shared among multiple queries with different window *ranges*. An algorithm for shared execution of window join operators is proposed in Hammad et al. [2003] where the join execution is shared among queries that are similar in the join predicate but with different window clauses. NiagraCQ [Chen et al. 2000] proposes a framework to share the execution among SPJ queries. However, the queries addressed by NiagraCQ use a restricted set of operators and cannot include windows.

In this article, we use views as a means for the shared execution of continuous queries. Sharing the execution through views is distinguished from the existing approaches in that: (1) it does not require the design of new window-aware operators. However, views are supported using differential operators that are general and can support the various types of windows; (2) queries are examined for sharing based on a whole query expression not only at the operator level; and (3) the framework is general and is not restricted to a specific class of queries or operators.

3. SUMMARY OF QUERIES IN THE ARTICLE

This section introduces the queries that we use in the rest of the article to demonstrate the semantics and syntax of SyncSQL. The illustrative queries are drawn from two different applications: a parking-lot monitoring application and a room temperature monitoring application.

3.1 Parking-Lot Monitoring Application

The first set of queries is drawn from the parking-lot monitoring application that is discussed in Section 1. The goal of this application is to show that the output of a continuous query over streams may not be append only, even if the input streams are append only. As discussed in Section 1, there are two sensors that generate two streams of identifiers, say S_1 and S_2, for vehicles entering and exiting the lot, respectively. Both S_1 and S_2 follow the same schema that has three attributes as follows: <VID, VType, VOwner>, where "VID" gives the vehicle's identifier, "VType" gives the vehicle type (e.g., car, bus, or truck), and "VOwner" gives the car's owner name. We use seven example queries over the input streams. The queries P_1, P_2, P_3, P_4, and P_5 are as explained in Section 1. In addition, consider the following queries.

—P_6 is a query that involves an event-based refresh condition, where P_6 is similar to P_2 but needs to be refreshed when a police car enters the parking lot.

—P_7 is a sliding window query as follows:"*Continuously monitor the identifiers of cars that entered the parking lot in the last 5 time units*."

3.2 Room Temperature Monitoring Application

The room temperature monitoring application is an application in which input stream tuples represent modifications to the temperatures of the various rooms. The input stream follows a schema of two attributes as follows: <RoomID, Temperature>, where "RoomID" gives the room identifier that represents the primary key for the input stream. In other words, an input stream tuple is an update over the previous tuples with the same "RoomID" value. The "Temperature" attribute gives the room's current temperature. The goal of the temperature monitoring application is to show that some data stream applications cannot be supported by a query model that assumes append-only semantics. We use five example queries over the input temperature stream. The first query, T_1: "Continuously keep track of the rooms that have a temperature greater than 80". The second query, T_2: "Continuously keep track of the rooms that have a temperature greater than 100". Then, T_3 is similar to T_1 in the query functionality but with different refresh requirements. The same is true for T_2 and T_4. Query T_5 gives an example of an event-based refresh condition since it is similar to T_2 in the functionality but asks to refresh the answer whenever a room reports a temperature greater than 120.

4. STREAM, QUERY, AND VIEW SEMANTICS

4.1 Stream Semantics

A data stream is defined as a sequence of tuples with a specified schema [Arasu et al. 2006; Chandrasekaran et al. 2003; Zaniolo et al. 2002]. The semantics of the stream is application dependent, that is, the different applications may interpret the same stream in different ways [Maier et al. 2005]. For example, one sequence of tuples may represent an infinite append-only relation (e.g., S_1 in the parking lot application as discussed in Section 3.1). On the other hand, another sequence of tuples may represent a concatenation of time-varying states of a fixed size relation (e.g., the RStream operator in CQL [Arasu et al. 2006]). A query language for data streams should first clearly specify the stream semantics, then explain the query operations given the specified stream semantics.

In the streaming literature, query languages model a stream as a representation for an infinite append-only relation [Arasu et al. 2006; Chandrasekaran et al. 2003; Zaniolo et al. 2002]. The append-only stream model effects the following limitations: (1) It limits the applicability of the language since the append-only model cannot represent streams from the various domains (e.g., update streams or streams that represent concatenation of the states of a fixed size relation). (2) The append-only stream model limits the types of queries that the language can express since only nonblocking queries can produce append-only streams as output. (3) The semantics of query composition in the append-only stream model is complex and the meaning of the composed queries is difficult to understand.

To overcome the limitations of the append-only model, we introduce *tagged* stream semantics as a model for representing streams in SyncSQL. Basically, SyncSQL distinguishes between two types of streams: *raw* and *tagged*. A *raw* stream is a sequence of tuples that is sent by remote data sources (e.g., sensors). On the other hand, a *tagged* stream is a stream of modify operations (i.e., insert (+), update(u), and delete(−)) against a relation with a specified schema. A raw stream must be transformed into a tagged stream before being used as input in a query. The raw-to-tagged stream transformation is similar to transforming raw data into tables in traditional databases.

The function that transforms a raw stream to a tagged stream is application dependent. For example, consider P_1 in Section 1. Since the input streams in P_1 (i.e., S_1 and S_2) represent append-only relations, the tagging function for S_1 (or S_2) is to attach a "+" tag to every input tuple. The output of a SyncSQL query over a tagged stream is another tagged stream. For example, the output of P_1 is a tagged stream with "+" and "−" tuples, where a "+" tuple is produced in P_1's output for every vehicle entering the lot and a "−" tuple is produced for every vehicle exiting the lot. P_1's tagged output gives an incremental answer for P_1, and hence, can be used as input to another query (e.g., P_2). The tagged stream model enables SyncSQL to be a powerful and a general-purpose language for the following reasons: (1) query composition is achieved due to the unified interpretation of query inputs and outputs as tagged streams, and (2) a wider class of applications can be supported since the tagged stream model is general and can represent streams from various domains.

Consider two different temperature-monitoring applications, say $\texttt{Application}_1$ and $\texttt{Application}_2$. Assume both application have a raw input stream with the following schema "<RoomID,Temperature>Timestamp". Assume also that $\texttt{Application}_1$ treats the input as an update stream over the various rooms' temperatures (Application_1 is the application that is discussed in Section 3.2). In this case, RoomID is considered a key and a tuple is considered an update over the previous tuple with the same key value. On the other hand, $\texttt{Application}_2$ treats the input stream as a series of temperature readings and the RoomID attribute is ignored. Given that the two streams have the same schema, the job of the tagging function is to tell the query processor that the two streams are interpreted differently.

In the query processing phase, the transformation (or tagging) function is implemented inside an operator, called Tagger. For example, in $\texttt{Application}_1$, the input stream tuples are correlated based on the key (i.e., RoomID), hence the Tagger needs to keep a list of all the observed key values (i.e., RoomID) so far. In $\texttt{Application}_1$, the output from the Tagger operator is a tagged stream, say RoomTempStr, that consists of *insert* and *update* operations. Notice that in $\texttt{Application}_1$, the functionality of the Tagger operator is similar to that of the MERGE (or UPSERT) operator in the SQL:2003 standard [Eisenberg et al. 2004]. On the other hand, in $\texttt{Application}_2$, the Tagger operator does not need to keep any state since tuples are not correlated. In $\texttt{Application}_2$, the output from the Tagger operator is a tagged stream, say TempStr, that consists of a sequence of *insert* operations.

The following is the SyncSQL syntax for defining raw streams.

REGISTER SOURCE $< raw - stream - name >$ $(< schema >)$
FROM $< portnum >$,

where < *raw-stream-name* > is the name of the stream, <*schema*> is the schema of the input stream tuples, and <*portnum*> is the port at which the stream tuples are received. For example, the raw stream TemperatureSource is defined in SyncSQL by the following statement:

REGISTER SOURCE TemperatureSource (*int RoomID*, *int Temperature*)
FROM port5501

The following is the SyncSQL syntax for defining tagged streams over raw streams.

CREATE TAGGED STREAM < *tagged-stream-name* >
OVER <*raw-stream-name*> *KEY* <*attrname*>,

where <*tagged-stream-name*> is the name of the tagged stream and *raw-stream-name*> is the name of the base raw stream. Notice that the raw stream should be defined first before being used in defining a tagged stream. The <*attrname*> is the name of the attribute (or list of attributes) that represents the primary key of the input stream.

The tagging function is very simple in the case of streams that represent append-only relations. The tagging function is more complex in the case of update streams because Tagger needs to keep a state in order to correlate the input tuples. However, the size of the Tagger's state has an upper bound that equals the number of distinct objects. For example, in $\texttt{Application}_1$ the Tagger's state size cannot exceed the maximum number of rooms. Moreover, Tagger does not need to store rooms that do not report temperature updates. Notice that Tagger's state is limited by the domain of the key attribute. As a result, SyncSQL is more efficient for applications with small domain streams. However, some applications may not need Tagger's state if the input stream represents append-only data with no notion of update. On the other hand, in case of applications that require a tagging state, optimizations can be applied in order to minimize the Tagger's state size, as will be discussed in Section 10.

Moreover, implementing the tagging function as an operator opens the room for the query optimizer to reorder the pipeline and optimize memory consumption. The cost of processing one tuple by the Tagger operator can be estimated by running Tagger for a transient period. For example, we can run Tagger for T time units and count the number of tuples that can be processed in T time units. The cost of processing one tuple is estimated from this transient period and then is used by the optimizer to produce the best query execution plan. Moreover, a selection predicate can be pushed into a tagger if there is an agreed-upon interface for doing so. Such an interface can take many forms. For example, the predicate could be passed in as pointer to a function that takes a row (or column values) as inputs and returns true or false. As we show in the experimental evaluation in Section 10, the overhead of the tagging transformation can be minimized by merging the functionality of the Tagger operator with the Select operator. For example, in $\texttt{Application}_1$, the Tagger operator can be merged with the Select operator so that only rooms that qualify the selection predicate are stored in the state. Notice that new applications may require the introduction of new tagging transformations and new tagging syntax. Each new tagging syntax requires the definition and implementation of a new Tagger operator.

The relational view of a tagged stream. In order to adopt the well-known semantics of relational operators, SyncSQL queries are expressed over the tagged streams' corresponding relations. Basically, any tagged stream, say S, has a corresponding time-varying relation, termed $\Re$(S), that is continuously modified by S's tuples. An input tuple in a tagged stream is denoted by "Tag<Attributes>Timestamp", where Tag can be either insert (+), update (u), or delete(−), and Timestamp indicates the time at which the modification takes place. The relational view is modified by the stream tuples as follows: an insert tuple modifies the relation by inserting a new record, an update tuple modifies the relation by changing the attributes of an existing record, while a delete tuple modifies the relation by deleting an existing tuple. $\Re$(S)'s schema consists of two parts as follows: (1) a set of attributes that corresponds to S's Attributes, and (2) a timestamp attribute, termed TS, that corresponds to the Timestamp

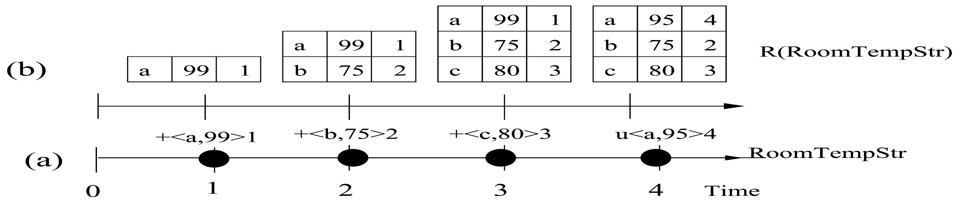


Fig. 1. Illustrating time-varying relations.

field of `S`'s tuples. `Timestamp` is mapped to $\Re$`(S)` in order to be able to express time-based windows over `S`, as will be discussed in Section 4.4. At any time point, say `T`, $\Re$`(S)` is denoted by `R[s(T)]` and is the relation resulting from applying `S`'s operations with timestamps less than or equal to `T` in an increasing order of timestamp.

Example 1. This example demonstrates the mapping from `RoomTempStr` to a time-varying relation. Figure 1(a) gives the following input tuples: "`+<a,99>1, +<b,75>2, +<c,80>3, u<a,95>4`". Figure 1(b) gives $\Re$`(RoomTempStr)` with a schema of three attributes: `RoomID`, `Temperature`, and `TS`. Figure 1(b) illustrates that at time `1`, the tuple for Room "a" is inserted into $\Re$`(RoomTempStr)` with temperature `99`. At time `4`, $\Re$`(RoomTempStr)` reflects the update of Room "a"s temperature to `95`.

4.2 Query Semantics

A continuous query over `n` tagged streams, $S_1 \ldots S_n$, is semantically equivalent to a *materialized view* that is defined by an SQL expression over the time-varying relations, $\Re(S_1) \ldots \Re(S_n)$. Whenever any of the underlying relations is modified by the arrival of a stream tuple, the modify operation is propagated to produce the corresponding set of modify operations in the answer in a way similar to the incremental maintenance of materialized views [Griffin and Libkin 1995]. The output of a query can be provided in two forms as follows: (1) `COMPLETE` output, where, at any time point, the query issuer has access to a table that represents the complete answer of the query. The answer's table is modified whenever any of the input relations is modified. Notice that the output in this case is nonincremental; (2) `STREAMED` output, where the query issuer receives a tagged stream that represents the *deltas* (i.e., incremental changes) to the answer.

Example 2. This example illustrates the syntax of `SyncSQL`. We use the keyword `STREAMED` to indicate that the query asks for an incremental output. The parking lot monitoring query $\mathtt{P}_1$, from Section 3.1, is expressed as follows.

$\mathtt{P}_1$: SELECT STREAMED R1.VID R1.VType
FROM $\Re(\mathtt{S}_1)$ R1 $-$ $\Re(\mathtt{S}_2)$ R2

$\mathtt{P}_1$'s output is a tagged stream that includes a "+" tuple whenever a vehicle enters the parking lot and a "−" tuple whenever a vehicle exits the lot. $\mathtt{P}_1$ gives an example for expressing queries over append-only streams. As another example for expressing queries over update streams, the temperature-monitoring query $\mathtt{T}_1$, given in Section 3.2, is expressed as follows.

```
T₁ :  SELECT STREAMED RoomID, Temperature
      FROM ℜ(RoomTempStr) R
      WHERE R.Temperature > 80
```

4.3 Views over Streams

The unified interpretation of SyncSQL query inputs and outputs enables SyncSQL to exploit *views* over streams. Basically, a view over streams is a function that maps a set of input streams into a derived output stream. Then, a query can reference the derived stream in a way similar to referencing base streams. Notice that the view is defined once and then can be referenced by any other query if the view's expression is contained in the query's expression. In Section 6, we give an algorithm to deduce the containment relationships among SyncSQL expressions.

Example 3. This example demonstrates answering queries using views. As discussed in Section 1, P_2 is an aggregate over P_1's output. Hence, we can define a view, say ParkLot, as follows.

```
CREATE STREAMED VIEW ParkLot AS
SELECT R1.VID, R1.VType
FROM ℜ(S₁) R1 − ℜ(S₂) R2
```

Then, both P_1 and P_2 can be rewritten in terms of ParkLot. For example, P_2 is rewritten as follows.

```
P₂:   SELECT STREAMED P.VType, Count(P.VID)
      FROM ℜ(ParkLot) P
      GROUP BY P.VType
```

4.4 Window Queries

In this section, we demonstrate the ability of SyncSQL to express sliding window queries over append-only streams. A sliding window is defined by two parameters as follows: (1) *range* that specifies window size, and (2) *slide* that specifies the step by which the window moves. In existing query languages, windows are defined using special constructs and may be assigned to streams (e.g., Arasu et al. [2006], Chandrasekaran et al. [2003]) or to operators (e.g., Abadi et al. [2003], Zaniolo et al. [2002]). One limitation of the specific window semantics is that a language that assumes the window-per-stream semantics, for example, cannot express a query with window-per-operator semantics and vice versa.

Unlike other languages, SyncSQL does not assume a specific window assignment. Instead, SyncSQL employs a predicate-window model [Ghanem et al. 2006] in which the window *range* is expressed as a regular predicate in the *where* clause of the query. The window's *slide* is expressed using the synchronization principle as explained in Section 5. The predicate-window model is a generalization of the existing window models, since all types of windows (e.g., window-per-stream and window-per-operator) can be expressed as predicate windows. For example, a window join (i.e., a window-per-operator) between two streams, say S_i and S_j, where two tuples are joined only if they are at most 5 time units apart, can be expressed by the following predicate: $\Re(S_i).\mathrm{TS} - 5 < \Re(S_j).\mathrm{TS} < \Re(S_i).\mathrm{TS}+5$. Similarly, a time-based sliding window over an append-only stream, say S, (i.e., a window-per-stream) is expressed as a predicate over $\Re$(S)'s TS attribute as shown in the following example.

Example 4. Consider the query P_7 as explained in Section 3. P_7 is a sliding window query and is essentially a *view* that, at any time point T, contains the identifiers of vehicles that entered the parking lot between times T - 5 and T. Such a window view is expressed in SyncSQL as follows.

```
CREATE STREAMED VIEW FiveUnitsWindow AS
SELECT *
FROM ℜ(S1) R
WHERE Now − 5 < R.TS ≤ Now
```

The view FiveUnitsWindow is refreshed when either $\Re$(S1) is modified or Now is changed. Notice that although the input stream S is append only, *delete* operations are produced in FiveUnitsWindow's output to represent expired tuples that fall behind the window boundaries.

4.5 Ad Hoc Queries over Data Streams

Consider the ad hoc query P_3 as discussed in Section 1. P_3 can be answered by maintaining a COMPLETE (not STREAMED) view that contains the cars that are currently in the parking lot as follows.

```
CREATE COMPLETE VIEW CompParkLot AS
SELECT R1.VID, R1.VType, R1.VOwner
FROM ℜ(S1) R1 − ℜ(S2) R2
```

Then, P_3 is expressed as follows:

```
SELECT *
FROM CompParkView C
WHERE C.VOwner = 'JOE DOE'
```

5. THE SYNCHRONIZATION PRINCIPLE

If we follow the traditional materialized view semantics, a SyncSQL query answer is refreshed whenever any of the input relations is modified. Unlike materialized views, in streaming applications, modifications may arrive at a higher rate. A continuous query issuer may be interested in having coarser refresh periods for the answer. For example, as we discuss in Section 3.1, P_4's issuer is interested in getting an update of the answer *every two minutes* independent of the rate of change in the parking lot state. The coarser refresh periods are achieved using sliding windows in other query languages and are restricted to be either time or tuple based [Arasu et al. 2006; Chandrasekaran et al. 2003; Li et al. 2005].

In this section, we introduce the synchronization principle as a generalization of sliding windows. The idea of the synchronization principle is to formally specify synchronization time points at which the input stream tuples are processed by the query pipeline. Input tuples that arrive between two consecutive synchronization points are not propagated immediately to produce query outputs. Instead, the tuples are accumulated and are propagated simultaneously at the following synchronization point. The synchronization principle distinguishes SyncSQL by being able to: (1) express queries with arbitrary refresh

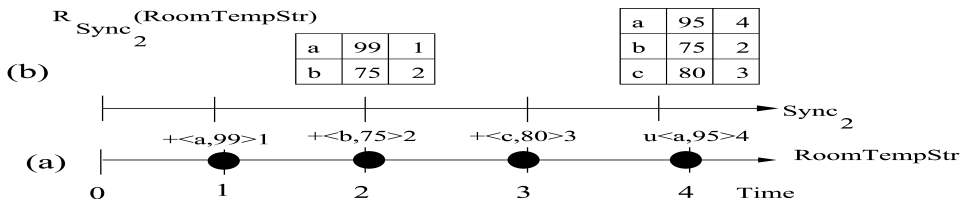


Fig. 2. Illustrating synchronized relations.

conditions, and (2) formally reason about the containment relationships among queries with different refresh periods.

5.1 Synchronized Relations

For each input stream in the query, the query issuer specifies time points at which the input stream tuples need to be reflected in the output. Basically, instead of mapping an input stream, say S, into a time-varying relation, S is mapped to a *synchronized relation*, say $\Re_{Sync}$(S). S's tuples are reflected in $\Re_{Sync}$(S) *only* at those time points that are specified by the synchronization stream, Sync. Notice that $\Re_{Sync}$(S) is of coarser granularity than $\Re$(S).

Example 5. This example illustrates expressing queries with coarser refresh periods. Consider Query T_3 from Section 3.2 that is interested in refreshing the query answer every two time units. To achieve the coarser refresh requirement of T_3, we use the synchronized relation $\Re_{Sync_2}(RoomTempStr)$ as input. The synchronization stream $Sync_2$ is defined as: 0, 2, 4, 6, ... Figure 2 illustrates that $\Re_{Sync_2}$(RoomTempStr) is modified by RoomTempStr tuples every two time units. For example, at Time 1, $\Re_{Sync_2}$(RoomTempStr) is empty and "+<a,99>1" is not inserted in $\Re_{Sync_2}$(RoomTempStr) until Time 2. T_3 is expressed as a view, say $HotRooms_2$, as follows.

```
CREATE STREAMED VIEW HotRooms₂ AS
SELECT RoomID, Temperature
FROM ℜSync₂(RoomTempStr) R
WHERE R.Temperature > 80
```

Notice that $HotRooms_2$ is not refreshed between the synchronization time points. For example, in Figure 2, at Time 3, the contents of the relation $\Re_{Sync_2}$(RoomTempStr) are the same as the contents of the relation at Time 2 and "+<c,80>3" is not inserted in $\Re_{Sync_2}$(RoomTempStr) until Time 4.

5.2 Discussion

The idea of accumulating the tuples of an input stream and propagating them in the query pipeline at once is similar in spirit to the idea of heartbeats [Srivastava and Widom 2004]. In Srivastava and Widom [2004], heartbeats are defined as a special type of tuples that are embedded in the stream such that, at any instant, a heartbeat τ for a set of streams provides a guarantee to the system that all tuples arriving on those streams after that instant will have a timestamp greater than τ. If the stream sources do not provide heartbeats, the DSMS needs to deduce them based on the given stream characteristics.

The context and objectives of heartbeats is totally different than those of synchronization time points. Basically, heartbeats are low-level constructs that are automatically generated by the query processor based on the underlying stream characteristics [Srivastava and Widom 2004]. In other words, the query issuer has no control over the generation of the heartbeats. Basically, changing the heartbeats does not change the semantics nor the output of a given query. On the other hand, synchronization is a high-level concept that is expressed through the query language. Unlike heartbeats, the synchronization principle affects the semantics of a query since the same query has different outputs under different synchronization time points. The synchronization principle and heartbeats are orthogonal, which means that the query processor can use heartbeats in order to generate a correct output for a given `SyncSQL` query.

Punctuation is another mechanism for expressing continuous queries over data streams [Tucker et al. 2003]. A punctuation marks the end of a subset of the data and is used to purge state and to unblock blocking operators. Similar to heartbeats, punctuations are low-level constructs that are not expressed through the query language. However, prior knowledge of the input stream characteristics is utilized in order to generate the appropriate punctuations.

5.3 Synchronization Streams

Before proceeding to the algebraic foundation of `SyncSQL`, this section discusses synchronization streams in more detail. A synchronization stream (e.g., $\texttt{Sync}_2$) specifies a sequence of time points. However, a synchronization stream is represented and is treated as a tagged stream. The tagged representation of a synchronization stream is characterized by the following: (a) The underlying stream schema has only one attribute, termed `TimePoint`, and (b) tuples in the stream are *insert* operations, where a tuple of the form "`+<TimePoint>Timestamp`" indicates a synchronization time of value `TimePoint` where TimePoint = Timestamp. Like any other stream, a synchronization stream `Sync` has a corresponding time-varying relation $\Re$`(Sync)`. The fact that synchronization streams are treated as tagged streams allows `SyncSQL` to compose synchronization streams in order to define a larger class of synchronization streams. The default clock stream, `clockStr: +<0>0, +<1>1, +<2>2, +<3>3, ...`, is the finest granularity synchronization stream. Coarser synchronization streams can be constructed using `SyncSQL` expressions over `clockStr`.

Example 6. The synchronization stream that has a tick at every i time point (e.g., i=2 for $\texttt{Sync}_2$) is constructed from `clockStr` as follows.

```
CREATE STREAMED VIEW Sync_i AS
SELECT C.TimePoint
FROM ℜ(clockStr) C
WHERE C.TimePoint mod i = 0
```

For i=2, a tuple is produced in the output of $\texttt{Sync}_2$ whenever an input tuple, say `c`, is inserted in $\Re$`(clockStr)` and `c.TimePoint` qualifies the predicate "`c.TimePoint mod 2 = 0`". The output of $\texttt{Sync}_2$ is as follows: `+<0>0, +<2>2, +<4>4,...`, which indicates the time points: `0,2,4,...`.

Event-based synchronization. The synchronization principle enables SyncSQL to express queries with event-based refresh conditions. Synchronization streams for event-based conditions can be constructed as in the following example.

Example 7. Consider Query P_6 from Section 3.1 that is to be refreshed only when a police car enters the parking lot. We use the tagged stream S_1 to generate a synchronization stream, say PoliceSync, such that PoliceSync includes time points that correspond to the entrance of a police car into the lot. PoliceSync is constructed as follows.

```
CREATE STREAMED VIEW PoliceSync AS
SELECT R.TS
FROM ℜ(S1) R
WHERE R.VType = POLICE
```

An S_1 tuple, of the form "+<VID,VType>Timestamp", results in producing a tuple of the form "+<Timestamp>Timestamp" in PoliceSync's output if "VType" is POLICE. As discussed in Section 4.1, the attribute R.TS reflects the Timestamp attribute of the input stream tuple which corresponds to the time at which a police car is reported in S_1. Notice that, assuming no delays, a police car is reported in PoliceSync's output at the same time instant at which the car is reported in S_1 (i.e., at time Timestamp).

6. ANSWERING CONTINUOUS QUERIES USING VIEWS OVER STREAMS

In this section, we lay the algebraic foundation for SyncSQL as the basis for efficient query execution.

6.1 Data Types

As discussed in Section 4, although the inputs in SyncSQL expressions are tagged streams, SyncSQL queries are expressed over the input streams' corresponding relations. The output from a SyncSQL expression is another relation that can be mapped into a tagged stream. Basically, a synchronized relation is the main data type over which SyncSQL expressions are expressed. A synchronized relation $\Re_{Sync}(S)$ possesses two logical properties:

—*data* that is represented by the tuples in the relation, where data is extracted from the input stream S; and

—*time* that is represented by the time points at which the relation is modified by the underlying stream S, where time is extracted from the synchronization stream Sync.

A tuple of the form "+<TimePoint>Timepoint" in the synchronization stream indicates a synchronization time with value TimePoint. Time points along the relation lifetime can be classified into two classes in the following way.

—*Full Synchronization Points.* A point in time T is termed a full synchronization time point iff $\Re_{Sync_i}(S_i)$ reflects *all* S_i's tuples up to Time T (i.e., $\Re_{Sync_i}(S_i)$ is up to date with S_i). Basically, the time points $T \in Sync_i$ represent the full synchronization points for $\Re_{Sync_i}(S_i)$.

—*Partial Synchronization Points.* A point in Time T is termed a partial synchronization point if $\Re_{Sync_i}(\mathtt{S}_i)$ does not reflect all $\mathtt{S}_i$ tuples up to Time T (i.e., $\Re_{Sync_i}(\mathtt{S}_i)$ is not up to date with $\mathtt{S}_i$). Basically, the time points that lie between two consecutive $\mathtt{Sync}_i$ represent the partial synchronization points for $\Re_{Sync_i}(\mathtt{S}_i)$.

The distinction between "full" and "partial" synchronization points is essential to judge the relationship between the synchronized relation $\Re_{Sync_i}(\mathtt{S}_i)$ and the underlying stream $\mathtt{S}_i$.

6.2 Operators

In this section, we discuss the logical SyncSQL operators. Logical operators in SyncSQL are classified into three classes: Stream-to-Relation (S2R), Relation-to-Relation (R2R), and Relation-to-Stream (R2S). This operator classification is similar to the classification used by CQL [Arasu et al. 2006], but with different instantiations of the operators in each class.

6.2.1 *The Stream-to-Relation Operator* $\Re$. The same tagged stream can be mapped to different synchronized relations using different synchronization streams. The operator $\Re$ takes a synchronization stream Sync as a parameter and maps an input stream S to a synchronized relation $\Re_{Sync}(\mathtt{S})$. As discussed in Section 4, if an input tuple from S is denoted by "Tag<Attributes>Timestamp", then $\Re_{Sync}(\mathtt{S})$'s schema is as follows: "<Attributes,TS>", where TS corresponds to the Timestamp field of S's tuples. $\Re$ performs the following: (1) buffers S's tuples, (2) modifies the output relation by the buffered tuples at every Sync's point, where the output relation at Sync's Point T is denoted by R[S(T)]. According to the tags of the buffered tuples, $\Re$ can modify $\Re_{Sync}(\mathtt{S})$ by three different operations as follows: (1) an insert "+" tuple causes $\Re$ to insert a new tuple into $\Re_{Sync}(\mathtt{S})$, (2) an update "u" tuple causes $\Re$ to change the values of some attributes of an existing tuple in $\Re_{Sync}(\mathtt{S})$, and (3) a delete "−" tuple causes $\Re$ to delete a tuple from $\Re_{Sync}(\mathtt{S})$. Notice that update and delete operations can be defined only for relations that have a primary key (specified by the create tagged stream KEY clause as explained in Section 4).

6.2.2 *The Relation-to-Stream Operator* ξ. The operator ξ is responsible for producing a STREAMED (or incremental) output of a relation. Any synchronized relation $\Re_{Sync}(\mathtt{S})$ can be transformed into only one tagged stream that represents the modifications to the relation.

Generation of delta tuples. ξ works as follows. At the i^{th} synchronization time point T_i, ξ generates the delta tuples between $\Re[S(T_{i-1})]$ and $\Re[S(T_i)]$ as follows. For every key value k, perform the following: (1) If there is a tuple in $\Re[S(T_{i-1})]$ with key k but there is no tuple in $\Re[S(T_i)]$ with key k, then generate a delete tuple for the key k. (2) If there is no tuple in $\Re[S(T_{i-1})]$ with key k but there is a tuple in $\Re[S(T_i)]$ with key k, then generate an insert tuple for the key k. (3) If there is a tuple with key k in both $\Re[S(T_{i-1})]$ and $\Re[S(T_i)]$ but with different attribute values, then generate an update tuple for the key k.

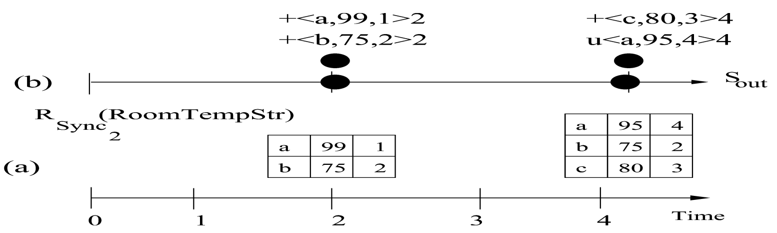


Fig. 3. The relation-to-stream operator.

ξ produces the minimum possible set of tuples that can represent the delta between two states of the relation. For example, one update tuple is produced for each key value k if k has different attribute values between the two consecutive $\Re_{Sync}$(S) states although k may have been modified by a chain of update operations. For example, in the temperature-monitoring application, the same room may report more than one temperature update in the same synchronization period. However, the set of delta tuples that is generated by ξ at the later synchronization point includes only one update tuple per room that represents the latest temperature update. Basically, in this article, we assume that ξ generates the minimum possible set of tuples that can represent the delta between the two states of the relation.

Example 8. Figure 3 gives the mapping from $\Re_{Sync_2}$(RoomTempStr) that is given in Figure 2, to the corresponding stream, S_{out} (i.e., S_{out} = ξ($\Re_{Sync_2}$(RoomTempStr))). For example, at Time 4, ξ produces +<c,80,3>4 and u<a,95,4>4 as the differences which have occurred since the previous synchronization point, 2. Notice that ξ assigns timestamps to the output stream tuples so that the output stream can be used as input in another continuous query.

6.2.3 *Extended R2R Operators.* The R2R class of operators includes extended versions of the traditional relational operators (e.g., σ, π, $\bowtie$, $\cup$, $\cap$, and -). The semantics of R2R operators in SyncSQL are the same as in the traditional relational algebra. The difference in SyncSQL is that an operator is continuously running to reflect the continuous modifications in the input relations. As with materialized views, the output from an R2R operator is refreshed whenever any of the input relations is modified. For a unary operator (e.g., σ, π), the output relation is modified at the input relation's synchronization points. In other words, the synchronization points (full and partial) for the output are the same as those for the input relation. However, for a binary operator, say O, that has two input synchronized relations, $R_{Sync_1}(S_1)$ and $R_{Sync_2}(S_2)$, the input relation $R_{Sync_1}(S_1)$ is modified at every time point in $Sync_1$ while $R_{Sync_2}(S_2)$ is modified at every point in $Sync_2$. As a result, the output of O is modified at every point $T \in (Sync_1 \cup Sync_2)$.

Definition 1 (*Unary Operators*). The output of a unary R2R operator Θ over a synchronized relation $\Re_{Sync}(S)$ is another synchronized relation, denoted by $\Theta(\Re_{Sync}(S))$, such that:
$\forall\, T \in Sync,\ T$ *is a full synchronization point, and*
$\Theta(\Re_{Sync}(S)) = \Theta(R[S(T)])$, while

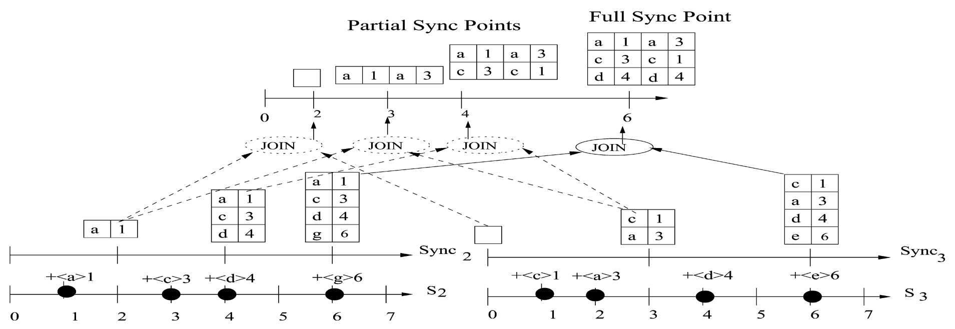


Fig. 4. Joining relations with different synchronization.

$\forall\ T \notin$ *Sync, T is a partial synchronization point, and*
$\Theta(\Re_{Sync}(S)) = \Theta(R[S(\tilde{T})])$
where $\tilde{T} = max\ (t \in$ *Sync and* $t < T)$.

Definition 2 (*Binary Operators*). The output of a binary R2R operator Θ over two synchronized relations $\Re_{Sync_i}(S_i)$ and $\Re_{Sync_j}(S_j)$ is a synchronized relation, denoted by $\Re_{Sync_i}(S_i)\ \Theta\ \Re_{Sync_j}(S_j)$, such that:
(1) $\forall\ T \in Sync_i \bigcap Sync_j$, *T is a full synchronization point, and*
$\Re_{Sync_i}(S_i)\ \Theta\ \Re_{Sync_j}(S_j) = R[S_i(T)]\ \Theta\ R[S_j(T)]$,
(2) $\forall\ T \in (Sync_i - Sync_j)$, *T is a partial synchronization point, and*
$\Re_{Sync_i}(S_i)\ \Theta\ \Re_{Sync_j}(S_j) = R[S_i(T)]\ \Theta\ R[S_j(\tilde{T})]$,
where $\tilde{T} = max(t \in Sync_j$ *and* $t < T)$,
(3) $\forall\ T \in (Sync_j - Sync_i)$, *T is a partial synchronization point, and*
$\Re_{Sync_i}(S_i)\ \Theta\ \Re_{Sync_j}(S_j) = R[S_i(\tilde{T})]\ \Theta\ R[S_j(T)]$,
where $\tilde{T} = max(t \in Sync_i$ *and* $t < T)$
(4) $\forall\ T \notin (Sync_j \cup Sync_i)$, *T is not a synchronization point, and*
$\Re_{Sync_i}(S_i)\ \Theta\ \Re_{Sync_j}(S_j)$ *is not modified.*

Example 9. This example demonstrates a join query between two relations, $\Re_{Sync_2}(S_2)$ and $\Re_{Sync_3}(S_3)$, where $Sync_2$ ticks every 2 units while $Sync_3$ ticks every 3 units. The SyncSQL expression is as follows.

select STREAMED *
from $\Re_{Sync_2}(S_2)\ R_2,\ \Re_{Sync_3}(S_3)\ R_3$
where R_2.ID = R_3.ID

Notice that the join output, say O, is refreshed at time points 2, 3, 4, and 6. Figure 4 illustrates the pipeline. The output at 2 is equal to $R[S_2(2)] \bowtie R[S_3(0)]$ and hence 2 is a partial synchronization point since it reflects S_3 only up to time 0. Similarly, 3 is a partial synchronization point since 3 reflects S_2 up to time 2. Also, 4 is a partial synchronization point since 4 reflects S_3 up to time 3. In contrast, 6 is a full synchronization point for the output since 6 reflects *all* input tuples up to time 6. Notice that in practice it makes more sense to use the same synchronization stream with all the join inputs to indicate the time points at which the query issuer is interested in the query output.

6.3 Equivalences and Relationships

In this section, we introduce preliminary relationships that are required by a query optimizer to enumerate the query plans.

6.3.1 *Containment Relationship among Synchronization Streams.* A synchronization stream, say $\mathtt{Sync}_1$, is contained in another synchronization stream, say $\mathtt{Sync}_2$, if every time point in $\mathtt{Sync}_1$ is also a time point in $\mathtt{Sync}_2$ (i.e., $\Re(\mathtt{Sync}_1) \subseteq \Re(\mathtt{Sync}_2)$). For example, the synchronization stream that is defined over `clockStr` by the predicate "`TimePoint mod 4=0`" is contained in the stream that is defined by the predicate "`TimePoint mod 2=0`".

Proposition 1. $\Re(\mathtt{Sync}_1) \subseteq \Re(\mathtt{Sync}_2)$ if $\forall\ \mathtt{I}\ (\mathtt{I} \in \mathtt{Sync}_1 \Rightarrow \mathtt{I} \in \mathtt{Sync}_2)$, *where* I *is an insert tuple of the form* "+<T>T".

6.3.2 *Containment Relationships among Synchronized Relations.* Reasoning about containment relationships between two synchronized relations must consider the two logical properties, state and time, of the relation. For example, consider two synchronized relations, say $\Re_{Sync_i}(\mathtt{S})$ and $\Re_{Sync_j}(\mathtt{S})$, that are defined over the same stream S. Notice that the *states* of $\Re_{Sync_i}(\mathtt{S})$ and $\Re_{Sync_j}(\mathtt{S})$ may not be equal at every time point if $\mathtt{Sync}_i$ and $\mathtt{Sync}_j$ are not the same. However, if $\mathtt{Sync}_i$ is contained in $\mathtt{Sync}_j$, then $\Re_{Sync_i}(\mathtt{S})$ is *contained* in $\Re_{Sync_j}(\mathtt{S})$. The containment relationship means that every full synchronization time point of $\Re_{Sync_i}(\mathtt{S})$ is also a full synchronization point of $\Re_{Sync_j}(\mathtt{S})$. The containment relationship is important since $\Re_{Sync_i}(\mathtt{S})$ can be computed from $\Re_{Sync_j}(\mathtt{S})$ without accessing S. The containment relationship is judged based only on the full synchronization time points of the relation because these are the time points at which the synchronized relation is completely up to date with the underlying streams.

Theorem 1. *If* $\Re(\mathtt{Sync}_i) \subseteq \Re(\mathtt{Sync}_j)$, *then* $\Re_{Sync_i}(S)$ *can be derived from* $\Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$.

Proof. (1) Based on the functionality of Operator $\Re$, applying $\Re$ with a synchronization stream $Sync_j$ to a stream S maps S's existing tuples into $\Re_{Sync_j}(S)$ without inserting, updating, or deleting any of the existing tuples of S. Then, $\Re_{Sync_j}(S)$ exactly represents $S\ \forall\ \mathrm{T} \in Sync_j$. (2) Similarly, based on the functionality of Operator ξ, applying ξ to a relation $\Re_{sync_j}(S)$ transforms the existing tuples of $\Re_{sync_j}(S)$ into S. Hence, $\xi(\Re_{sync_j}(S))$ exactly represents $\Re_{sync_j}(S)\ \forall$ points in time. (3) From 1 and 2 given before, $\xi(\Re_{sync_j}(S))$ exactly represents $S\ \forall\ \mathrm{T} \in Sync_j$. (4) For a synchronization stream $Sync_i$ such that $\Re(Sync_i) \subseteq \Re(Sync_j)$, then, $\forall\ \mathrm{T} \in Sync_i \Rightarrow \mathrm{T} \in Sync_j$. (5) From 3 and 4 given previously, $\xi(\Re_{sync_j}(S))$ exactly represents $\mathrm{S}\ \forall\ \mathrm{T} \in Sync_i$, hence $\Re_{Sync_i}(S)$ can be derived from $\Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$. □

Theorem 1 means that $\Re_{Sync_i}(\mathtt{S})$ can be derived from $\Re_{Sync_j}(\mathtt{S})$ by applying $\mathtt{Sync}_i$ over the output stream from $\xi(\Re_{Sync_j}(\mathtt{S}))$.

6.3.3 *Commutability between Synchronization and R2R Operators.* R2R operators in a `SyncSQL` expression are executed over synchronized relations. In this section, we show that the order of applying the synchronization and R2R operators can be switched. The commutability between the synchronization and R2R operators allows executing the query pipeline over finest granularity relations and hence allows sharing the execution among queries that have similar R2R operators but with different synchronization.

THEOREM 2. *For any* unary *R2R operator* Θ, $\forall$ `T` *such that* `T` *is a full synchronization point of* $\Theta(\Re_{Sync}(S))$, `T` *is a full synchronization point of* $\Re_{Sync}(\xi(\Theta(\Re(S))))$.

PROOF. (1) From the definition of R2R operators, the full synchronization points of $\Theta(\Re_{Sync}(S))$ are the full synchronization points of $\Re_{Sync}(S)$. In other words, the full synchronization points of $\Theta(\Re_{Sync}(S))$ are the time points that belong to the synchronization stream $Sync$. (2) Since applying the synchronization stream *Sync* is the outermost operation in $\Re_{Sync}(\xi(\Theta(\Re(S))))$, then the full synchronization points of $\Re_{Sync}(\xi(\Theta(\Re(S))))$ are the time points that belong to the synchronization stream *Sync*. (3) From 1 and 2, the full synchronization points of $\Theta(\Re_{Sync}(S))$ and $\Re_{Sync}(\xi(\Theta(\Re(S))))$ are the same and are the time points of *Sync*. □

THEOREM 3. *For any* binary *R2R operator* Θ, $\forall$ `T` *such that* `T` *is a full synchronization point of* $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$, `T` *is a full synchronization point of* $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1)\ \Theta\ \Re(S_2)))$.

PROOF. (1) From the definition of a nonunary R2R operator, the full synchronization points of $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$ are the time points that are full synchronization points for both $\Re_{Sync_1}(S_1)$ and $\Re_{Sync_2}(S_2)$, then, the full synchronization points of $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$ are the time points that belong to $Sync_1 \bigcap Sync_2$. (2) Since applying the synchronization stream $Sync_1 \bigcap Sync_2$ is the outermost operation in $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1)\ \Theta\ \Re(S_2)))$, then the full synchronization time points of $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1)\ \Theta\ \Re(S_2)))$ are the time points that belong to $Sync_1 \bigcap Sync_2$. (3) From 1 and 2, the full synchronization points of both $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$ and $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1)\ \Theta\ \Re(S_2)))$ are the same and equal the time points that belong to the synchronization stream $Sync_1 \bigcap Sync_2$. □

The main idea of Theorems 2 and 3 is that we can pull the synchronization streams out of an R2R operator. Basically, an R2R operator can be executed over the finest granularity relations and produce the finest granularity output. Then, the desired synchronization is applied over the fine granularity output.

7. SYNCSQL QUERY MATCHING

In this section, we introduce a query-matching algorithm for `SyncSQL` expressions. The goal of the algorithm is that, given a `SyncSQL` query, say Q_i, the algorithm determines whether Q_i (or a part of it) is contained in another view, say Q_j. If such Q_j exists, the algorithm rewrites Q_i in terms of Q_j in a way similar to answering queries using views in traditional databases.

7.1 Peeling SyncSQL Expressions

To reason about the containment of SyncSQL expressions, we isolate the synchronization streams out of the expression's data. The containment relationship is then tested in two separate steps: one step to test data containment, and another step to test synchronization containment. We term the resulting form of the expression a *peeled* form.

Definition 3 (*Peeled* SyncSQL *Expression*). The peeled form of a SyncSQL expression is a derived synchronized relation that is defined with: (a) *state*, which is a SQL expression over the finest granularity relations, and (b) *time*, which is a global synchronization stream that specifies the full synchronization points of the expression.

Theorems 2 and 3 are used to transform any SyncSQL expression into the corresponding peeled form. Notice that we can match two expressions only at the full synchronization points because they are the points at which the query answer is up to date with *all* the input streams.

Example 10. This example derives the peeled form for the SyncSQL expression Q = $\sigma(\Re_{Sync_1}(S_1) \bowtie \Re_{Sync_2}(S_2))$. The derivation is performed in two steps as follows.

—Using Theorem 3, pull the synchronization streams out of the join operator.
Q = $\sigma(\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1) \bowtie \Re(S_2))))$.

—Using Theorem 2, pull the synchronization stream out of the selection operator.
Q = $\Re_{Sync_1 \bigcap Sync_2}(\xi(\sigma(\Re(S_1) \bowtie \Re(S_2))))$.

The constructed peeled form indicates that Q is equivalent to a synchronized relation with the following: (1) *data:* $\sigma(\Re(S_1) \bowtie \Re(S_2))$, and (2) *full synchronization time points:* $\mathtt{Sync}_1 \bigcap \mathtt{Sync}_2$ that gives the full synchronization points for the expression.

7.2 Query-Matching Algorithm

SyncSQL query matching is similar to view exploitation in materialized views [Goldstein and Larson 2001; Larson and Yang 1985]. However, a matching algorithm for SyncSQL expressions matches the two parts of the peeled forms: state and time. In the following, we give the high-level steps of the SyncSQL query matching algorithm. The input to the algorithm is a SyncSQL query expression, say Q, and a set of peeled forms for the concurrent views.

Example 11. This example illustrates the matching of the temperature monitoring query $\mathtt{T}_4$ with the view $\mathtt{HotRooms}_2$ that is created in Example 5. Assume that the input expressions are as follows.
$\mathtt{HotRooms}_2 = \sigma_{Temp\,>\,80}(\Re_{Sync_2}(RoomTempStr))$
$\mathtt{T}_4 = \sigma_{Temp\,>\,100}(\Re_{Sync_4}(RoomTempStr))$
The corresponding peeled forms for the two expressions are as follows.
$\mathtt{HotRooms}_2 = \Re_{Sync_2}(\xi(\sigma_{Temp\,>\,80}(\Re(RoomTempStr))))$
$\mathtt{T}_4 = \Re_{Sync_4}(\xi(\sigma_{Temp\,>\,100}(\Re(RoomTempStr))))$

Algorithm. SyncSQL-Expression-Matching.

(1) Using Theorems 2 and 3, transform Q to a peeled form by constructing the two components: (1) Q's data, Q^d, and (2) Q's synchronization, $Sync_Q$;
(2) Match Q^d with data parts of the other input peeled forms using a view-matching algorithm from the materialized view literature (e.g., Goldstein and Larson [2001]). The result of the matching is a peeled form (if any) for a matching expression, say $\tilde{Q}$, such that $\tilde{Q}$ consists of a data part $\tilde{Q}^d$ with synchronization stream $Sync_{\tilde{Q}}$.
(3) If such a $\tilde{Q}$ exists, use Proposition 1 to check the containment relationship between the synchronization streams $Sync_Q$ and $Sync_{\tilde{Q}}$;
(4) If $\Re(Sync_Q) \subseteq \Re(Sync_{\tilde{Q}})$, then Query Q can be rewritten in terms of $\tilde{Q}$ as follows. First, rewrite Q^d in terms of $\tilde{Q}^d$ using the same algorithm used in step 2 given before. In other words, find the function F such that $Q^d = F(\tilde{Q}^d)$.
(5) Apply Q's synchronization $Sync_Q$ to the result of the rewrite in order to get the desired Q's output. In other words, we have $Q = \Re_{Sync_Q}(\xi(F(\tilde{Q})))$.

By comparing the two peeled forms we can conclude that: (1) $\Re(Sync_4) \subset \Re(Sync_2)$, and (2) using a view matching algorithm (e.g., [Goldstein and Larson 2001]) shows that the "Temp > 100" ⇒ "Temp > 80". Then, the algorithm concludes that $T_4 \subset HotRooms_2$. Then, the data part of T_4 can be re-written in terms of $HotRooms_2$ as follows:
$T_4 = \sigma_{Temp > 100}(\xi(\Re(HotRooms_2)))$.
Then, T_4's synchronization is applied to the output of the re-write as follows:
$T_4 = \Re_{sync_4}(\xi(\sigma_{Temp > 100}(\xi(\Re(HotRooms_2)))))$

8. THE NILE-SYNCSQL PROTOTYPE

In this section, we present the design of Nile-SyncSQL, a prototype server to support SyncSQL queries. Nile-SyncSQL uses a pipelined queuing model for the evaluation of continuous SyncSQL queries. Query operators in the pipeline are connected via first-in-first-out queues. An operator, say *p*, is scheduled once there is at least one input tuple in *p*'s input queue. Upon scheduling, *p* processes its input and produces output tuples in *p*'s output queue, which is the input queue for the next operator in the pipeline.

The physical implementation of SyncSQL pipelines follows an incremental evaluation approach in order to avoid the reexecution of the pipeline with every input stream tuple. In the incremental evaluation approach, only modifications in the input relations are processed by the query pipeline in order to produce a corresponding set of modifications in the output. Basically, an incremental query pipeline is constructed using differential operators instead of the relational operators. Each R2R operator (e.g., σ and $\bowtie$) has a corresponding incremental (or differential) operator (e.g., σ^d and $\bowtie^d$). We can say that the physical SyncSQL operators are incremental operators that form a class of Stream-to-Stream (S2S) operators. Some of the incremental operators need to keep an internal state to be used to process the input modifications and produce the corresponding modifications in the output. In effect, the functionality of an S2S operator combines three functions as follows: (1) takes an input modification tuple (i.e., +, u, or -) and applies the modification to the operator's internal state (if any), (2) performs the relational operator's function over the internal state, then (3) reports the output modifications as an output tagged stream. SyncSQL's differential operators use the same semantics of differential operators that are used in the incremental maintenance of materialized views [Griffin and Libkin 1995]. Two equations are given for every operator. One equation gives the semantics when the input changes by *inserting* a tuple and the other equation gives the semantics when the input changes by *deleting* a tuple. There are no specific equations for the semantics when the input changes by *updating* a tuple, since the "update" semantics can be derived as the composition of two operations: "deletion of the old values" and "insertion of the new values".

Tuples in the pipeline are Tagged tuples and can be either insertion (+), update (u), or deletion (-) tuples. The tagged tuple's attributes follows the stream's defined schema. An update tuple has an additional part to hold the old attribute values. The old attribute values are first attached by the Tagger operator that is the first operator to produce update tuples in the pipeline. As the update tuples propagate in the pipeline, the old attributes are processed by the various operators. If an operator is to produce an update tuple as output, the operator is responsible for attaching the old attributes to the output tuple according to the operator's semantics. An operator gets the old attributes either from the input tuple's old attributes or from the operator's stored state. The old values are needed by the various operators in the pipeline in order to maintain a correct query answer.

In addition to the incremental operators, two new operators are needed to implement the tagging and synchronization principles. The tagging principle is implemented via a Tagger operator. A Tagger operator is needed to transform the input raw streams into tagged streams. Notice that the tagging function is application dependent and different Tagger operators may need to be implemented. On the other hand, the synchronization principle is implemented via the Synchronizer operator. A synchronizer operator is needed if the query has coarser refresh requirements. Synchronizer is a buffering operator that buffers the input stream tuples and only releases them to the query pipeline at specified synchronization points. For tagged streams, the Synchronizer operator performs summarization on the input tuples. For example, if an object, say O, is inserted then deleted in the same synchronization period, then O is not of interest to the query issuer and hence the processing of O's tuples can be avoided. Hence, the Synchronizer operator digests both O's insert and delete tuples and does not produce them in the output. Moreover, if another object receives two updates in the same synchronization period, then the processing of the earlier update can be avoided since it is not of interest to the query issuer. Such summarizations reduce the number of tuples in the pipeline without affecting the correctness of the answer.

Operators in Nile-SyncSQL are push based. The push-based nature of continuous operators helps avoid deadlocks in case of shared execution of continuous queries. A traditional multiquery optimization pipeline may encounter a deadlock because the operators depend on the pull-based approach [Dalvi et al.

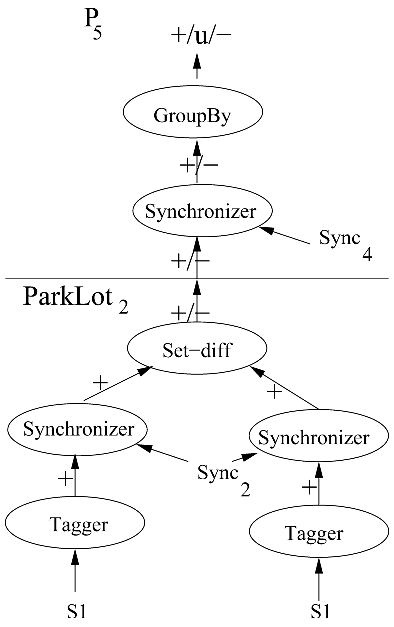


Fig. 5. Example SyncSQL query pipeline.

2001]. Basically, a deadlock happens if an operator does not pull tuples from the shared buffers. Hence, the shared buffers remain full and the shared pipeline cannot produce more tuples in the shared buffers. However, in the case of continuous queries, the shared pipeline pushes the output tuples to all queries. Each query pipeline has input queues to hold the tuples that are pushed to the query. Operators always read tuples from the input queues and store the tuples in the operator's private state (if needed). For example, Join continuously reads tuples from the Join's input queues. If one of the Join's input queues is empty, Join keeps on reading tuples from the other input queue, hence, the input queues will not be full. A problem might appear if Join's internal state is full. However, this problem is addressed by using windows and expiration. For example, a tuple is deleted from Join's state when the corresponding negative tuple is received.

Figure 5 gives the pipeline for the parking-lot monitoring view $\texttt{ParkLot}_2$ and the subsequent query $\texttt{P}_5$. Figure 5 illustrates that $\texttt{ParkLot}_2$'s pipeline consists of the following operators: (1) A Tagger operator is attached with each one of the input streams. Tagger's output is a stream of "+" tuples since the input streams (i.e., $\texttt{S}_1$ and $\texttt{S}_2$) represent append-only relations; (2) A Synchronizer operator is placed on top of each Tagger operator. The Synchronizer's job is to buffer the input tagged tuples and to produce them in the output every 2 time units when a synchronization point is received from $\texttt{Sync}_2$; (3) A Set-difference operator that processes the input "+" tuples and produces a tagged stream as output. The Set-difference's output stream represents $\texttt{ParkLot}_2$'s output that includes "+" tuples for vehicles entering the parking lot and "−" tuples for vehicles exiting the parking lot. Query P_4 is expressed in terms of the `ParkLot`$_2$ view. As a result, `ParkLot`$_2$'s output is used as input in `P`$_5$'s pipeline which consists of two operators, a Synchronizer and a Group-by. `P`$_5$'s output stream is a tagged stream that includes a "+" tuple for each new group, a "u" tuple for a group whenever the number of vehicles in the group changes, and a "−" tuple whenever a group needs to be deleted because all vehicles in that group exit the lot.

8.1 Discussion

Some data streams have high arrival rates. Load-shedding techniques are proposed to discard some fraction of the unprocessed data when the DSMS cannot process the tuples as fast as they arrive (e.g., see Abadi et al. [2003]). The effect of load shedding is that the produced query answer is just an approximation of the accurate answer. An example of a load-shedding technique in `Nile-SyncSQL` is to limit the number of updates per object per unit of time. When the system is overloaded, the load shedder chooses tuples to drop from objects that have the largest number of updates. Similar to the load-shedding techniques, applying load shedding in `Nile-SyncSQL` will result in the production of approximate query answers. If load shedding is to be used, the incremental operators should be furnished with some new rules to maintain the correctness of the operators' state. Basically, the following rules need to be considered: (1) When an update tuple is dropped from the pipeline, this update will not be reflected in the query answer. (2) When Insert tuples are to be dropped, then operators should accept update tuples for nonexisting keys. Basically, an update tuple for a nonexisting key is treated as an insert of the new attributes while the old attributes are ignored. (3) Delete tuples cannot be dropped. Hence, when a tuple is deleted without a corresponding insert (if the insert was dropped), then that delete tuple is ignored. This previous item is just a sample load-shedding policy. Other more sophisticated policies can be explored. However, this issue is beyond the scope of this article.

9. COST ANALYSIS OF `SYNCSQL` QUERY PIPELINES

In this section, we present a cost model to be adopted by the query optimizer to estimate the cost of a given `SyncSQL` execution pipeline. The task of a query optimizer is to find the best execution plan for a given query or a given set of queries. Usually, this goal is accomplished by examining a large space of possible execution plans and comparing these plans according to their "estimated" execution cost. The cost model takes several inputs such as the input arrival pattern, the estimated input size, and the estimated selectivity of the individual operations.

Traditional database management systems use selectivity information to estimate the cost of a given execution plan up to completion. However, this cost metric does not apply to continuous queries, where the time to complete the query is infinite [Kang et al. 2003]. Hence, the cost model presented in this section finds the cost of executing a given pipeline for a specified period of time. The CPU cost of executing a given plan depends on the following: (1) the number and the organization of operators in the pipeline, (2) the number of tuples processed by each operator, and (3) the CPU cost of processing one tuple in each operator. Basically, the CPU cost of executing a pipeline that consists of `n` operators for t time units can be estimated as follows.

$$C_{pipeline}(t) = \sum_{i=1}^{n} C_{O_i}(t)$$

where $C_{O_i}(t)$ is the CPU cost of running operator $\mathtt{O}_i$ for `t` time units. $C_{O_i}(t)$ can then be estimated as follows.

$$C_{O_i}(t) = T_i^{in}(t) \ * \ c_i$$

where $T_i^{in}(t)$ is the number of input tuples that arrive to $\mathtt{O}_i$ during the execution period of t time units and c_i is the CPU cost of processing one tuple in $\mathtt{O}_i$. Notice that c_i is an input parameter that depends on both the system parameters and the implementation. Let $T_i^{out}(t)$ be the number of output tuples from O_i during the execution period. Then $T_i^{in}(t) = T_{i-1}^{out}(t)$. Notice that $T_1^{in}(t)$ is also an input parameter that gives the estimated number of input tuples during t time units. If the bottommost operator is a nonunary operator, then $T_1^{in}(t)$ is the summation of all the input tuples from all the input streams. Notice also that the output cardinality of an operator depends on the number of input tuples (i.e., T^{in}) and on the operator functionality. The reader is referred to the Electronic Appendix that can be accessed through the ACM Digital Liberary for a complete analysis of the relationship between T^{in} and T^{out} for the various types of operators.

10. EXPERIMENTAL EVALUATION OF `NILE-SYNCSQL`

In this section, we give an experimental evaluation of the `Nile-SyncSQL` prototype. The goal of the experimental evaluation is to (1) analyze the factors that affect the performance of `SyncSQL` queries, and (2) demonstrate the effectiveness of supporting views in DSMSs.

10.1 Experimental Setup

The `Nile-SyncSQL` prototype is implemented on Intel Pentium 4 CPU 2.4 GHz with 512MB RAM running Windows XP. A continuous query is evaluated via a pipeline of operators where each operator in the pipeline runs as an independent thread. The threads communicate with each others via FIFO queues. A producer-consumer locking mechanism is implemented to control the queue access in a way that a queue is accessed by at most one thread at a time. Operators' threads are scheduled using a round-robin scheduling.

10.1.1 *Workload Queries.* We use queries from the temperature-monitoring application (that is discussed in Section 4) to evaluate the performance of `Nile-SyncSQL`. The temperature-monitoring application facilitates the study tagging performance since a tagging function is defined to transform the input streams into tagged streams by correlating the input tuples based on the key attribute, `RoomID`. Hence, the temperature-monitoring application facilitates testing the performance while three different types of tuples (i.e., insert, update, and delete) flow in the query pipeline. Two input streams are generated, namely `TemperatureSource` and `HumiditySource`. `TemperatureSource` is a stream that reports the various rooms' temperature and has a schema of three attributes as follows: (`RoomID, Building, Temperature`), where `RoomID` is an integer attribute that gives the room identifier, `Building` is an integer attribute that represents the building in which the room resides, and `Temperature` is an integer attribute that gives the temperature reading of the given room. Similarly, `Humiditysource` is a stream that reports the various rooms' humidity and has a schema of three attributes as follows: (`RoomID, Building, Humidity`). The `RoomID` is the key attribute for both the `TemperatureSource` and `HumiditySource` streams, and an input stream tuple is an update over the previous tuple with the same `RoomID` value. A tagging transformation is defined to transform `TemperatureSource` and `HumditiySource` streams into the tagged streams `RoomTempStr` and `RoomHumStr`, respectively. We use the following operators to construct various query pipelines: Tagger, Synchronizer, Select, Project, Join, Group-by, and Aggregate.

10.1.2 *Data Generation.* We use randomly generated synthetic data in our experiments. To generate the `TemperatureSource` stream, we specify the number of distinct identifiers (i.e., number of rooms) and the number of buildings, where the rooms are evenly distributed among buildings. Then, we specify the input stream's arrival rate which is defined as the number of stream tuples to be received in one second. The interarrival time between two data items follows the exponential distribution with mean λ tuples/second. The arrival rate of the input streams is changed by varying the parameter λ. We generate the stream tuples such that the arrival rate is evenly distributed among the rooms (if not mentioned otherwise). For example, in a stream that reports readings from 200 rooms with an arrival rate of 20000 tuples/second, each room reports its temperature 100 times/second. The temperature readings are varied from 73 to 100.

10.2 NILE-SYNCSQL vs. Traditional Window Processing Techniques

It is important to note that processing different types of tuples is addressed when discussing continuous query processing [Ryvkina et al. 2006; Babu et al. 2005; Ghanem et al. 2007], however, SyncSQL is the first language to address the different types of tuples from the query language (not query processor) point of view. Comparing the performance of incremental evaluation of sliding window queries with the traditional approach is addressed in Ghanem et al. [2007]. The conclusion from Ghanem et al. [2007] is that the straightforward incremental evaluation of sliding window queries (similar to what we have in Nile-SyncSQL) has some advantages and some disadvantages. The advantage is providing accurate query answer independent from the input stream characteristics. However, the disadvantage is the overhead of processing different types of tuples. However, Ghanem et al. [2007] shows that the incremental evaluation opens a room for optimizing query performance. Then, Ghanem et al. [2007] illustrates that performance of incremental evaluation along with optimizations is better than the traditional sliding window evaluation approaches. In the article at hand, we focus on evaluating the performance of `Nile-SyncSQL` as an incremental evaluation approach. However, the performance of `Nile-SyncSQL` can be further enhanced by applying similar optimizations as the ones proposed

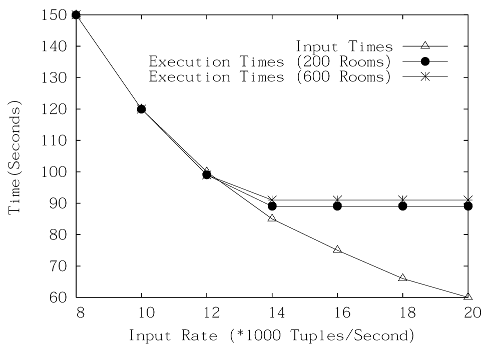


Fig. 6. Effect of arrival rate.

in Ghanem et al. [2007]. It would be repetitious to repeat these here. The reader is referred to Ghanem et al. [2007] for further detail.

10.3 Performance of the Tagger Operator

In this section, we analyze the factors that affect the performance of the Tagger operator and propose optimizations to minimize the overhead of tagging. We first run an experiment to measure the Tagger's throughput, where the throughput is defined as the maximum number of tuples that can be processed by the Tagger operator per time unit. Notice that the Tagger's throughput depends on the complexity of the tagging transformation. We run a query pipeline that consists of only a Tagger operator where `TemperatureSource` is used as input. We run the experiment several times while varying the number of distinct room identifiers in `TemperatureSource`. The pipeline works as follows: Tagger reads a tuple from `TemperatureSource`, uses the tuple to maintain the state, attaches the corresponding tag, and produces the tagged tuple in the output.

Figure 6 gives the effect of the input arrival rate on the query execution time. We measure the time taken by the pipeline to process 1.2 million input tuples while varying the arrival rate from 6000 to 20000 tuples/second. The graphs in Figure 6 give the input and execution times. The "Input Times" graph illustrates that for the same number of input tuples, the input time decreases as the arrival rate increases. However, the "Execution Times" graphs illustrate that the execution time initially decreases with the increase in the arrival rate, then saturates when the arrival rates reaches 14000 tuples/second. Two "Execution Times" graphs are given to illustrate the execution time when the updates in `TemperatureSource` are sent by 200 and 600 rooms. Before saturation (i.e., for arrival rates less than 14000 tuples/second) the execution time is the same as the input time, which means that the system is not overloaded and that the input tuples are processed as fast as they arrive. At saturation, the execution time is fixed at 90 seconds even if the arrival rate is larger than 14000 tuples/second. The conclusion is that the maximum throughput of the Tagger operator is around 14000 tuples/second. The graphs in Figure 6 also illustrate that the Tagger's throughput is almost the same when the number of rooms is

200 or 600. The throughput is independent of the number of distinct key values because both streams have the same number of input tuples, and each input tuple takes the same amount of time to be processed independent of how many tuples are processed for the same room identifier.

10.3.1 *Merged Select-Tagger Operator*. In the temperature-monitoring application, the Tagger operator maintains information for all the distinct room identifiers in order to correlate the input tuples. At the same time, a query may be interested in only a small number of rooms (e.g., by having a selection predicate on the `RoomID` attribute). As a result, the overhead of the Tagger operator can be reduced if the Tagger is aware of the query's selection predicate. In order to minimize the tagging overhead, we propose to merge the tagging functionality with the Select operator. The merged `Select-Tagger` operator receives the raw input stream tuples, evaluates the selection predicate, and assigns appropriate tags to the output tuples. The merged `Select-Tagger` operator stores only rooms that qualify the selection predicate. Moreover, a room is deleted from the state once the room reports a temperature update that disqualifies the selection predicate.

We use the $\texttt{HotRooms}_2$ view from Example 5 (in Section 4) to evaluate the performance of the merged `Select-Tagger` operator. The straightforward $\texttt{HotRooms}_1$'s pipeline consists of two operators: Tagger and Select. `TemperatureSource` is the input stream to $\texttt{HotRooms}_1$'s pipeline and the output is a stream that represents the rooms with temperature > 80. Tagger maintains a state that contains one entry for each distinct room identifier and produces insert and update tuples for the various rooms. The output from the Tagger operator is then used as input to Select. When processing an update tuple, Select applies the selection and projection predicates twice, once on the old values and once on the new values. If we apply the merged `Select-Tagger` optimization, the optimized $\texttt{HotRooms}_1$'s pipeline will consist of one operator, namely the merged `Select-Tagger` operator. The merged operator improves both the memory and CPU consumption as follows.

—*Memory.* Only rooms that qualify the selection predicate are stored in the state. Memory savings can be considerable when the query employs highly selective predicates.

—*CPU.* The merged `Select-Tagger` operator reduces the CPU cost of the query pipeline due to the following: (1) avoids updating the state by tuples which correspond to rooms that do not qualify the selection predicate, and (2) avoids the reexecution of select and project predicates on the old part of an update tuple by getting the old part processing result from the stored tagging state.

Figure 7 gives the execution times that are taken to process different input sizes to $\texttt{HotRooms}_1$'s view. The number of rooms is set to 200 and the arrival rate is fixed to 20000 tuples/second, while the input size is varied from 400000 to 1.2 million tuples. The three graphs in Figure 7 compare three cases: (1) a pipeline of two operators, namely Tagger and Select, (2) a pipeline with one operator, namely the merged `Select-Tagger` operator, and (3) a pipeline with a `Select`

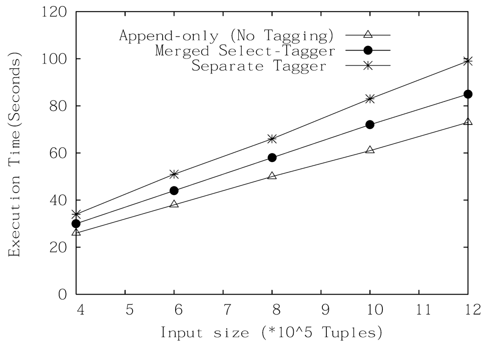
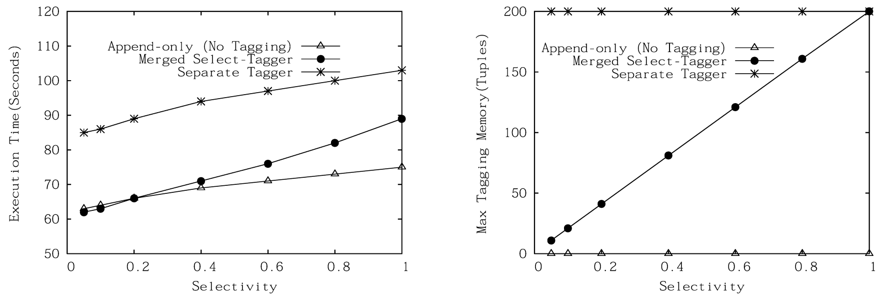


Fig. 7. Cost of the tagging operation.

operator only. The pipeline in Case (3) does not give the desired query semantics, since the input tuples are not correlated based on the `RoomID` attribute. However, we include this case to quantify the tagging overhead. Figure 7 gives the throughput of the three different pipelines as follows: (1) 12K tuples/second (2) 14K tuples/second, and (3) 16K tuples/second. These throughput values indicate that the merged `Select-Tagger` results in a 15% increase in throughput compared to the separate Tagger. The increase in throughput is due to the reduction in the numbers of state modifications, selection, and projection evaluations. Moreover, Figure 7 illustrates that the tagging overhead reduces the throughput by 10% in contrast to the no-tagging pipeline (i.e., Pipeline 3). The conclusion is that although processing update tuples doubles the number of selection and projection evaluations, it does not double the execution time because it does not double the communication cost nor the cost of constructing the output tuples.

10.3.2 *Effect of Selectivity.* In this section, we study the effect of selectivity on the performance of the merged `Select-Tagger` operator. We divide this study into two sections as follows: key selectivity and nonkey selectivity. In Key-selectivity queries, the selection predicate is defined on the key attribute of the input stream (e.g., RoomID). In contrast, in nonkey selectivity queries, the selection predicate is on a nonkey attribute (e.g., Temperature). In the case of key selectivity, once an object (i.e., room) qualifies the predicate, the object continues to qualify the predicate for as long as the query is running. As a result, once a qualified object is inserted in the `Select-Tagger`'s state, the object will not be deleted and the size of the Tagger's state will be fixed during the query runtime. On the other hand, for nonkey selectivity, an object may fluctuate between qualifying and disqualifying the query predicate. As a result, the size of the `Select-Tagger`'s state will vary during the query runtime.

Effect of key selectivity. Figure 8 gives the effect of key selectivity on the tagging cost. This experiment is performed for a pipeline that is similar to that of $\texttt{HotRooms}_1$'s view pipeline while changing the selection predicate. The graphs in Figure 8 compare the performance of the same three pipelines in Figure 7. Figure 8(a) gives the effect of selectivity on execution time, while Figure 8(b)



(a) execution time (b) memory

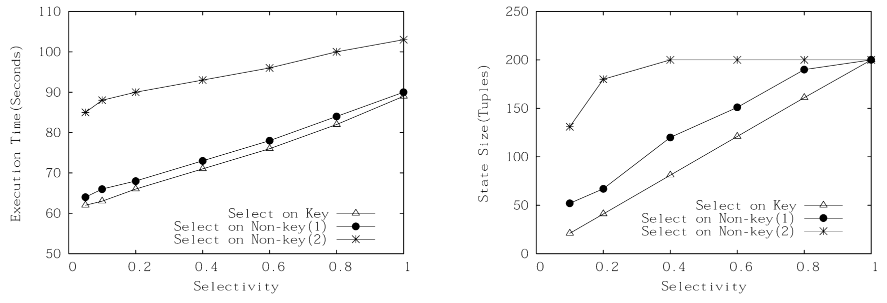
Fig. 8. Effect of key selectivity on Tagger's performance.

gives the effect of selectivity on memory consumption. The input size in this experiment is 1.2 million tuples. The selection predicate is on `RoomID` attribute and selectivity is varied from 0 to 1. Figure 8(a) illustrates that the merged `Select-Tagger` operator achieves a 30% improvement in the execution time if compared to the separate Tagger. The reason for this improvement is that the separate Tagger performs many unneeded state maintenance operations, since all room identifiers are stored and used to update the Tagger's state. Figure 8(a) also illustrates that for low selectivity values (less than 0.5) the tagging overhead is almost zero and the merged `Select-Tagger` pipeline has the same execution time as the No-Tagging pipeline. The tagging overhead started to appear from selectivity values larger than 0.5 when the merged operator performs slightly worse than the append-only performance because of the state maintenance.

Figure 8(b) gives the effect of the selectivity on the memory. For the separate Tagger pipeline, the memory requirement is independent from the selectivity and equals to the maximum number of rooms because Tagger stores all rooms, even the rooms that do not qualify the query predicate. For the merged `Select-Tagger` operator, the state size is proportional to the selectivity because only rooms that qualify the selection predicate are stored. In other words, the merged `Select-Tagger` operator has the minimum possible memory requirement for the correct query evaluation. For the append-only stream semantics, no tagging is needed, and hence the memory requirements equal zero. However, in this case, the output stream does not convey the required semantics.

Effect of nonkey selectivity. In this section, we illustrate the difference between key and nonkey selectivity. We use a pipeline that consists of one merged `Select-Tagger` operator. Notice that the number of output tuples may exceed the selectivity factor if the selection predicate is on a nonkey attribute. The extra, basically negative, tuples are produced because some rooms may be deleted from the output several times depending on the object update pattern. The number of output tuples gives an indication to the query execution time.

Figure 9 compares the performance of the `Select-Tagger` pipeline in the case of the key and nonkey selectivities. Moreover, Figure 9 illustrates the effect of



(a) execution time (b) memory

Fig. 9. Effect of nonkey selectivity on Tagger's performance.

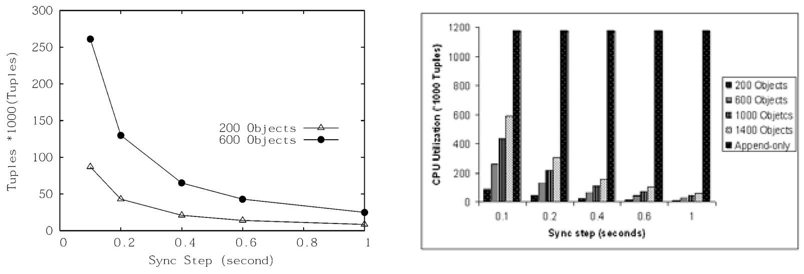
the input data distribution on the performance. We run the same query with a nonkey selectivity predicate on two different input streams. The two input streams differ in the update pattern of each room. For example, assume that a certain room, say R_i, reports 4 temperature readings in the following order: 89, 87, 79, and 78. Assume further that the selection predicate is as follows: Temperature $>$ 80. As a result, Room R_i will result in producing three output tuples as follows: +, u, -. However, assume that in another distribution, Room R_i reports the same four readings, but in a different order as follows: 89, 79, 87, 78. In this latter distribution, Room R_i will result in producing four output tuples as follows: +, -, +, -. Notice that although Room R_i has the same number of qualified readings (i.e., 0.5 selectivity), the number of output tuples depends on the distribution of the qualified tuples.

Figure 9(a) and Figure 9(b) give the effect of data distribution on the execution time and memory, respectively. The input size is 1.2 million tuples and the selectivity is varied from 0 to 1. The graphs illustrate that for the same selectivity value, a query with a nonkey predicate may encounter more processing time and memory than a query with a key predicate. Moreover, the execution time of the nonkey predicate varies from one data distribution to another. For example, for nonkey distribution 2, objects fluctuate in and out of the query boundary more than the fluctuation in distribution 1. As a result, distribution 2 causes more deletions and insertions into the state and hence results in more processing of negative tuples.

Figure 9(b) illustrates that the merged `Select-Tagger`'s state size may reach the maximum number of distinct key values which is the same as the separate Tagger's state size. This occurs due to the possibility that all the rooms might satisfy the query predicate at the same time. However, the CPU cost of the merged operator is always better than that of the separate operator, due to the savings in the number of selections and projections.

10.4 Performance of the Synchronizer Operator

In this section, we analyze the factors that affect the performance of the Synchronizer operator and study the effect of synchronization on query performance. In the first experiment, we studied the effect of the following two factors



(a) (b)

Fig. 10. Effect of number of objects.

on the performance of the Synchronizer operator: (1) the synchronization period, and (2) the number of distinct key values. We run a query pipeline that consists of Tagger and Synchronizer, where `TemperatureSource` is used as input to the Tagger, and the Tagger's output is used as input to the Synchronizer. The pipeline works as follows: Tagger reads a tuple from `TemperatureSource`, attaches the corresponding tag, and produces the tagged tuple in the output. Then, Synchronizer reads a tagged tuple, performs the corresponding summarizations in the buffer, and produces the buffered tuples as output when a synchronization tuple is received. Firgure 10 illustrates that as the synchronization period increases, the number of output tuples decreases. The reason for this decrease is that, in a bigger synchronization period, a larger number of update tuples are digested (i.e., summarized) by the Synchronizer operator, and hence a fewer number of tuples are processed by the upper Tagger operator. At every synchronization step, at most one output tuple can be produced for each room. For example, when the synchronization period is 0.1 second, one tuple is produced for each room every 0.1 second. However, when the synchronization step is 0.2, one tuple is produced for each room every 0.2 second. As a result, the number of output tuples for synchronization step 0.2 is almost half the number of tuples for synchronization step 0.1. The number of output tuples from the Synchronizer operator gives an indication for the required query resources, since these output tuples are processed by the upper operators in the pipeline. The conclusion is that although the synchronizer operator has its own cost, it reduces the number of tuples that are processed by the query pipeline. Hence, the cost of processing tuples by the synchronizer operator is balanced by reducing the number of tuples that are to be processed by other operators in the pipeline.

Figure 10 illustrates also that the number of distinct key values (e.g., number of rooms) affects the number of output tuples from Synchronizer. The reason is that at every synchronization point, Synchronizer produces at most one output tuple for every distinct key value. As a result, for the same number of input tuples and the same synchronization step, the number of output tuples from Synchronizer increases as the number of distinct key values increases. Figure 10(a) gives a comparison of the number of output tuples from Synchronizer when the number of objects in the input stream is 200 and 600, while the synchronization step is varied from 0.1 to 1 second. Synchronizer produces more tuples when the underlying stream has 600 identifiers. Figure 10(b) gives a summary of the relationships among the number of key values, the synchronization step, and the number of tuples. From the figure, we notice that: (1) For the same synchronization step, as the number of distinct key values increases, the number of tuples in the pipeline increases, and (2) for the same number of key values, as the synchronization step increases, the number of tuples flowing in the pipeline decreases. Notice that in append-only streams, each tuple in the stream has a distinct key value, hence the synchronization step has no effect on the number of tuples flowing in the pipeline. However, synchronizing an append-only stream has the effect of refreshing the query answer at regular time intervals, independent of the arrival pattern of the input tuples.

10.5 Aggregate Queries and Presynchronization

Consider the following aggregate query from the temperature-monitoring application: “ Find the number of hot rooms in each building, report modifications in the answer every 2 time units”. This aggregate query is expressed in `SyncSQL` in two steps as follows. First, we need to define a view that finds the number of hot rooms in each building as follows.

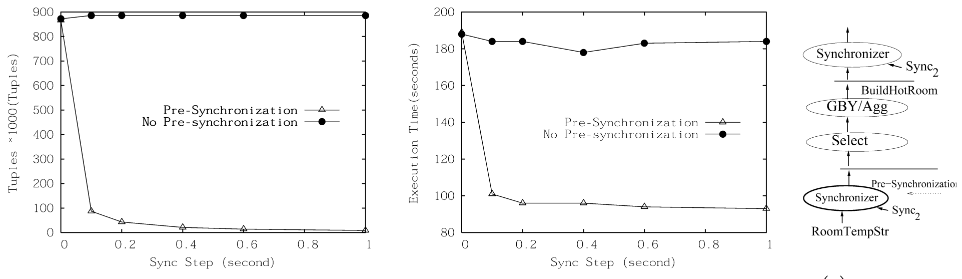
```
CREATE STREAMED VIEW BuildHotRooms AS
SELECT R.Building, Count(R.RoomID) as cntRooms
FROM ℜ(RoomTempStr) R
WHERE Temperature > 85
GROUP BY R.Building
```

Notice that Attribute `Building` represents the key attribute for the output stream from the `BuildHotRooms` view. An update tuple is produced in the output from the `BuildHotRooms` view whenever a room enters or exits the query range. Notice that the same building receives several updates if the building has more than one hot room. Notice also that the query issuer asks to be notified by the modifications in each building “once” every two time units. In order to get the desired output, we apply the desired synchronization (i.e., every 2 time units) on `BuildHotRooms`’s output as follows.

```
SELECT V.Building, V.cntRooms
FROM ℜ_{Sync_2}(BuildHotRooms) V
```

The output stream from the last query includes at most one update tuple for each building on every synchronization time point, hence achieving the desired query semantics. Figure 11(c) gives the query pipeline for `BuildHotRooms` view and the subsequent query. Notice that the Synchronizer’s state size equals the maximum number of buildings because `R.Building` represents the key field for the output stream tuples `BuildHotRooms`.

The presynchronization optimization. In the room temperature-monitoring application, each room sends temperature updates more than once in every time unit as explained in Section 10.1. As a result, the same room may result in producing several update tuples in `BuildHotRooms`’s output stream for the corresponding building. The update tuples that are produced from the same



(a) tuples (b) execution time (c) query pipeline

Fig. 11. Effect of presynchronization.

building are summarized by Synchronizer to produce a single output for each building in every synchronization step. Notice that all the updates that result from the same room belong to the same building, and hence are also summarized by Synchronizer. This observation highlights the possibility of presummarizing the updates for each room and includes only one update from each room in the final building summarization. The presummarization can be achieved by performing a presynchronization on the `RoomTempStr` stream before being processed by `BuildHotRooms`'s pipeline. Figure 11(c) gives the optimized query pipeline by adding an additional Synchronizer operator at the bottom of the pipeline. With the added Synchronizer, each room has at most one update tuple to be processed by the aggregate operator in each synchronization period. Presynchronization results also in reducing the CPU time taken by the aggregate operator, since less tuples flow in the pipeline. Presynchronization is similar in spirit to eager aggregation in traditional databases [Yan and Larson 1995].

Figure 11 gives a performance comparison of the pipeline in Figure 11(c) before and after adding the bottommost Synchronizer, when processing an input stream of 1.2 million tuples. The input stream has 200 distinct key values and the arrival rate is 20000 tuples/second. Figure 11(a) gives the number of tuples processed by the aggregate operator while varying the synchronization step from 0 to 1. All the input tuples are processed by the aggregate operator when no presynchronization is performed. However, the number of tuples is reduced significantly when pre-synchronization is applied, since the bottom Synchronizer digests many input tuples. Figure 11(b) gives the query execution time that is proportional to the number of tuples processed by the pipeline, hence, presynchronization reduces the execution time by about 50%.

10.6 Experimental Verification of the Cost Model

In this section, we experimentally verify the accuracy of the proposed cost model to estimate the CPU cost of `SyncSQL` pipelines. The experiments are conducted over a given set of concurrent `SyncSQL` queries. First, we enumerate several execution pipelines for the given set of queries. Then, we estimate the cost of executing the different pipelines while changing the following parameters: the input update pattern, the input data distribution, the synchronization period, and the

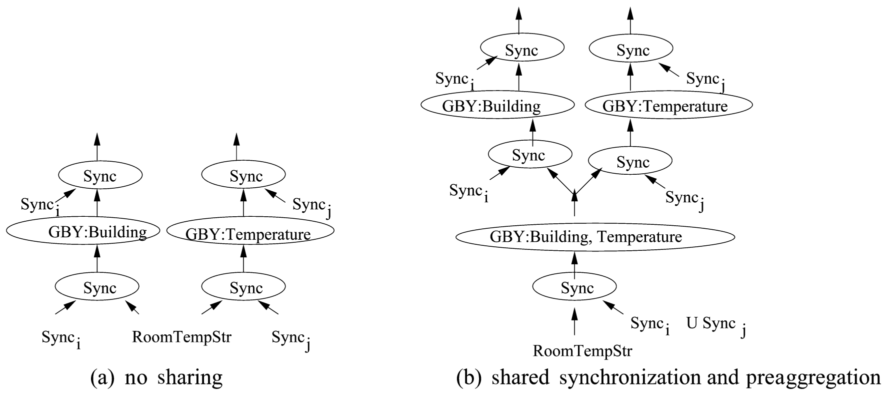


Fig. 12. Possible execution plans for two concurrent aggregate queries.

number of queries. Next, we run the query pipelines in the `Nile-SyncSQL` prototype and measure the execution times. The cost model is verified by matching the measured results with the estimated results.

10.6.1 *Workload Queries and Plan Enumeration.* Experiments in this section are conducted over a set of Group-by queries from the temperature-monitoring application. The goal of these experiments is to illustrate the benefits of using views for the shared execution of continuous queries. The results in this section are conducted from the shared execution of the following two queries (including more queries is straightforward).

—`BuildingGroups`: For each building, find the number of rooms with a temperature greater than 80. Report modifications in the answer every `i` time units.

—`TemperatureGroups`: For each temperature value `t` that is greater than 80, find the number of rooms that have `t` as the room's temperature. Report modifications in the answer every `j` time units.

Both the `BuildingGroups` and the `TemperatureGroups` queries are aggregate queries over the stream `RoomTempStr`. However, Query `BuildingGroups` groups the input tuples based on Attribute `Building` while Query `TemperatureGroups` groups the input tuples based on Attribute `Temperature`. Also, the two queries differ in the refresh granularity (i.e., require different synchronization streams). The options for sharing the execution of the two queries are worth exploring, since the two queries are executed over the same input stream (i.e., `RoomTempStr`). In the following we examine two possible execution paradigms for the concurrent queries (the corresponding query pipelines are given in Figure 12).

(1) *Nonshared execution:* where the two queries are executed independently without sharing any operations as shown in Figure 12(a), and (2) *Shared synchronization and preaggregation:* Another shared pipeline where both aggregation and synchronization are shared between the two queries is shown in Figure 12(b). The shared view's Synchronizer uses a synchronization stream

that represents the union of the two queries' synchronization streams. The shared views' Aggregate (the operator that is labeled "GBY:Building, Temperature" in Figure 12(b)) groups the input tuples based on both the `Building` and `Temperature` attributes and counts the number of tuples in each group. The output groups from the shared view are then aggregated by the upper Group-by operators (the operators that are labeled "GBY:Building" and "GBY:Temperature" in Figure 12(b)) to produce the required building and temperature groups. Notice that "GBY:Building" and "GBY:Temperature" add up the number of tuples in the subgroups to produce the count of tuples in the final group. Notice also that the output stream from the "GBY:Building, Temperature" operator has a primary key that consists of two attributes, namely the Building and Temperature attributes. If the number of `Building-Temperature` groups is less than the number of rooms, then the number of tuples processed by the upper Group-by operators is less than those of the corresponding operators in Pipeline a. However, the shared aggregate operator is an additional overhead in Pipeline b. Then, the output groups from the shared view are aggregated by the upper Group-by operators. The shared Synchronizer performs presynchronization and hence reduces the number of input tuples to the view's Aggregate.

The pipelines in Figure 12 consist of two types of operators, Synchronizer and Group-by. Notice that two synchronizer operators are used with each aggregate operator to apply the presynchronization optimization as described in Section 10.5. In order to estimate the cost of executing the pipelines in Figure 12, we need to estimate two numbers as follows: (1) NS: the number of tuples processed by the Synchronizer operators, and (2) NG: the number of tuples processed by the Group-by operators. Assume that the cost of processing one tuple in any Synchronizer equals c_1 while the cost of executing one tuple in any Group-by equals c_2. Hence, using the equations in Section 9, the cost of executing a pipeline is

$$C_{Pipeline} = NS * c_1 + NG * c_2.$$

The values of NS and NG differ from one pipeline to another and depend on the following parameters: (1) the input streams update patterns, (2) the number of key values in the input streams, (3) the number of Groups that are produced by the Group-by operators, and (4) the synchronization periods. In the following section, we study the effect of the various parameters on the execution cost.

10.6.2 *Improving the Performance Using Views.* In this section, we study the effect of using views for the shared execution of queries. We run an experiment to compare the performance of the nonshared execution pipeline in Figure 12(a) and the shared execution pipeline in Figure 12(b) using following parameters.

—Number of rooms is 2000, number of buildings is 20, the number of different temperature values is 10. As a result, the maximum possible number of building-temperature groups is 200.
—Rooms report temperature updates in a uniform pattern where each room reports an update every 1 time unit.

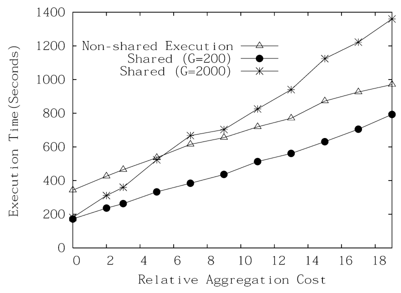


Fig. 13. Nonshared vs. shared execution of aggregate queries.

—`BuildingGroups`'s synchronization is every 12 time units, and `TemperatureGroups`'s synchronization is every 15 time units.

Using the cost model that is presented in Section 9, the execution costs of the nonshared paradigm in Pipeline a and the shared paradigm in Pipeline b for 650 time units can be estimated by the following equations.

—$C_a(650) = 2798000 * c_1 + 198000 * c_2$
—$C_b(650) = 1841300 * c_1 + 191800 * c_2$

The cost equations show that the shared execution in Pipeline b causes a 40% reduction in NS and a 10% reduction in NG. The reason for the reduction in NS is that, in Pipeline b, the input tuples are processed by only one Synchronizer operator (i.e., the shared Synchronizer operator) in contrast to being processed twice in Pipeline a. The reason for the reduction in NG is that in Pipeline b, the upper Group-by operators process only one tuple for each building-temperature group at every synchronization point in contrast to processing one tuple for each room in Pipeline a. Notice that, in this experiment, the update rate of the objects (i.e., every 1 time unit) is much higher than the rate of the synchronization points. This means that at every synchronization point, several updates for the same object are accumulated, hence causing a big reduction in the number of tuples that are processed by the upper operators in the query pipeline.

Effect of the grouping factor. If we change the input parameters such that there are 200 buildings, then the number of building-temperature groups can reach up to 2000 at which the cost of Pipeline a is not affected, while the cost of Pipeline b is estimated by the following equation.

$$C_b(650) = 2069000 * c_1 + 370000 * c_2$$

When the number of building-temperature groups is 2000, the shared execution pipeline consumes 25% less of the synchronization operations and 1.8% more aggregations than the nonshared execution in Pipeline a. Hence the preference between the two pipelines depends on the values of c_1 and c_2.

Figure 13 gives a comparison of the execution times of Pipeline a, Pipeline b with 200 building-temperature groups, and Pipeline b with 2000 building-temperature groups while changing the cost of aggregation, c_2. The experimental results show that when the number of groups is 200, the shared execution

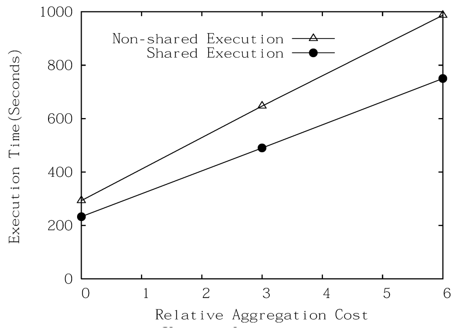


Fig. 14. Effect of input parameters.

can achieve up to a 50% savings in the execution time as compared to the nonshared execution. However, when the number of groups is 2000, the shared execution performs better than the nonshared execution only for small values of c_2 (i.e., for inexpensive aggregate functions). As the cost of aggregation increases, the execution time of the shared pipeline increases and the nonshared execution is preferred, since it can achieve up to a 70% reduction in the execution time. The conclusion is that the preference whether to share the execution or not depends on (1) the grouping factor (i.e., in the number of groups in each Group-by operator), and (2) the cost of the aggregation function.

Effect of the input parameters. The experiment in this section illustrates the effect of the input parameters on the performance. Assume that we run the same experiment as before but with the following parameters.

—Number of rooms is 5000, number of buildings is 100, number of different temperatures is 20. Then, the maximum number of building-temperature groups is 2000.
—Rooms report temperature updates in a uniform pattern, but different rooms have different intervals between the updates as follows: 2500 rooms each report an update every 2 time units, 1500 rooms each report an update every 10 time units, and 100 rooms each report an update every 15 time units.
—`BuildingGroups`'s synchronization is every 6 time units, while `TemperatureGroups`'s synchronization is every 12 time units.

Figure 14 illustrates that the shared execution pipeline improves the execution time over that of the independent execution pipeline. However, the percentage of execution time reduction is less than that in Figure 13. The percentage of reduction in execution time is 40% in contrast to a 70% in Figure 13. The reason for the difference in the performance gain is that in the earlier parameter settings, the update rate is much higher than the frequency of the synchronization points. However, in the parameters in this section, the synchronization points are as frequent as the object update rate. Hence, not too many updates are accumulated by the Synchronizer operator. As a result, synchronization has only a small effect on the number of tuples that flow in the query pipeline.

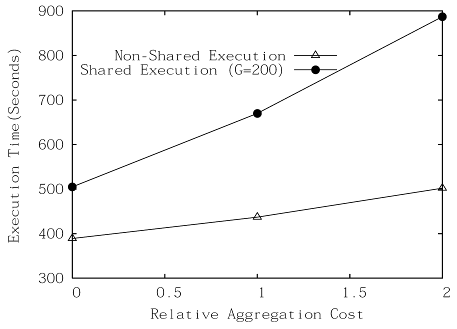


Fig. 15. Effect of using views.

10.6.3 *Stream Views Can Worsen the Performance.* The experiments of the previous section illustrate that using views improves the query performance. However, the improvement factor depends on the query settings. In this section, we show that for some input parameters, using views may worsen the query performance. Consider two queries, `BuildingGroups` and `TemperatureGroups`, with the following input parameters.

—Number of rooms is 2000, number of buildings is 100, number of different temperature values is 20. Then, the number of building-temperature groups is 2000.

—Rooms report temperature updates in a uniform pattern but with different intervals as follows: 500 rooms each report an update every 10 time units, 500 rooms each report an update every 13 time units, and 1000 rooms each report an update every 17 time units.

—`BuildingGroups`'s synchronization is every 15 time units while `TemperatureGroups`'s synchronization is every 12 time units.

Using the proposed cost model and the equation that is presented in Section 10.6.1, the execution cost of the nonshared execution pipeline (Pipeline a) and the shared execution pipeline (Pipeline b) for 650 time units can be estimated by the following equations.

—$C_a(650) = 444924 * c_1 + 171886 * c_2$

—$C_b(650) = 743007 * c_1 + 288318 * c_2$

The cost equations show that the shared execution paradigm requires more synchronization operations and more aggregation operations than the nonshared execution. As a result, in this case, nonshared execution is always preferred over shared execution. The analytical results are confirmed by the experimental results that are given in Figure 15. The graphs in Figure 15 illustrate that nonshared execution achieves up to 50% reduction in the execution time. The reason for the winning performance of nonshared execution is that the update rates for most of the rooms are slower than that of the synchronization rate. As a result, synchronization does not result in any accumulation of updates and hence does not reduce the number of tuples to be processed. Moreover, the number of intermediate building-temperature groups is the same as the number of rooms. Hence, shared execution does not result in any reduction in the number of tuples. The shared view is nothing but an additional overhead, hence causing bad execution times.

11. CONCLUSIONS

This article introduces a framework to support views in data stream management systems. First, the article proposes the SyncSQL query langauge that expresses composable queries (or views) over streams. SyncSQL uses the tagged stream model in which a data stream is a sequence of modifications over a relation. Then, the article introduces the synchronization principle that empowers SyncSQL by a mechanism to express queries with arbitrary refresh conditions. The article introduces an algebraic framework for SyncSQL queries in which synchronized relations are the main data type over which queries are expressed. Several new equivalences and transformation rules are given to govern the relationship among SyncSQL operators. The transformation rules are needed by a query optimizer to enumerate the query plans. Then, based on the introduced algebraic framework, the article introduces a query-matching algorithm to judge the containment relationship among SyncSQL expressions. Then, the Nile-SyncSQL prototype server is introduced to support SyncSQL queries over streams. In addition, a cost model is proposed to estimate the CPU cost of executing a SyncSQL query. The cost model is used by the query optimizer to choose the best execution plan for a given set of queries. An experimental study is provided to evaluate the performance of Nile-SyncSQL. The experimental results illustrate that sharing the execution using views can achieve up to a 70% improvement in performance. At the same time, views may worsen the performance for some query settings. The decision as to whether to share the execution using views or not can be made in advance with the proposed cost model.

REFERENCES

Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stone-braker, M., Tatbul, N., and Zdonik, S. B. 2003. Aurora: A new model and architecture for data stream management. *VLDB J. 12*, 2, 120–139.

Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: Semantic foundations and query execution. *VLDB J. 15*, 2, 121–142.

Arasu, A. and Widom, J. 2004. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the International Conference on Very Large Databases (VLDB)*.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. 2002. Models and issues in data streams. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.

Babu, S., Munagala, K., Widom, J., and Motwani, R. 2005. Adaptive caching for continuous queries. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

Bonnet, P., Gehrke, J. E., and Seshadri, P. 2001. Towards sensor database systems. In *Proceedings of the Internationa Conference on Mobile Data Management (MDM)*.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD Intenational Conference on Management of Data*.

CRANOR, C. D., JOHNSON, T., SPATSCHECK, O., AND SHKAPENYUK, V. 2003. Gigascope: A stream database for network applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

DALVI, N. N., SANGHAI, S. K., ROY, P., AND SUDARSHAN, S. 2001. Pipelining in Multi-Query Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

EISENBERG, A., MELTON, J., KULKARNI, K., MICHELS, J.-E., AND ZEMKE, F. 2004. SQL:2003 has been published. *SIGMOD Rec. 33*, 1, 119–126.

GHANEM, T. M., AREF, W. G., AND ELMAGARMID, A. K. 2006. Exploiting predicate-window semantics over data streams. *SIGMOD Rec. 35*, 1, 3–8.

GHANEM, T. M., HAMMAD, M. A., F. MOKBEL, M., AREF, W. G., AND ELMAGARMID, A. K. 2007. Incremental evaluation of sliding-window queries over data streams. *IEEE Trans. Knowl. Data Engin. 19*, 1, 57–72.

GOLAB, L. AND OZSU, M. T. 2003. Issues in data stream management. *SIGMOD Rec. 32*, 2, 5–14.

GOLDSTEIN, J. AND LARSON, P.-Å. 2001. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

GRIFFIN, T. AND LIBKIN, L. 1995. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

GUPTA, A. AND MUMICK, I. S., EDS. 1999. *Materialized Views: Techniques, Implementation, and Applications*. MIT Press.

HAMMAD, M. A., FRANKLIN, M. J., AREF, W. G., AND ELMAGARMID, A. K. 2003. Scheduling for shared window joins over data streams. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*.

KANG, J., NAUGHTON, J. F., AND VIGLAS, S. 2003. Evaluating window joins over unbounded streams. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

LARSON, P.- A. AND YANG, H. Z. 1985. Computing queries from derived relations. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*.

LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKER, P. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

MAIER, D., LI, J., TUCKER, P., TUFTE, K., AND PAPADIMOS, V. 2005. Semantics of data streams and operators. In *Proceedings of the International Conference on Database Theory (ICDT)*.

RYVKINA, E., MASKEY, A. S., CHERNIACK, M., AND ZDONIK, S. 2006. Revision processing in a stream processing engine: A high-level design. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

SRIVASTAVA, U. AND WIDOM., J. 2004. Flexible time management in data stream systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.

TUCKER, P., MAIER, D., SHEARD, T., AND FEGARAS, L. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Engin. 15*, 3, 555–568.

YAN, W. P. AND LARSON, P.-A. 1995. Eager aggregation and lazy aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

ZANIOLO, C., LUO, R., WANG, H., BAI, Y., AND THAKKAR, H. 2002. An introduction to the expressive stream language. WEB Information System Laboratory, UCLA, CS Department. http://wis.cs.ucla.edu/stream-mill.

Received April 2009; Accepted July 2009