2006

# SyncSQL: A Language to Express Views over Data Streams

Thanaa M. Ghanem

Per-Ake Larson

Walid G. Aref
*Purdue University*, aref@cs.purdue.edu

Ahmed K. Elmagarmid
*Purdue University*, ake@cs.purdue.edu

## Report Number:

06-012

# SYNCSQL: A LANGUAGE TO EXPRESS VIEWS OVER DATA STREAMS

**Thanaa M. Ghanem**
**Per-Ake Larson**
**Walid G. Aref**
**Ahmed K. Elmagarmid**

**CSD TR #06-012**
**July 2006**

# *SyncSQL*: A Language to Express Views over Data Streams

Thanaa M. Ghanem[1] Per-Åke Larson[2] Walid G. Aref[1] Ahmed K. Elmagarmid[1]

[1]Department of Computer Science, Purdue University, ghanemtm,aref,ake@cs.purdue.edu

[2]Microsoft Research, palarson@microsoft.com

## Abstract

*Prior work on languages to express continuous queries over streams has defined a stream as a sequence of tuples that represents an infinite append-only relation. In this paper, we show that composition of queries is not possible in the append-only model. Query composition is a fundamental property of any query language - composition makes it possible to build up complex queries from simpler queries. We then propose a query language, termed* Synchronized SQL *(or* SyncSQL*), that defines a stream as a sequence of modify operations (i.e., insert, update, and delete) against a relation with a specified schema. Inputs and outputs in any* SyncSQL *query are interpreted in the same way and, hence,* SyncSQL *expressions can be composed. Coarser periodic refresh requirements are typically expressed as sliding-window queries. We generalize this approach by introducing the synchronization principle that empowers* SyncSQL *with a formal mechanism to express queries with arbitrary 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.*

## 1 Introduction

Query languages in the streaming literature (e.g., [2, 7, 8, 11, 24]) define a stream as a sequence of tuples that represents an infinite append-only relation. Languages based on the append-only model are not closed, that is, the result of a query expression is not necessarily an append-only relation. This has the effect that query expressions cannot be freely composed, that is, expressing a query in terms of one or more sub-queries as can be done, for example, with SQL queries in relational databases. Composition is a fundamental property of any query language but it requires that query inputs and outputs are interpreted in the same way. However, in the append-only stream model a continuous query may not be able to produce an append-only output even when input streams represent append-only relations.

For example, consider an application monitoring 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 cars entering and exiting the lot, respectively. A reasonable query in this environment is $Q_1$:"*Continuously keep track of the identifiers of all cars inside the parking lot*". The answer of $Q_1$ is a *view* that, at any time point $T$, contains the identifiers for cars 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, $Q_1$ can be regarded as a *materialized view* that is defined as the set-difference between the two relations $\Re(S_1)$ and $\Re(S_2)$. As tuples arrive on $S_1$ and $S_2$, the corresponding relations are modified, and the relation representing the result of $Q_1$ is updated to reflect the changes in the inputs. The result of $Q_1$ is updated by *inserting* identifiers of cars entering the lot and *deleting* identifiers of cars exiting the lot. Notice that, although the input relations in $Q_1$ change by only inserting tuples (i.e., append only), the output of $Q_1$ changes by both insertions and deletions.

The answer to query $Q_1$ can be output either as (1) a *complete* answer, or (2) an *incremental* answer. In the first case, at any time point $T$, the issuer of $Q_1$ sees a state, i.e., a relation containing identifiers of all cars inside the lot at time $T$. In the second case, the issuer of $Q_1$ receives a stream that represents the changes (i.e., insertions and deletions) in the state. The output in the incremental case is interpreted in the same way as the inputs, namely, as a stream that represents modifications to an underlying relation. However, $Q_1$'s incremental answer cannot be produced or consumed by a query in a language that models a stream as an append-only relation. Existing languages may produce output streams from $Q_1$ but the output streams are interpreted differently from the input streams. For example, the output may be modeled as a stream representing a concatenation of serializations of the complete answer (e.g., RStream in CQL [2], and the output of window queries in TelegraphCQ [8]). As another alternative, CQL divides the output into two append-only streams such that one stream represents the insertions in the output and the other stream represents the deletions (i.e., IStream and DStream).

The different interpretation and the division of an output stream prevents composition of queries, that is, using the output of a query as the input to another queries or building up complex query expressions from simpler expressions. Composition is a fundamental requirement on any query language and particularly important in streaming environments that are characterized by concurrent, overlapping queries. For example, consider the following query, $Q_2$, from the same application: *"Group the cars inside the parking lot by type (e.g., trucks, cars, or buses). Continuously keep track of the number of cars in each group"*. By analyzing the two queries, $Q_1$ and $Q_2$, it is obvious that $Q_2$ is an aggregate query over the output of $Q_1$. This observation motivates the idea of defining $Q_1$ as a view, say $V_1$ and then, expressing both $Q_1$ and $Q_2$ in terms of $V_1$. However, realizing this requires a language that allows query composition.

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 cars with every change in the parking lot, $Q_2$ may be interested in updating the count of cars in each group *every five minutes*. This refresh condition is based on time but a powerful language should allow a user to express more general refresh conditions based on time, tuple arrival, events, relation state, and so on.

In addition to preventing query composition, the append-only model limits the applicability of the language because streams may have denotations other than the append-only relation [22]. For example, update streams are used in applications where objects continuously update their values. 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, $Q_3$ : *"Continuously keep track of the rooms that have temperature greater than 80"*. Neither the input nor the output streams in $Q_3$ represent append-only relations. The input in $Q_3$ 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. The output tuples from $Q_3$ represent incremental changes in the answer and include insertions and deletions for rooms that switch between satisfying and not satisfying the query predicate.

## 1.1 Our Approach

We can summarize the limitations of the existing continuous query languages as follows. (a) Cannot express queries over streams other than the append-only relation representation. (b) Cannot produce incremental answer for queries that do not produce an append-only output. (c) Cannot always compose queries because of the different interpretation and/or division of the output streams. (d) Refresh condition are restricted to be either time or tuple-based.

In this paper, we introduce a continuous query language for data streams, termed `Synchronized SQL` (`SyncSQL` for short), that avoids the previous limitations. In contrast to other languages, `SyncSQL` defines the stream as a sequence of modify operations (i.e., insert, update, and delete) against a relation with a specified schema. Basically, a continuous query in `SyncSQL` is semantically equivalent to a *materialized view* where the inputs are relations that are modified by streams of modify operations. The answer of the query is another stream of modify operations that represent changes in the result of the view. This is equivalent to incremental maintenance of materialized views [17]. The unified representation of query inputs and outputs enables the composition of `SyncSQL` expressions, and as a result, gives the ability to express and exploit views over streams.

To cope with the coarser refresh requirement of continuous queries, we introduce the *synchronization principle*. The idea is to formally specify synchronization time points at which the input tuples are processed by the query pipeline. Input tuples that arrive between two consecutive synchronization points are accumulated and reflected in the output at once at the next synchronization point. The synchronization principle makes it possible to (1) express queries with arbitrary refresh conditions, and (2) formally reason about the containment relationship among queries with different refresh periods.

The contributions of this paper are summarized as follows:

- **SyncSQL semantics and syntax:** We define concise semantics and syntax for continuous queries and views over streams.
- **SyncSQL algebra:** We lay the algebraic foundation for `SyncSQL` by providing data types, operators, algebraic laws and transformation rules that are needed to enumerate query plans.
- **Shared execution using query composition:** Based on the algebraic framework, we propose a query matching algorithm that is used to deduce the containment relationships among query expressions. The containment relationship is used to achieve shared execution using query composition.
- **Execution model:** We present a pipelined and incremental execution model to efficiently realize `SyncSQL` queries in a data stream management system.

### 1.2 Paper Outline

The rest of the paper is organized as follows. Section 2 introduces the semantics and syntax of SyncSQL. The synchronization principle is explained in Section 3. In Section 4, we lay the algebraic foundation for SyncSQL. The shared query execution algorithm is given in Section 5. In Section 6, we give an incremental execution model for SyncSQL queries. Section 7 surveys the existing works for continuous queries and contrasts our approach with the other approaches. Finally, Section 8 concludes the paper.

## 2 SyncSQL Semantics and Syntax

In short, a continuous SyncSQL query is semantically equivalent to a materialized view over one or more relations where the input relations are updated by *streams* of modify operations.

### 2.1 Stream, Query, and View Semantics

**Stream semantics.** We distinguish between two types of streams: *raw* input streams and *tagged* streams. A *raw* input stream is a sequence of tuples (or values) that are sent by the remote data sources (e.g., sensors). On the other hand, a *tagged* stream is a stream of modify operations over a specified schema where the modify operations can be either insert (+), update(u) or delete(-). A raw input stream is transformed into a tagged stream before being used as input in a query. This is similar to the relational model in traditional databases where the raw data has to be transformed into relations before being used in a query.

The function that transforms a raw input stream to a corresponding tagged stream is application-dependent where the same raw input stream can produce different tagged streams under different transformation functions. For example, in a temperature-monitoring application, a *raw* input stream, say TemperatureSource, is sent by remote sensors where an input tuple in the raw stream reports a room temperature. A tuple in the TemperatureSource stream consists of two attributes: RoomID and Temperature. One application, say Application$_1$, may consider TemperatureSource as an update stream over the various rooms temperature. 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, another application, say Application$_2$, may view the TemperatureSource stream as just a sequence of temperature readings and ignore the RoomID attribute.

Assume that an input tuple in TemperatureSource is denoted by "<RoomID,Temperature>Timestamp". In Application$_1$, TemperatureSource represents an update stream over the various room temperatures and the corresponding tagged stream, say RoomTempStr, consists of *insert* and *update* operations. Basically the tagging procedure takes an input TemperatureSource tuple and produces a corresponding tagged tuple in RoomTempStr as follows: the first tuple in TemperatureSource with a certain RoomID value is transformed into a corresponding *insert* operation "**+**<RoomID,Temperature>Timestamp" in RoomTempStr. A subsequent tuple in TemperatureSource with the same RoomID is transformed into an *update* tuple "**u**<RoomID,Temperature>Timestamp" in RoomTempStr. Notice that the tagging function needs to keep a list of the observed key (i.e., RoomID) values so far.

On the other hand, in the case of Application$_2$, TemperatureSource represents an infinite append-only relation and the corresponding tagged stream, say TempStr, is a sequence of *insert* operations where each tuple "<RoomID,Temperature>Timestamp" in TemperatureSource is transformed to a corresponding insert operation "**+**<RoomID,Temperature>Timestamp" in TempStr.

In the query processing phase, the transformation (or tagging) function is implemented inside an operator, called Tagger, that is placed at the bottom of a query pipeline. In Application$_1$, the functionality of the Tagger operator is similar to that of the MERGE (or UPSERT) operator in the SQL:2003 standard [12]. Basically, in Application$_1$, Tagger needs to keep a list of all the observed key values (i.e., RoomID) so far. The size of the key list has an upper bound that is equal to the maximum number of rooms. However, implementing the tagging function as an operator opens the room for the query optimizer to re-order the pipeline and optimize the memory consumption. For example, the Tagger operator can be pulled above the Select operator so that only qualified rooms are stored in the key list. The details of query processing and optimization is beyond the scope of this paper.

**Example 1** This example demonstrates the syntax for defining streams and the mapping from raw to tagged streams. The raw TemperatureSource stream is defined in SyncSQL by the following statement:

```
REGISTER SOURCE TemperatureSource
(char RoomID, int Temperature) From
port5501
```

where RoomID and Temperature represent the stream schema and port5501 is the port at which external sources report tuples. The tagged streams are defined over the source TemperatureSource as follows:

```
RoomTempStr:   CREATE STREAM
RoomTempStr
```

```
                OVER TemperatureSource
                KEY RoomID

TempStr:        CREATE STREAM TempStr
                OVER TemperatureSource
                KEY NULL
```

**Running example:** Assume the following tuples arrived at `TemperatureSource`: `<a,100>1, <b,75>2, <c,80>3, <a,95>4, <b,85>5`.

`RoomTempStr`**:** The following tuples represent the corresponding tagged `RoomTempStr` stream: **+**`<a,100>1`, **+**`<b,75>2`, **+**`<c,80>3`, **u**`<a,95>4`, **u**`<b,85>5`. Notice that the tuple `<a,100>1` is mapped to **+**`<a,100>1` while `<a,95>4` is mapped to **u**`<a,95>4`.

`TempStr`**:** The following tuples represent the corresponding `TempStr` tuples: **+**`<a,100>1`, **+**`<b,75>2`, **+**`<c,80>3`, **+**`<a,95>4`, **+**`<b,85>5`. Notice that all the tuples in `TempStr` are *insert* operations.

**The relational view of a tagged stream:** All input and output streams in a `SyncSQL` query are tagged streams. An input tuple in a tagged stream is denoted by "`Type<Attributes>Timestamp` where type can be one of three values: +, u, or -. Any tagged stream, say `S`, has a corresponding continuous relational view, termed $\Re$`(S)`. The relational view of a tagged stream is a time-varying relation that is continuously modified by the arriving `S`'s tuples. $\Re$`(S)`'s schema consists of two parts as follows: (1) a set of attributes that corresponds to `S`'s underlying schema, and (2) a timestamp attribute, termed `TS`, that corresponds to the `Timestamp` field of `S`'s tuples. For example, the relational view of the `RoomTempStr`, stream that is defined in Example 1, is denoted by $\Re$(RoomTempStr) and the relation's schema consists of three attributes: RoomID, Temperature, and TS. Notice that although `Timestamp` is not a part of `S`'s schema, `Timestamp` is mapped to $\Re$(S) in order to be able to express time-based windows over `S` as will be shown in Section 2.2. At any time point, say `T`, $\Re$`(S)` is denoted as `R[S(T)]` and is the relation resulting from applying the modify operations with timestamps less than or equal to `T` in an increasing order of timestamp. According to the underlying application, $\Re$`(S)` can be modified by either inserting tuples (i.e., append-only), or by general modify operations.

**Definition 1. Time-varying relation.** A time-varying relation $\Re$`(S)` is the relational view of a tagged stream `S` such that $\Re$`(S)` = `R[S(T)]` $\forall$ `T`, where `T` is any point in time.

**Example 2** This example illustrates the mapping from an input stream, say `S`, to the corresponding time-varying relation relation $\Re$`(S)`. Assume that `S`'s underlying schema has only one attribute, say $\texttt{Attr}_1$, that is considered a key. Figure 1a shows the following `S`'s tuples: `+<a>1`, `+<b>3`, `-<a>4`, `+<c>5`. Figure 1b shows the corresponding time-varying relation, $\Re$`(S)` where each record in $\Re$`(S)` has two attributes, $\texttt{Attr}_1$ and `TS`. The relation `R[S(T)]` is the relation that reflects the input stream tuples that have `Timestamp` less than or equal `T`. For example, `R[S(1)]` reflects the insertion of one tuple with value "a" and `R[S(3)]` reflects, in addition to "a", the insertion of "b" and so on.

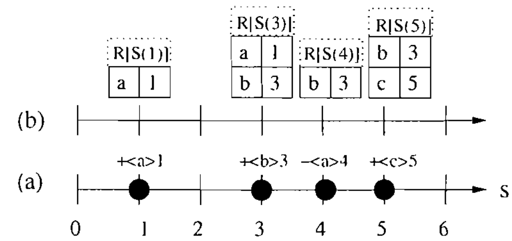


**Figure 1. Illustrating Time-varying Relations.**

**Query semantics.** A continuous query over `n` tagged streams, $S_1$ ... $S_n$, is semantically equivalent to a *materialized view* that is defined by an SQL expression over the time-varying relations, $\Re(S_1)$ ... $\Re(S_n)$. At any time point, `T`, the query answer reflects the contents of the underlying relations at time `T`, (i.e., $\texttt{R}[S_1\texttt{(T)}]$ ... $\texttt{R}[S_n\texttt{(T)}]$). 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 query answer in a way similar to incremental maintenance of materialized views [17].

**Query outputs.** The output of a query can be provided in two forms as follows:
(1) **`STREAMED`** output where the output is a tagged stream that consists of modify operations that represent the *deltas* in the answer. The output of a `STREAMED` query is *incremental* in the sense that a modify operation is produced in the output whenever a modification (i.e., insert, update, or delete) takes place in the query answer. As will be discussed in Section 3.1, timestamps are attached to the output stream tuples so that the output stream can be used as input in another query (i.e., query composition).
(2) **`COMPLETE`** output where the output of the query is stored in a time-varying relation. The time-varying relation is modified by the query pipeline whenever any of the input relations is modified. In this case, at every time point, the query issuer gets a non-incremental *complete* query answer.

**Example 3** This example demonstrates the semantics and syntax of `SyncSQL` queries. The temperature monitoring query $\texttt{Q}_3$ that is used in Section 1 , is expressed in `SyncSQL` as follows:

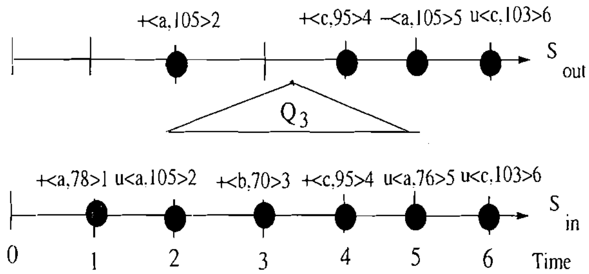


**Figure 2. $Q_3$ Running Example.**

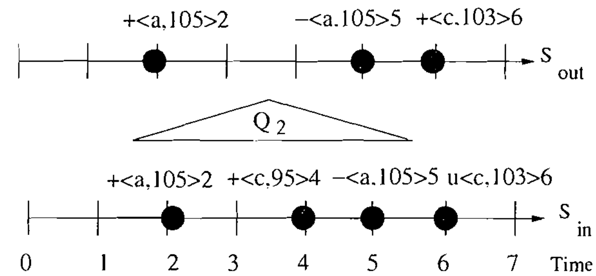


**Figure 3. Query Composition.**

```
select STREAMED RoomID, Temperature
from ℜ(RoomTempStr) R
where R.Temperature > 80
```

where RoomTempStr is the input tagged stream that is defined in Example 1. ℜ(RoomTempStr) is the corresponding time-varying relation. The keyword STREAMED indicates that the output needs to be another stream of modify operations. The output stream of this query includes *insert* (or *update*) operations for rooms that qualify the predicate "R.Temperature $>$ 80" and/or *delete* operations for previously qualified rooms that disqualify the predicate due to a temperature update.

**Running example.** Assume the following RoomTempStr's tuples have arrived at $Q_3$: +<a,78>1, u<a,105>2, +<b,70>3, +<c,95>4, u<a,76>5, u<c,103>6. Figure 2 shows the input and output streams in $Q_3$ are as follows. The input tuple +<a,78>1 does not result in producing any output tuples, while u<a,105>2, which arrives at time 2, results in *inserting* Room "a" in the answer via the output tuple +<a,105>2. Similarly, when +<c,95>4 arrives, Room "c" is *inserted* in the query answer via +<c,95>4. Later, when u<a,76>5 arrives, Room "a" is *deleted* from the output via -<a>5. Notice that the "Attributes" part of the delete tuple -<a>5 specifies only the key value which is enough to perform deletion. Finally, when u<c,103>6 arrives, a corresponding tuple u<c,103>6 is produced in the query answer to report that Room "c" still qualifies the query predicate, but with a new temperature.

**Views over streams.** The unified interpretation (as tagged streams) of SyncSQL query inputs and outputs enables SyncSQL to define and exploit views over streams. Basically, a view over streams is a named SyncSQL query expression that is defined once and, then, can be used as input in any other query. For example, a view, say $V_i$, can be used as input in a query, say $Q_i$, if $Q_i$'s expression (or part of it) is *equivalent* or is *contained* in $V_i$'s expression. In Section 5, we give an algorithm to deduce the containment relationships among SyncSQL expressions.

**Example 4** This example demonstrates answering queries using views. Consider the following query, $Q_4$ (from the same temperature-monitoring application as $Q_3$): *"Continuously keep track of the rooms that have temperature greater than **100**"*. Similar to $Q_3$, $Q_4$ can be expressed over RoomTempStr as follows:

```
select STREAMED RoomID, Temperature
from ℜ(RoomTempStr) R
where R.Temperature > 100
```

It is obvious that $Q_4$ is contained in $Q_3$. As a result we can benefit from query composition by defining $Q_3$ as a view, say $\text{HotRooms}_1$, as follows:

```
create STREAMED view HotRooms_1 as
select RoomID, Temperature
from ℜ(RoomTempStr) R
where R.Temperature > 80
```

Then, the two queries $Q_3$ and $Q_4$ can be re-written in terms of $\text{HotRooms}_1$ as follows:

$Q_3$:
```
select STREAMED RoomID, Temperature
from ℜ(HotRooms_1) R
```

$Q_4$:
```
select STREAMED RoomID, Temperature
from ℜ(HotRooms_1) R
where R.Temperature > 100
```

**Running example.** Figure 3 shows the execution of $Q_4$ over the output of $\text{HotRooms}_1$. Notice that the output stream from $\text{HorRooms}_1$ is the same output stream from $Q_3$ that is shown in Figure 2. Basically, when the tuple +<a,105>2 arrives at $Q_4$ at time 2, a corresponding tuple +<a,105>2 is produced in the output. In contrast, +<c,95>4 does not result in producing any output tuples since 95 does not qualify $Q_4$'s predicate. Later, u<c,103>6 results in inserting Room "c" in $Q_4$'s answer via +<c,103>6.

## 2.2 Window Queries

In addition to expressing queries over non append-only streams, SyncSQL still can express sliding-window queries over append-only streams. The sliding-window query model is the most widely used window model in the existing streaming literature. A sliding window is defined by two parameters: (1) *range* that specifies the size of the window, and (2) *slide* that specifies the step by which the window moves over the stream.

Windows may be assigned to streams (e.g., [2, 8]) or to operators (e.g., [7, 24]). However, the same relational operator (e.g., join) may have different semantics under the different window usages. For example, if we consider the window-per-operator usage, a window join with window size $w$, joins the input stream tuples that are within at most $w$ time units from each other [7]. On the other hand, if we consider the window-per-stream usage, a binary window join has two different window sizes, $w_1$ and $w_2$, one for each stream [2].

The difference in window semantics makes it difficult for a language that is defined by one window semantics to express queries from the other window semantics. To overcome this difficulty, SyncSQL does not assume specific window semantics. Instead, SyncSQL uses a general window model that can be used to express the various windows.

### 2.2.1 Expressing Window Queries in SyncSQL

In SyncSQL, raw input streams that represent append-only relations are mapped to tagged streams of *insert* operations (e.g., the TempStr stream in Example 1). SyncSQL does not use specific constructs to express sliding windows over the append-only streams. Instead, SyncSQL employs the predicate-window query model [15] 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 will be explained in Section 3.

The predicate-window model is a generalization of the existing window models, since all types of windows (e.g., window-per-stream, window-per-operator) can be expressed as predicate windows. A time-based sliding window over an append-only stream, say S, is expressed as a predicate over $\Re$(S)'s TS attribute. For example, a window join between two streams, $S_1$ and $S_2$, where two tuples are joined only if they are at most 5 time units apart, can be expressed by the following predicate:
$\Re(S_2)$.TS - 5 < $\Re(S_1)$.TS < $\Re(S_2)$.TS+5. The window predicate can be expressed over any attribute in the input stream tuple (ordered or non-ordered). For example, the temperature monitoring query, $Q_3$, is a predicate-window query in which the predicate is defined over the unordered Temperature attribute. Moreover, sliding-window queries in which a separate window is attached to each input stream can be expressed using predicate windows as shown by the following example.

**Example 5** Consider a road-monitoring application in which sensors are distributed to report car identifiers for cars passing through a specified intersection. The input stream S of car identifiers represents an append-only relation. A sliding window over S of size 5 time units is essentially a *view* that, at any time point T, contains the car identifiers that are reported between times T - 5 and T. Such window view is expressed in SyncSQL as follows:

```
create STREAMED view FiveUnitsWindow as
select *
from ℜ(S) R
where Now − 5 < R.TS ≤ Now
```

The view FiveUnitsWindow is refreshed when either $\Re$(S) is modified or Now is changed. $\Re$(S) is modified by the arrival of S tuples where new S tuples produce *insert* operations in the view's output. On the other hand, Now is continuously changing to indicate the current time, and, as a result, *delete* operations are produced in the output to represent expired tuples that fall behind the window boundaries. Notice that even if S consists of only insert operations, FiveUnitsWindow's output stream includes both insert and delete operations. In Section 3.3 we show that the value of Now can be represented as a view that is continuously updated to reflect the current time.

**Example 6** This example demonstrates query composition by using of FiveUnitsWindow as input in another continuous query. Assume the following continuous query from the road monitoring application, $Q_4$: *"Group the input cars by type (e.g., trucks, cars, or buses). Then continuously report the number of cars passed in the last five time units in each group"*. The query $Q_4$ is expressed over FiveUnitsWindow as follows:

```
select STREAMED COUNT(*)
from ℜ(FiveUnitsWindow)
groupby CarType
```

CarCount'output is a stream of *update* operations that represents the *incremental* query answer. An *update* operation is produced for a group, G, only whenever a car enters and/or expires from G. Notice that if the same query is expressed using COMPLETE output, then whenever the query is refreshed, the query issuer sees the non-incremental answer that includes the count of cars in each group independent from whether the group has been changed or not. The non-incremental output of aggregate queries is the approach that is followed by most of the existing systems to evaluate aggregates over data streams (e.g., [7, 20]).

## 3 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 high rates. Usually, a continuous query issuer is interested in having coarser refresh periods for the answer. For example, as we discussed in Section 1, the issuer of the query $Q_2$ may be interested in getting an update of the answer every five minutes independent of the rate of changes in the parking lot state. The coarser refresh period is achieved via special constructs in other query languages, for example, the *slide* parameter in the sliding-window query model [3, 20] and the *for loop* in [8].

In this section, we introduce the synchronization principle as a generalization for 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. In the rest of the paper, we show that the synchronization principle distinguishes SyncSQL by being able to: (1) express queries with arbitrary refresh conditions, and (2) formally reason about the containment relationships among continuous queries with different refresh periods.

## 3.1 Synchronized Relations

We introduce the *synchronization* principle as a means for expressing coarser refresh periods in SyncSQL. The purpose of the synchronization principle is to define specific synchronization time points at which the query answer is refreshed in response to the input stream tuples. Input stream tuples that arrive between two consecutive synchronization points are not propagated immediately to produce query outputs. Instead, the tuples are accumulated and propagated simultaneously at the following synchronization point.

Similar to the *slide* parameter, the synchronization time points are specified independently for each input stream in the query. Each input stream, say S, is mapped to a corresponding *synchronized relation* $\Re_{Sync}$(S) that is modified by the input stream tuples *only* at the time points that are specified by the synchronization stream, Sync. For example, a *slide* parameter of two time units is specified by the synchronization stream $Sync_2$: 0, 2, 4, 6, .... In Section 3.2 we show how to define and construct synchronization streams.

**Definition 2. Synchronized relation.** A synchronized relation $\Re_{Sync}$(S) is a time-varying relation such that $\Re_{Sync}$(S) = R[S(T)] $\forall$ T $\in$ Sync.

**Example 7** This example illustrates the mapping from an input stream, say S, to S's corresponding synchronized relation $\Re_{Sync_2}$(S). We use the same input stream S as in Example 2. Figure 4 shows the synchronized relation $\Re_{Sync_2}$(S), that is modified by the input stream tuples at time points: 2, 4, 6, .... For example, R[S(1)] is

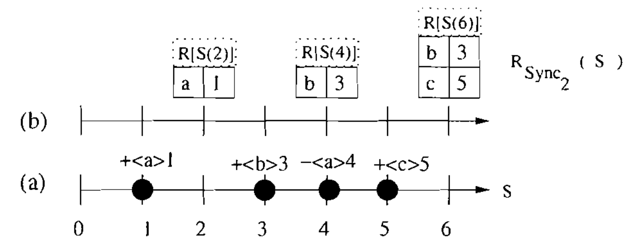


**Figure 4. Illustrating Synchronized Relations.**

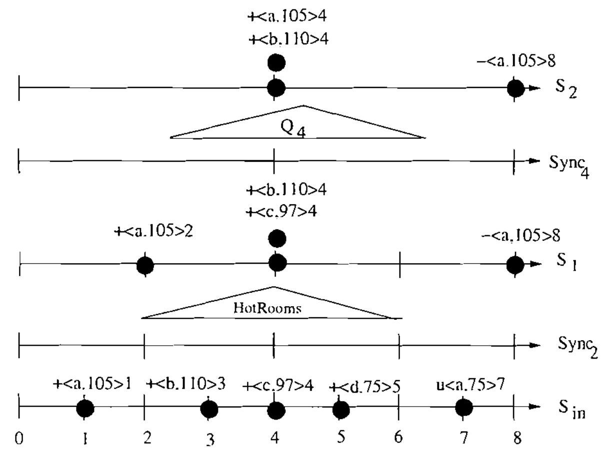


**Figure 5. The Synchronization Principle.**

empty while R[S(2)] reflects the insertion of "a". Moreover, +<b>3 is not reflected in $\Re_{Sync_2}$(S) until time 4.

**Example 8** For the temperature monitoring query $Q_3$, to achieve the coarser refresh (every two minutes) we use the synchronization stream $Sync_2$. Then, the view $HotRooms_1$ is expressed as follows:

```
create STREAMED view HotRooms1 As
select RoomID, Temperature
from ℜSync2(RoomTempStr) R
where R.Temperature > 80
```

Due to the use of $Sync_2$, $\Re_{Sync_2}$(RoomTempStr) is modified every two minutes. As a result, $HotRooms_1$ is refreshed every two minutes as is originally requested by $Q_3$.

**Example 9** Figure 5 shows the execution of $HotRooms_1$ and the subsequent $Q_4$ when using the synchronization principle. For simplicity, we assume that the basic time unit is "minute". Hence, $HotRooms_1$'s answer is refreshed every two time units. Assume that the following input stream $S_{in}$ has arrived at $HotRooms_1$: +<a,105>1, +<b,110>3, +<c,97>4, +<d,75>5, u<a,75>7. In Figure 5, $Sync_2$ represents $HotRooms_1$'s synchronization stream while $S_1$ shows HotRooms$_1$'s output. The input tuple +<a,105>1 that arrived at time 1 results in producing the tuple +<a,105>2 at time 2, which is the first synchronization time point after 1. Similarly, +<b,110>3 results in producing +<b,110>4, and u<a,75>7 results in producing -<a>8.

**Query composition.** $S_1$ is used as input in $Q_4$, which uses the synchronization stream Sync$_4$: 0, 4, 8, .... As a result, tuple +<a,105>2 that arrives at $Q_4$ at time 2 results in producing the tuple +<a,105>4 at time 4 in $S_2$. Other tuples are processed similarly by $Q_4$'s pipeline.

**Timestamps of the output stream tuples.** Timestamps need to be attached to the output tuples from a STREAMED view so that the output stream can be used as input in another continuous query. When considering the synchronization principle, an input tuple possesses two timestamps as follows. (1) The *Arrival* timestamp that is equal to the timestamp attribute of the tuple, and (2) The *Release* timestamp that is equal to the time at which the input tuple is reflected in the query. The *arrival* and *release* timestamps may not be equal for tuples that arrive between two consecutive synchronization points. However, the timestamp of an output tuple is constructed as a function of the *release* timestamp(s) of the input tuple(s) that caused this output because the output necessarily follows the *release* time point. For example, in Example 9, the input tuple +<a,105>2 in $Q_4$, that has arrival timestamp of value 2, has a release timestamp of value 4. As a result, +<a,105>2 results in producing the output tuple +<a,105>4 which has a timestamp equals to 4.

## 3.2 Synchronization Streams

Before proceeding to the algebraic foundations of SyncSQL, this section discuses synchronization streams in more detail. Basically, a synchronization stream specifies a sequence of time points. However, the representation of a synchronization stream follows the tagged stream semantics in Section 2.1, and is treated as any other stream. 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 of the form "+<Timepoint>Timepoint". Like any other stream, a synchronization stream Sync has a corresponding time-varying relation $\Re$(Sync) where each "+<Timepoint>Timepoint" adds a new time point of value Timpepoint to $\Re$(Sync). The default clock stream, clockStr: +<0>0, +<1>1, +<2>2, +<3>3, ..., is the finest granularity synchronization stream where there is a time point for every clock tick. Coarser synchronization streams can be constructed using SyncSQL expressions over clockStr.

**Example 10** The synchronization stream that has a tick every two time points is constructed from clockStr using the following view expression:

create STREAMED view Sync$_2$ As
select C.Timepoint
from $\Re$(clockStr) C
where C.Timepoint mod 2 = 0

A tuple is produced in the output of Sync$_2$ whenever a tuple, c, is inserted in $\Re$(clockStr) and c.Timepoint qualifies the predicate "c.Timepoint mod 2 = 0". The output of Sync$_2$ is as follows: +<0>0, +<2>2, +<4>4, +<6>6, ... which indicates the time points: 0, 2, 4, 6, ..., which is the same as Sync$_2$ that is used in Example 8.

**Composition of synchronization streams.** The fact that synchronization streams are treated as regular streams allows us to compose synchronization streams to define a larger class of synchronization streams. For example, a synchronization stream can be defined as the *union* or *intersection* of two or more streams.

**Example 11** The following view expression produces a synchronization stream that is the union of two input synchronization streams (Note that *duplicate elimination* is required so that every time point exists only once in the output stream):

create STREAMED view UnionSyncStr as
select DISTINCT(Timepoint)
from $\Re$(Sync$_2$) $S_2$ $\cup$ $\Re$(Sync$_5$) $S_5$

The output from UnionSyncStr includes a time point T whenever T belongs to either Sync$_2$ or Sync$_5$.

**Event-based synchronization:** The synchronization principle enables SyncSQL to express a wider class of continuous queries including queries that use event-based refresh conditions. Synchronization streams for event-based conditions can be constructed using SyncSQL expressions as in the following example.

**Example 12** Consider another temperature monitoring query, $Q_5$, that is similar to $Q_4$ except that $Q_5$ needs to be refreshed only whenever a room reports a temperature greater than 120. We use the tagged stream TempStr, which is defined in Example 1, to generate a synchronization stream, say HotSync, such that HotSync includes time points that corresponds to reporting a temperature greater than 120. As explained in Section 2, TempStr consists of only *insert* operations and its corresponding relation $\Re$(TempStr) has three attribute: RoomID, Temperature, and TS. A synchronization stream, HotSync, can then be constructed by the following query over $\Re$(TempStr):

create STREAMED view HotSync as

```
select R.TS
from ℜ(TempStr) R
where R.Temperature > 120
```

An input tuple from TempStr, of the form "+<RoomID,Temperature>Timestamp", results in an output tuple, "+<Timestamp>Timestamp", if "Temperature" is greater than 120. HotSync can be, then, used as a synchronization stream for $Q_5$.

### 3.3 The Now View

In Example 5, FiveUnitsWindow's contents depend on the value of Now. In order to be consistent with the SyncSQL semantics, the value of Now is defined as a view that is continuously modified by the clock stream clockStr: +<0>0, +<1>1, +<2>2, .... Notice that ℜ(clockStr) is an append-only relation in which the value of the last inserted tuple indicates the current time, Now.

**Example 13** The following view, NowView, over ℜ(clockStr) always contains the value of Now:

```
create STREAMED view NowView as
select 1 as KEY, MAX(T.Timepoint) as currTime
from ℜ(clockStr) T
```

The output of NowView is a time-varying relation that has a primary key, KEY. The view always contains one tuple with key value 1, and the tuple is continuously updated in response to insertions in ℜ(*clockStr*). As tuples are appended to ℜ(clockStr), the function MAX(T.Timepoint) selects the last appended tuple that has a value equals to the current time, Now. The output stream from NowView is as follows: +<1,0>0, u<1,1>1, u<1,2>2, u<1,3>3, ..., where the tuple u<1,3>3, for example, means update the record with KEY value 1, to have a currTime value 3. The view FiveUnitsWindow over stream S from Example 5 is rewritten in terms of NowView as follows:

```
create STREAMED view FiveUnitsWindow as
select R.*
from ℜ(S) R, ℜ(NowView) N
where N.currTime − 5 < R.TS ≤ N.currTime
```

**Example 14** This example shows how to use SyncSQL to define a sliding window that is defined by both the *range* and *slide* parameters. Assume we extend the definition of the sliding window in Example 5 such that the window is refreshed every 2 time units instead of every point in time (this corresponds to a sliding window with range 5 units and slide 2 units). In a way similar to using clockStr to define NowView, we use the synchronization stream $Sync_2$ to define a view, say TwoUnitsSlide, as follows:

```
create STREAMED view TwoUnitsSlide as
select 1 as KEY, MAX(T.Timepoint) as currTime
from ℜ(Sync2) T
```

The TwoUnitsSlide view consists of only one tuple that is updated by $Sync_2$'s tuples. The TwoUnitsSlide view can, then, be used to express a sliding window of range 5 and slide 2 over a stream S as follows:

```
create STREAMED view RangeFiveSlideTwo as
select R.*
from ℜ_Sync2(S) R, ℜ(TwoUnitsSlide) N
where N.currTime − 5 < R.TS ≤ N.currTime
```

Only at the time points that belongs to $Sync_2$, RangeFiveSlideTwo's output is refreshed to include S's tuples that arrived in the last 5 time units.

## 4 SyncSQL Algebra

In this section, we lay the algebraic foundation for SyncSQL as the basis for efficient execution and optimization of SyncSQL queries. One of our goals while developing SyncSQL is to minimize the extensions over the well-known relational algebra. By levering the relational algebra, SyncSQL execution and optimization can benefit from rich literature of traditional databases. We achieved our goal by mapping continuous queries to the traditional materialized views. However, the synchronization principle differentiates continuous queries from materialized views. In this section, we introduce the data types and transformation rules that are imposed by the synchronization principle.

### 4.1 Data Types

As discussed in Section 2, although the inputs in a 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** (or state) that is represented by the tuples in the relation and is extracted from the input stream S.
- **Time** that is represented by the time points at which the relation is modified by the underlying stream S and is extracted from the synchronization stream Sync.

The time points at which $\Re_{Sync_i}(S_i)$ reflects **all** $S_i$'s tuples up to time $T_i$ (i.e., $\Re_{Sync_i}(S_i) = R[S_i(T_i)]$) are called "***full** synchronization points*" for the relation. Basically, the time points $T_i \in Sync_i$ represent the full synchronization points for $\Re_{Sync_i}(S_i)$. On the other hand, the time points at which $\Re_{Sync_i}(S_i)$ does not reflect all $S_i$ tuples are called "***partial** synchronization points*". Basically, the time points that lies between two consecutive $Sync_i$ represent the partial synchronization points for $\Re_{Sync_i}(S_i)$.

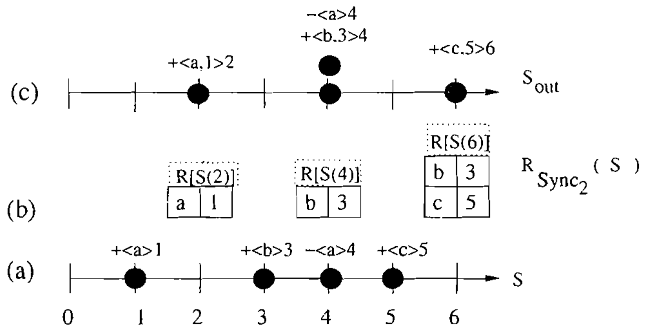


**Figure 6. The Relation-to-Stream Operator.**

## 4.2 Operators

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 [2], but with different instantiations of operators in each class. Basically, the S2R class includes one operator that is used to express the desired synchronization points. The R2R class includes the traditional relational operators. Finally, the R2S class includes one operator that is used in a query to express the desire of an `incremental` output.

### 4.2.1 S2R and R2S Operators

**The stream-to-relation operator $\Re$.** $\Re$ takes a tagged stream of modify operations, say `S`, as input and a synchronization stream, say `Sync`, as a parameter and produces a synchronized relation, $\Re_{Sync}$`(S)`, as output. Similar to $\Re$`(S)`, $\Re_{Sync}$`(S)`'s schema consists of `S`'s underlying schema plus the timestamp attribute `TS` as explained in Section 2.1. Basically, $\Re$ performs the following: (1) buffers `S`'s tuples, (2) modifies the output relation by the buffered tuples at every `Sync`'s point, `T`. The output relation at `Sync`'s point `T` is denoted by `R[S(T)]`.

**The relation-to-stream operator $\xi$.** $\xi$ takes a synchronized relation, $\Re_{Sync}$`(S)`, as input and produces a tagged stream as output. $\xi$ produces output tuples only when the input relation is modified (i.e, at the time points that belongs to `Synch`). Basically, at every `Sync`'s time point, `T`, the input relation is `R[S(T)]` and $\xi$ performs the following: (1) generates delta tuples that represent $\Re_{Sync}$`(S)`'s modifications (i.e., +, u, or -) since the previous synchronization point, (2) assigns `T` as the timestamp of every generated tuple and produces the delta tuples in the output. Notice that non append-only relations can be mapped to streams according to the `SyncSQL` stream semantics.

**Example 15** The functionality of the S2R operator, $\Re$, has been demonstrated before in Example 7. In this example we demonstrate the functionality of the R2S operator, $\xi$. Figure 6 shows the mapping from a synchronized relation, $\Re_{Sync_2}$`(S)`, to the corresponding stream, $S_{out}$ (i.e., $S_{out}$ = $\xi$`(`$\Re_{Sync_2}$`(S))`). Consider the same `S` and $Sync_2$ that are used in Example 7. At time 2, $\Re_{Sync_2}$`(S)` is denoted as `R[S(2)]` and $\xi$ produces `+<a,1>2` in the output. At time 4, $\xi$ produces `-<a>4` and `+<b,3>4` as the differences since the previous synchronization point, `2`. Notice that every `S`'s tuple has a corresponding tuple in $S_{out}$, although because of the synchronization, the corresponding tuples in `S` and $S_{out}$ may not have the same timestamps. Notice also that $S_{out}$'s schema differs from `S`'s schema by having an additional attribute that corresponds to the `Timestamp` field of `S`'s tuples (e.g., the tuple `+<a>1` in `S` is mapped to +<a,1>2 in $S_{out}$). This additional attribute is due to the composition of $\Re$ and $\xi$ operators. Recall that an additional `TS` attribute is added by $\Re$ when `S` is mapped to $\Re_{Sync_2}$`(S)`. As a result, `TS` is produced as an attribute in $S_{out}$'s schema when $\xi$ maps $\Re_{Sync_2}$`(S)` to $S_{out}$. `TS` can be eliminated from $\Re_{Sync_2}$`(S)` by using an R2R project operator, $\pi$.

### 4.2.2 Extended R2R Operators.

The R2R class of operators includes extended versions of the traditional relational operators (e.g., $\sigma$, $\pi$, $\bowtie$, $\cup$, $\cap$, and -). The semantics of R2R operators in `SyncSQL` are the same as in the traditional relational algebra. The difference in `SyncSQL` is that the operators are continuous (not snapshot). A continuous operator means that, inputs to the operator are continuously changing and the operator is continuously running to produce a new output whenever any of the inputs changes.

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., $\sigma$, $\pi$), the output relation is modified at the input relation's synchronization points. In other words, the synchronization points (full and partial) for the output relation are the same as those for the input relation. However, a problem arises in non-unary operators if the input relations have different synchronization points. Notice that operating over relations with different synchronization points is similar to operating over windowed streams with different *slide* parameters (the latter has not been discussed in the existing literature).

For example, consider 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 $T \in (Sync_1 \cup Sync_2)$. The output of `O` is interpreted as follows:

- For every time point

$T_1 \in$ (Sync$_1$-(Sync$_1$ $\cap$ Sync$_2$)), $T_1$ is a *full* synchronization point for $R_{Sync_1}(S_1)$ (i.e., at time $T_1$, $R_{Sync_1}(S_1)$ reflects **all** $S_1$ tuples up to $T_1$). However, the same point $T_1$ is a *partial* synchronization point for $R_{Sync_2}(S_2)$ (i.e., at $T_1$, $R_{Sync_2}(S_2)$ does not reflect all $S_2$ tuples up to $T_1$). Hence, as a result, $T_1$ is a *partial* synchronization point for the output of O because at time $T_1$, the output of O does not reflect **all** input tuples from **all** input streams.

- Similarly, every time point $T_2 \in$ (Sync$_2$-(Sync$_1$ $\cap$ Sync$_2$)) is a *partial* synchronization point for the output of O because it does not reflect all input tuples from all input streams.
- Every time point $T \in$ (Sync$_1$ $\bigcap$ Sync$_2$) is a *full* synchronization point for the output of O since it reflects all input tuples from all input streams.

**Proposition 1. Unary operators.** The output of a unary R2R operator, say $\Theta$, over a synchronized relation, say $\Re_{Sync}(S)$, is another synchronized relation, denoted by $\Theta(\Re_{Sync}(S))$, such that:
$\forall T \in Sync,\ T\ is\ a\ full\ sync\ point,\ and$
$\Theta(\Re_{Sync}(S)) = \Theta(R[S(T)])$ , while
$\forall T \notin Sync,\ T\ is\ a\ partial\ sync\ point,\ and$
$\Theta(\Re_{Sync}(S)) = \Theta(R[S(\tilde{T})])$
$where\ \tilde{T} = max\ (t \in Sync\ and\ t < T)$

**Proposition 2. Binary operators.** The output of a binary R2R operator, say $\Theta$, over two synchronized relations, say $\Re_{Sync_1}(S_1)$ and $\Re_{Sync_2}(S_2)$, is a synchronized relation, denoted by $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$, such that:
(1) $\forall T \in Sync_1 \bigcap Sync_2$,
$T\ is\ a\ full\ sync\ point,\ and$ ,
$\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2) = R[S_1(T)]\ \Theta\ R[S_2(T)]$,
(2) $\forall T \in (Sync_1 - (Sync_1 \bigcap Sync_2))$,
$T\ is\ a\ partial\ sync\ point,\ and$ ,
$\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2) = R[S_1(T)]\ \Theta\ R[S_2(\tilde{T})]$,
$where\ \tilde{T} = max(t \in Sync_2\ and\ t < T)$,
(3) $\forall T \in (Sync_2 - (Sync_1 \bigcap Sync_2))$,
$T\ is\ a\ partial\ sync\ point,\ and$ ,
$\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2) = R[S_1(\tilde{T})]\ \Theta\ R[S_2(T)]$,
$where\ \tilde{T} = max(t \in Sync_1\ and\ t < T)$

According to Proposition 1, at any time point, say $\tilde{T}$, that does not belong to the output synchronization stream, the output synchronized relation from a unary operator reflects the input stream only up to a time point $\tilde{\tilde{T}}$ where $\tilde{\tilde{T}} < \tilde{T}$. Similarly, according to Proposition 2, at any time point, say $\tilde{T}$, that does not belong to the output synchronization stream, the output from a binary R2R operator reflects one input streams up to time point $\tilde{T}$ while reflects the other input stream only up to time $\tilde{\tilde{T}}$ where $\tilde{\tilde{T}} < \tilde{T}$.

**Query pipeline.** In order to express a query over tagged stream, the SyncSQL expression is constructed as follows. (1) S2R: transform each input stream to the corresponding synchronized relation via an $\Re$ operator using the desired synchronization. (2) R2R: using R2R operators, and in a way similar to traditional SQL, express the query over the synchronized relations. The output of is another synchronized relation. (3) R2S: the output synchronized relation is transformed into an incremental output via an $\xi$ operator.

**Example 16** This example shows the execution pipeline for a join query between two synchronized relations, $\Re_{Sync_2}(S_2)$ and $\Re_{Sync_3}(S_3)$, where Sync$_2$ ticks every 2 time units while Sync$_3$ ticks every 3 time units. The SyncSQL expression is as follows:

```
select STREAMED *
from ℜSync2(S2) R2, ℜSync3(S3) R3
where R2.ID = R3.ID
```

Figure 7 illustrates the pipeline and shows that the output of join is refreshed at time points 2, 3, 4, and 6. 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 it reflects $S_2$ up to time 2. 4 also is a *partial* synchronization point since it reflects $S_3$ up to time 3. However, 6 is a ***full*** synchronization point for the output since it reflects **all** input tuples up to time 6.

### 4.3 Equivalences and Relationships

Achieving query composition is one of the main goals of SyncSQL. In order to achieve query composition, a query optimizer must be empowered by algorithms to reason about the equivalences and containment relationships among query expressions. In this section, we introduce preliminary relationships that are required by a query optimizer to enumerate the query plans and deduce query containment.

#### 4.3.1 Containment Relationship among Synchronization Streams

A synchronization stream, say Sync$_1$, is contained in another synchronization stream, say Sync$_2$, if every time point in Sync$_1$ is also a time point in Sync$_2$ (i.e., $\Re$(Sync$_1$) $\subset$ $\Re$(Sync$_2$)). Recall that, as explained in Section 3.2, a synchronization stream consists of only *insert* operations of the form +<Timepoint>Timepoint. Containment relationships between synchronization streams can be deduced from the constructing SyncSQL expressions. For example, a synchronization stream that is defined over clockStr

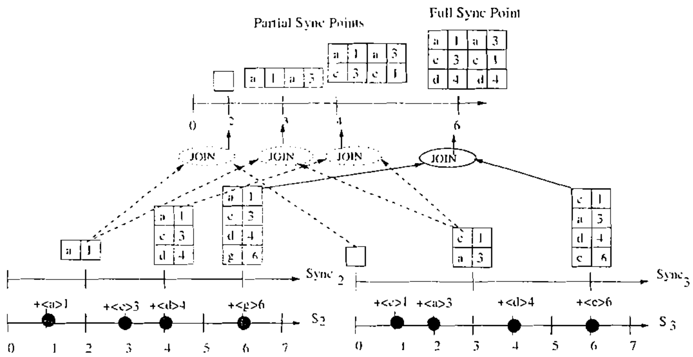


**Figure 7. Joining Synchronized Relations.**

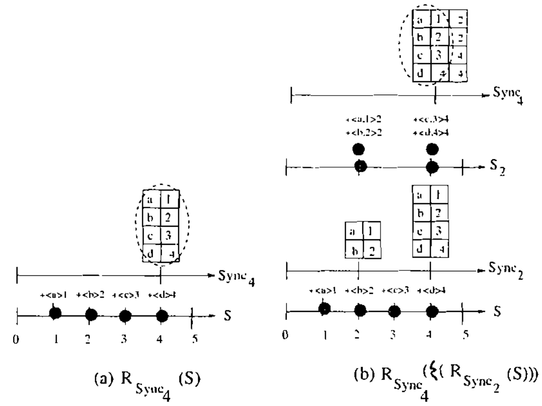


**Figure 8. Relation Containment.**

by the predicate "`Timepoint mod 4 = 0`" (i.e., a stream that ticks every 4 time units) is contained in the synchronization stream that is defined by the predicate "`Timepoint mod 2 = 0`" (i.e., a stream that ticks every two time units).

**Proposition 3.** $\Re(\texttt{Sync}_1) \subseteq \Re(\texttt{Sync}_2)$ if $\forall\ \texttt{I} \in \texttt{Sync}_1 \Rightarrow \texttt{I} \in \texttt{Sync}_2$ where `I` is an insert operation of the form "+<T>T".

#### 4.3.2 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, $\Re_{Sync_i}(\texttt{S})$ and $\Re_{Sync_j}(\texttt{S})$, that are defined over the same stream `S`. Notice that, the *states* of $\Re_{Sync_i}(\texttt{S})$ and $\Re_{Sync_j}(\texttt{S})$ may not be equal at every time point if $\texttt{Sync}_i$ and $\texttt{Sync}_j$ are not the same. However, if $\texttt{Sync}_i$ is contained in $\texttt{Sync}_j$, then $\Re_{Sync_i}(\texttt{S})$ is *contained* in $\Re_{Sync_j}(\texttt{S})$. The containment relationship means that every *full* synchronization time point of $\Re_{Sync_i}(\texttt{S})$ is also a *full* synchronization point of $\Re_{Sync_j}(\texttt{S})$. The containment relationship is beneficial since $\Re_{Sync_i}(\texttt{S})$ can be computed from $\Re_{Sync_j}(\texttt{S})$ without accessing `S`. Notice that, the containment relationship is judged based only on the *full* synchronization time points of the relation because those are the time points of interest to the issuer of a query.

**Theorem 1** *For any stream* `S`*, a synchronized relation* $\Re_{Sync_i}(S)$ *is contained in* $\Re_{Sync_j}(S)$ *if* $\Re(\texttt{Sync}_i) \subseteq \Re(\texttt{Sync}_j)$.

**Proof:**

1. Based on Definition 2:
   $\Re_{Sync_j}(\texttt{S}) = \texttt{R[S(T)]}\ \forall\ \texttt{T} \in \Re(\texttt{Sync}_j)$;
2. Given that $\Re(\texttt{Sync}_i) \subseteq \Re(\texttt{Sync}_j)$, then, based on Proposition 3
   $\forall\ \texttt{T} \in \Re(\texttt{Sync}_i) \Rightarrow \texttt{T} \in \Re(\texttt{Sync}_j)$;
3. From 1 and 2 above,
   $\Re_{Sync_j}(\texttt{S}) = \texttt{R[S(T)]}\ \forall\ \texttt{T} \in \Re(\texttt{Sync}_i)$;
4. Based on Definition 2,
   $\Re_{Sync_i}(\texttt{S}) = \texttt{R[S(T)]}\ \forall\ \texttt{T} \in \Re(\texttt{Sync}_i)$;
5. From 3 and 4 above:
   $\Re_{Sync_i}(S) = \Re_{Sync_j}(S) = \texttt{R[S(T)]}$
   $\forall\ T \in \Re(Sync_i)$.

**Corollary 1.** If $\Re(\texttt{Sync}_i) \subseteq \Re(\texttt{Sync}_j)$, then $\Re_{Sync_i}(S) \subseteq \Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$.

Corollary 1 means that $\Re_{Sync_i}(\texttt{S})$ can be constructed from $\Re_{Sync_j}(\texttt{S})$ without accessing `S`. This is done by applying $\texttt{Sync}_i$ over the output stream from $\xi(\Re_{Sync_j}(\texttt{S}))$.

**Example 17** This example illustrates Theorem 1 and Corollary 1. Consider two synchronization streams, $\texttt{Sync}_2$ and $\texttt{Sync}_4$, where $\Re(\texttt{Sync}_4) \subset \Re(\texttt{Sync}_2)$. Figure 8a gives the derivation of $\Re_{Sync_4}(\texttt{S})$ while Figure 8b gives the derivation of $\Re_{Sync_4}(\xi(\Re_{Sync_2}(\texttt{S})))$. Notice that, all the *full* synchronization points for $\Re_{Sync_4}(\texttt{S})$ are also *full* synchronization points for $\Re_{Sync_2}(\texttt{S})$. Moreover, if only the `STREAMED` version of $\Re_{Sync_2}(\texttt{S})$ is available (i.e., $\xi(\Re_{Sync_2}(\texttt{S}))$ or $\texttt{S}_2$ in Figure 8b), $\Re_{Sync_4}(\texttt{S})$ can be computed by applying $\texttt{Sync}_4$ over $\texttt{S}_2$ (i.e., $\Re_{Sync_4}(\texttt{S})$ at time 4 is contained in $\Re_{Sync_4}(\xi(\Re_{Sync_2}(\texttt{S})))$ at time 4).

#### 4.3.3 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 points.

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

**Corollary 3.** For any *binary* R2R operator, say $\Theta$, $\forall$ 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 $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1)\ \Theta\ \Re(S_2)))$.

The main idea of Corollaries 2 and 3 is that we can pull the synchronization streams out of an R2R operator. Basically, an R2R operator can be executed over finest granularity relations and produce a finest granularity output. Then, the desired synchronization is applied over the fine granularity output. Notice that, Corollaries 2 and 3 can also be used in the opposite direction by a query optimizer to push the synchronization inside R2R operators and, hence, reducing the number of operator executions.

Based on Corollaries 2 and 3, a SyncSQL expression can be executed as follows: (1) transform the input streams to the finest granularity synchronized relations, $\Re(S)$, using the finest granularity synchronization stream (i.e., the clock stream), (2) execute the query pipeline over the finest granularity input producing a fine granularity output relation, (3) map the output relation to a stream using $\xi$, and finally (4) transform the output stream to the desired synchronized output using $\Re$.

# 5 Shared Execution using Query Composition

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 re-writes $Q_i$ in terms of $Q_j$ in a way similar to answering queries using views in traditional databases.

## 5.1 Skinning SyncSQL Expressions

To reason about containment of SyncSQL expressions, we isolate the synchronization streams out of the expression. We term the resulting form of the expressions a "*skinned*" form. The skinned form of a SyncSQL expression is an equivalent expression that consists of: (a) a global synchronization stream that specifies the *full* synchronization points of the expression, and (b) a SQL expression over finest granularity relations. Corollaries 2 and 3 are used to transform any SyncSQL expression into the corresponding skinned form.

**Theorem 2** *Any* SyncSQL *expression has an equivalent normal form.*

Theorem 2 is proved using Corollaries 2 and 3.

**Example 18** This example derives the normal 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 Corollary 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 Corollary 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 normal 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) **Time:** $Sync_1 \bigcap Sync_2$.

## 5.2 Query Matching

SyncSQL query matching is similar to "view exploitation" in materialized views [16, 19]. However, SyncSQL queries differ from the traditional materialized views by the notion of synchronization. A matching algorithm for SyncSQL expressions matches the two parts of the skinned forms: the query expression and the global synchronization points.

After introducing the main tools, we now give the high-level steps of the query matching algorithm. The input to the algorithm is a SyncSQL query expression, say Q, and a set of skinned forms for the concurrent queries.

**Algorithm SyncSQL-Expression-Matching:**

1. Using Corollaries 2 and 3, transform Q to the corresponding normal form by constructing the two components: (1) Q's data, $Q^d$, and (2) Q's synchronization, $Q^s$;
2. Match $Q^d$ with data parts of the other input normal forms using a view matching algorithm from the materialized view literature (e.g., [16]). The result of the matching is a normal form (if any) for a matching expression, say $\tilde{Q}$, such that $Q^d \subset \tilde{Q}^d$;
3. If such $\tilde{Q}$ exists, check whether $Q^s \subset \tilde{Q}^s$;
4. If $Q^s \subset \tilde{Q}^s$, then rewrite $Q^d$ in terms of $\tilde{Q}^d$ using the same algorithm used in Step 2 above. The output expression of the re-write operation is denoted as $Q^D$;
5. The input query, Q, is then equivalent to the synchronized relation with: (1) **Data:** $Q^D$, and (2) **Time:** $Q^s$.

Notice that, the query matching algorithm is used to match an input query against a set of already existing views. On the other hand, if we know the whole set of queries in advance, the skinned forms are constructed using the greatest common divisor of all synchronization streams instead of the default clock stream.

**Example 19** This example illustrates the steps performed to match the temperature monitoring query $Q_4$ with the view $\texttt{HotRooms}_1$ as explained in Example 4. Assume that the input expressions are as follows:
$\texttt{HotRooms}_1 = \sigma_{Temp\,>\,80}(\Re_{Sync_2}(RoomTempStr))$
$Q_4 = \sigma_{Temp\,>\,100}(\Re_{Sync_4}(RoomTempStr))$

The corresponding normal forms for the two expressions are as follows:
$\texttt{HotRooms}_1 = \Re_{Sync_2}(\xi(\sigma_{Temp\,>\,80}(\Re(RoomTempStr))))$
$Q_4 = \Re_{Sync_4}(\xi(\sigma_{Temp\,>\,100}(\Re(RoomTempStr))))$

By Comparing the two normal forms we can conclude that: (1) $\Re(\texttt{Sync}_4) \subset \Re(\texttt{Sync}_2)$, and (2) using a view matching algorithm (e.g., [16]) shows that the "$\texttt{Temp} > 100$" $\subset$ "$\texttt{Temp} > 80$". Then, the algorithm concludes that $Q_4 \subset \texttt{HotRooms}_1$. Then, $Q_4$ is re-written as follows:
$Q_4 = \sigma_{Temp\,>\,100}(\Re_{Sync_4}(\xi(\Re(HotRooms_1))))$.

# 6 Incremental Execution Model

Although the goal of this paper is to introduce the SyncSQL semantics for queries over data streams, in this section we briefly outline an execution model for SyncSQL queries. Detailed implementation and optimization techniques is beyond the scope of this paper.

As discussed in Section 2, a SyncSQL query over streams is semantically equivalent to a materialized view over the input streams' relational views. Similar to materialized views, the straightforward way to keep the query answer (or view) consistent with the underlying relations is to re-evaluate the query expression whenever any of the inputs is modified. However, incremental approaches have been proposed to reduce the cost of maintaining the materialized views. In the incremental maintenance of materialized views, instead of re-evaluating the view expression, only the changes in the input relations are processed in order to produce a corresponding set of changes in the output. SyncSQL physical execution plans follows the incremental maintenance approach of materialized views. Basically, at every synchronization time point, a differential operator processes only the modifications in the input relations and produce a corresponding set of modifications in the output.

As discussed in Section 4, inputs and outputs in any R2R operator are synchronized relations. According to SyncSQL algebra, a relational operation (e.g., $\sigma$ or $\bowtie$) over an input stream $S_{in}$ is executed as follows. At every synchronization time point, say $T_1$, $S_{in}$ is mapped to a corresponding relation, $R[S_{in}(T_1)]$. Then, the relational operation, say $\sigma$, is executed over $R[S_{in}(T_1)]$ and produce a corresponding output relation, say $R[S_{out}(T_1)]$. When the input relation is modified at a following synchronization point, say $T_2$, $\sigma$ is re-executed over $R[S_{in}(T_2)]$ and produce the corresponding output relation $R[S_{out}(T_2)]$. If the output of $\sigma$ is needed to be STREAMED, a $\xi$ operator is executed at time $T_2$ to produce tuples in the output stream $S_{out}$ that represent the deltas between $R[S_{out}(T_1)]$ and $R[S_{out}(T_2)]$. The delta tuples is a set of +, u or - operations that need to be performed over $R[S_{out}(T_1)]$ in order to get $R[S_{out}(T_2)]$. In short, SyncSQL algebra assumes that an R2R operator is *re-executed* at every synchronization time point.

In contrast to the algebra, SyncSQL physical execution plans employs an incremental approach. At every synchronization time point, an incremental relational operator processes only the modifications in the input relations and produce a corresponding set of modifications in the output relation. For example, at a synchronization time point, $T_2$, the incremental $\sigma$ operator processes a set of delta tuples between $R[S_{in}(T_2)]$ and $R[S_{in}(T_1)]$ and produce another set of delta tuples between $R[S_{out}(T_2)]$ and $R[S_{out}(T_1)]$.

## 6.1 Derived Operators

**The S2S counterparts of R2R operators.** A SyncSQL execution plan consist of a set of S2S operators where each R2R operator (e.g., $\sigma$ and $\bowtie$) has a corresponding incremental (or differential) S2S operator (e.g., $\sigma^d$ and $\bowtie^d$). Basically, the functionality of an S2S operator is composed of three functions (S2R, R2R, then R2S) as follows: (1) S2R: takes an input modification tuple (i.e., +, u, or -) and apply the modification to the operator's internal state. (2) R2R: perform the relational operator's function over the operator's internal state. (3) R2S: report the modifications in the internal state as an output tagged stream. Detailed implementation of S2S operators is addressed in [14].

The relationship between the input and output tagged streams from an S2S operator is defined algebraically by differential equations [17]. The functionality of a differential operator, say $\theta$, is defined by two equations: one equation defines the modifications in $\theta$'s output in response to an *insert* in $\theta$'s input while the other equation defines the changes in $\theta$'s output in response to a *delete* in $\theta$'s input. An *update* in $\theta$'s input is processed as a *deletion* of the old tuple followed by an *insertion* of the new tuple. For example, the functionality of the differential $\sigma$ is defined by the following equations:
$\sigma_p(R + r) = \sigma_p(R) + \sigma_p(r)$

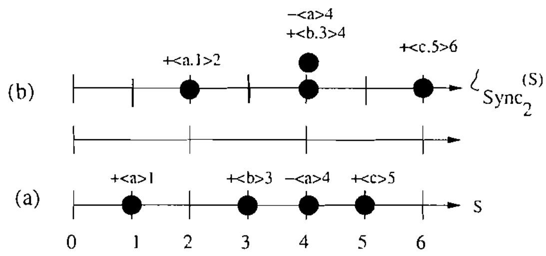


**Figure 9. The Regulator, $\zeta$, Operator.**

$\sigma_p(R - r) = \sigma_p(R) - \sigma_p(r)$
where +r (-r) represents the insertion (deletion) of a tuple r into (from) $\sigma$'s input relation R, while $+\sigma_p$(r) ($-\sigma_p$(r)) represents the corresponding insertion (deletion) into $\sigma$'s output relation, $\sigma_p$(R). Algebra for the various differential operators is introduced in [17].

**The S2S counterpart of the S2R operator.** In order to apply the synchronization principle with S2S operators, we introduce the regulator operator, $\zeta$ as the S2S counterpart of $\Re$. Similar to $\Re$, $\zeta$ takes a stream, $S_{in}$, as input, a synchronization stream, Sync, as a parameter and produces another stream, $S_{out}$, as output where

$$S_{out} = \zeta_{Sync}(S_{in}) = \xi(\Re_{Sync}(S_{in})).$$

Notice that, as discussed in Section 4.2 and Example 15, the schema of the resulting stream from $\xi(\Re_{Sync}(S_{in}))$ differs from $S_{in}$'s underlying schema by having an additional timestamp attribute that corresponds to the *arrival* timestamp of $S_{in}$'s tuples. The additional timestamp attribute is used to evaluate time-based predicates (if any) over $S_{in}$ and is also included in the output stream, $S_{out}$, from $\zeta$. Basically, $\zeta$ works as follows: buffers the input stream tuples and at every synchronization time point, say T, $\zeta$ performs the following for each buffered input tuple of the form "Type<Attributes>Timestamp": (1) constructs a corresponding tuple of the form "Type<Attributes,Timestamp>", by pushing the arrival timestamp, Timestamp, inside the tuple's schema. , and (2) assigns a timestamp to the tuple that is equal to the release time, or T. As a result, $\zeta$'s output tuples will have the form "Type<Attributes,Timestamp>syncTimestamp".

**Handling timestamps by the physical operators.** An output tuple from $\zeta$ has two timestamps as follows: (1) Timestamp that is equal to the tuple's *arrival* timestamp and is used by the subsequent R2R differential operators to evaluate time-based predicates (if any), and (2) sycnTimestamp that is equal to the tuple's *release* timestamp and is propagated by the subsequent R2R operators to the corresponding output tuples.

**Example 20** This example shows the functionality of the regulator operator, $\zeta$. Consider the same S and $Sync_2$ as those used in Example 7. Figure 9 shows S and the corresponding $\zeta_{Sync_2}$(S). $\zeta$ transforms, for example, +<a>1 into +<a,1>2 by pushing the arrival timestamp of value 1 into the schema and attaching the release timestamp of value 2 as the timestamp of the output tuple. Figures 6 and 9 show that $\zeta_{Sync_2}$(S) = $\xi(\Re_{Sync_2}$(S)).

## 7 Related Work

**Continuous queries over data streams.** Many research efforts have developed semantics and query languages for continuous queries over data streams, e.g., [2, 6, 7, 8, 11, 24]. The existing continuous query languages restrict the stream definition to the representation of an append-only relation. The restricted 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 non-append only output. Different approaches have been followed by the existing languages to handle the non append-only *outputs* as follows:

**-Restricted expressibility:** To guarantee that the output of the query can be incrementally produced as a stream, a language restricts the set of operators that can be used to express queries over data streams. The restricted set of operators includes, for example, Select, Project, and Union. Sliding windows with the window-per-stream usage, for example, are not allowed since they produce non-append only output. Examples of systems that follow this approach include Aurora [7], Cougar [6], and Gigascope [11].

**-Non-incremental output streams:** Produce the output of the query in a *non-incremental* manner by representing the output as a relation then periodically stream out the relation. Notice that this non-incremental output stream does not follow the input stream definition and, hence, cannot be used as input in another query. Examples of systems that follow this approach include TelegraphCQ [8], and the RStream operator in CQL [2].

**-Non-incremental output relations:** Does not allow queries that produce non append-only output to produce streams. Instead, such queries produce concrete views as outputs. Moreover, only *snapshot* queries are allowed over the view. A snapshot query has to be re-issued in order to know the modifications in the view. This approach is followed by ESL [24].

**-Divided output:** CQL [2] divides the query into two separate queries that produce append-only streams such that one query produces a stream, IStream, to represent the inserted tuples and the other query produces a stream, DStream, to represent the deleted tuples . It is unclear how to compose the two streams in order to produce a single output stream that can be used as input in another query.

SyncSQL semantics avoids these previous limitations by allowing the output of any continuous query to be produced incrementally in a single stream.

There are two SQL-based languages that are closest to SyncSQL: CQL [2] and ESL [24]. SyncSQL uses the same three classes of operators (i.e., S2R, R2R, and R2S) as that of CQL but use a different instantiation of operators in each class. CQL defines two types of sliding windows (time-based and tuple-based) and defines the window as an S2R operator. However, there are no algebraic or transformation rules to show how the window operator interacts with the other (R2R) operators in the pipeline. Moreover, semantics of non-unary operators on two streams with different *slide* parameters is not discussed. ESL [24] is another SQL-based continuous query language that is designed mainly for data mining and time-series queries. Only unary operators (e.g., selection and projection) can be used in queries to produce output streams. On the other hand, since a window function produces a non append-only output, window queries produce concrete views as output. Streams can be joined with the concrete views, but in this case, the modifications in the view do not affect the already produced stream tuples but they affect only the incoming stream tuples. ESL focuses on aggregate queries but does not thoroughly address set-based operators and queries.

**Positive and negative tuples.** Streams of positive and negative tuples (i.e., insert and delete tuples) are frequently used when addressing continuous query processing [1, 5, 13, 14]. 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. SyncSQL overcomes this obstacle by unifying the stream definition between the language and the execution model.

**Continuous queries in traditional databases.** Continuous queries are used in traditional databases before being used over data streams. Examples of systems that support continuous queries over database tables include Tapestry [23] and OpenCQ [21]. In these systems, both inputs and outputs of the continuous query are relations. Although the input relations in Tapestry are append-only, queries may produce non append-only output if the query includes either a reference to the current time (e.g., GetDate()), or a set-difference between two relations. In order to guarantee the append-only output,Tapestry uses a query transformation to transform a given query into the minimum bounding append-only query. The coarser refresh of the query is achieved via a "*FOREVER DO, SLEEP*" clause where the query is re-executed after every *SLEEP* period. On the other hand, in OpenCQ, input and output relations can be modified by general modify operations. A continuous query is periodically re-executed and the output is produced as the delta between two consecutive query executions. Triggers are used to schedule the query re-execution. Our notion of synchronization time points is similar to OpenCQ's Triggers, but synchronization streams are distinguished by the fact that they can be generated using regular queries. Unlike Tapestry and OpenCQ, SyncSQL assumes that query inputs and outputs are streams and hence requires special handling of the timestamps. Moreover, we introduce an algebraic framework and address composition of SyncSQL expressions, which is not addressed by the previous systems.

**Shared query execution.** A typical streaming environment has a large number of concurrent 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 the execution at the operator level. Shared aggregates are addressed in [4] where an aggregate operator is shared among multiple queries with different window *ranges*. Shared window join is addressed in [18]. NiagraCQ [10] proposes a framework for shared execution of non-windowed SPJ queries. Shared predicate indexing is used in [9, 10] to enhance the performance of a continuous query processor. Our approach for shared execution is distinguished from the existing approaches in that: (1) based on query composition; (2) matches window queries that differ in both the *range* and *slide* parameters, and (3) queries are examined for sharing based on a whole query expression not only at the operator level.

**Materialized views:** Our definitions of synchronized relations and predicate-windows enable us to benefit from the existing literature in materialized view. However, we extend the materialized view algorithms to work with synchronized relations. Our query matching algorithm extends the traditional view exploitation algorithms (e.g., [16]) by matching the synchronization time points in addition to matching the query expression. Moreover, the physical design of SyncSQL execution pipelines follows the incremental maintenance of materialized views [17].

# 8 Concluding Remarks

This paper provides the first language, SyncSQL, to express continuous queries over streams of modify operations. Modify streams are general since they can represent both raw input streams and streams that are generated as output from executing continuous queries. The unified definition of query inputs and outputs enables the composition of SyncSQL expressions. The paper provides the first shared execution algorithm for continuous queries that is based on query composition. Shared execution decisions are based on a query matching algorithm that is able to reason about the equivalence and containment relationships among SyncSQL expressions. Efficient execution of SyncSQL queries is an important issue. We outlined an execution model to incrementally evaluate a SyncSQL query.

Detailed implementation and optimization techniques will be reported in a separate paper.

## References


[1] D. Abadi. et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] A. Arasu. S. Babu. and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDBJ*. to appear.

[3] A. Arasu and J. Widom. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Record*, 33(3):6–12, 2004.

[4] A. Arasu and J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*. 2004.

[5] S. Babu. K. Munagala. J. Widom. and R. Motwani. Adaptive Caching for Continuous Queries. In *ICDE*. 2005.

[6] P. Bonnet. J. E. Gehrke. and P. Seshadri. Towards Sensor Database Systems. In *MDM*, 2001.

[7] D. Carney. et al. Monitoring Streams - A New Class of Data Management Applications. In *VLDB*. 2002.

[8] S. Chandrasekaran. et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*. 2003.

[9] S. Chandrasekaran and M. J. Franklin. PSoup: A System for Streaming Queries over Streaming Data. *VLDBJ*. 12(2):140–156. 2003.

[10] J. Chen, D. J. DeWitt, F. Tian. and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*. 2000.

[11] C. D. Cranor. et. al. Gigascope: A Stream Database for Network Applications. In *SIGMOD*. 2003.

[12] A. Eisenberg. et. al.. SQL:2003 Has Been Published. *SIGMOD Record*, 33(1):119–126. 2004.

[13] S. Ganguly. et. al. Processing Set Expressions over Continuous Update Streams. In *SIGMOD*. 2003.

[14] T. M. Ghanem. et al. Incremental Evaluation of Sliding-Window Queries over Data Streams. In *Purdue University Technical Report, CSD TR 04-040*.

[15] T. M. Ghanem. W. G. Aref. and A. K. Elmagarmid. Exploiting Predicate-Window Semantics over Data Streams. *SIGMOD Record*, 35(1):3–8, 2006.

[16] J. Goldstein and P.-Å. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD*, 2001.

[17] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *SIGMOD*. 1995.

[18] M. A. Hammad. M. J. Franklin, W. G. Aref, and A. E. Elmagarmid. Scheduling for Shared Window Joins over Data Streams. In *VLDB*, 2003.

[19] P.-Å. Larson and H. Z. Yang. Computing Queries from Derived Relations. In *VLDB*, 1985.

[20] J. Li, D. Maier, K. Tufte. V. Papadimos. and P. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams . In *SIGMOD*, 2005.

[21] L. Liu. C. Pu. and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *TKDE*, 11(4):610–628, 1999.

[22] D. Maier. J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of Data Streams and Operators. In *ICDT*. 2005.

[23] D. B. Terry, D. Goldberg. D. Nichols. and B. M. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*. 1992.

[24] WEB Information System Laboratory. UCLA, CS Department. An introduction to the Expressive Stream Language (ESL). http://wis.cs.ucla.edu/stream-mill.