A Semantics for a Query Language over Sensors, Streams and Relations

Christian Y.A. Brenninkmeijer, Ixent Galpin,
Alvaro A.A. Fernandes, Norman W. Paton

School of Computer Science, University of Manchester,
Manchester M13 9PL, United Kingdom
{brenninkmeijer,ixent,alvaro,norm}@cs.man.ac.uk

Abstract. We introduce a query language over sensors, streams and relations and formally describe its semantics. Although the language was specifically designed for sensor network querying, where data is pulled into streams, the semantics contributed in the paper also encompasses the case in which data is pushed onto streams or else lies stored in classical relations. The approach taken is that continuous queries over streams are an extension of classical queries over stored extents. Apart from the fact that query evaluation over streams is reactive, or periodic, the main difference is the conception of windows as an additional collection type with the consequent use of type converter operations to and from streams and windows (which, as bounded collections of tuples, can be operated on in a relational-algebraic setting). The language and the semantics we provide for it advance on previous work in being more comprehensive with respect to the collection types allowed and in being more flexible as to the number and content of the windows contributing to the result at each evaluation event of a continuous query. The formalization advances on previous work in clarifying the implementation onus.

Key words: Stream/Sensor Network Data, Query Language Semantics

1 Introduction

Data streams [4, 9] have become an important information resource in both commercial and scientific contexts. In the last ten years, many query languages and stream data management systems have been designed and implemented [1, 2, 5–7, 13]. This burst of activity stems, at least in part, from the fact that certain characteristics of data streams challenge some foundational assumptions underpinning classical database management systems. Among the many issues raised in [4], this paper focusses on the issue of assigning a semantics to continuous queries over extents with unbounded size in the presence of blocking operators. One of the issues arising is that blocking operations such as cross-product (and hence, in general, joins) and group-by aggregation are not well-defined over unbounded extents.

While systems and languages that constrain themselves to operating on unbounded extents exist (e.g., [7]), most rely on one or more mechanisms to cope with the unbounded cardinality of data streams. Broadly, one may resort to data

reduction techniques (e.g., filtering, data merging and data dropping, synopses and sketches) or one may characterize bounded subsets of the stream that one can operate upon using either punctuation (thereby relying on stream semantics) or sliding windows. A variation on the latter relies on materializing bounded subsets of the stream as views. A survey of most of these techniques is given in [12]. We focus on windows as the mechanism to cope with unbounded cardinality.

The remainder of the paper is structured as follows. Sec. 2 briefly describes the background for the paper, its motivation and the contributions reported, which centre on a query language over sensors, streams and relations called SNEEql. Sec. 3 describes the SNEEql data model; Sec. 4, its syntax; Sec. 5, its translation to a logical algebra; and Sec. 6, the formal definition of the algebraic operators. Sec. 7 draws contrasts with related work, Sec. 8 concludes the paper.

2 Background, Motivation, Contributions

Two of the questions raised in [4] as providing directions for future work implicitly touch on the issue of the semantic relationship between streams and classical relations as they impact on the data modelling and query language traditions. Semantic issues have been explored informally by most system-description papers [1, 2, 5–7, 13], but formal accounts [1, 3, 11] are comparatively fewer, often not exhaustive, and not as informative as could be wished by an implementer.

We explore the relationship of window-based continuous query semantics over streams and relations. In particular, our treatment encompasses push-based streams (which have been predominant in the literature so far) and pull-based streams (as arise in sensor network query processing [10, 14]). Our treatment assigns a semantics to queries over unbounded streams and over bounded subsets of unbounded streams. While this paper does not provide a detailed account of it, the semantics for continuous queries we describe can accommodate classical queries as the special case of one-off queries over stored extents only. In this way, the paper contributes a wide-ranging account that is distinctive in encompassing (1) streams and relations, (2) push- and pull-based streams, and (3) blocking and non-blocking operators, and that clarifies the relationship with classical relational-algebraic semantics. Our approach is inspired by CQL [2] in that we also view windows as a collection type resulting from a conversion operation that maps from unbounded extents (i.e., streams) to bounded ones (i.e., relations, as bags of tuples), thereby allowing both non-blocking and blocking operators to be supported. The account given here advances on previous work in providing a formal, unified account for more expressive queries than previously done. Thus, the reactive, or periodic, nature of result production is explained in terms of tuples that either simply arrive in push-based streams, or are acquired from sensors, or are scanned from stored tables. If windows are used, slide events may be triggered by new inputs and evaluation may produce new results.

Secs. 3-6 describe (briefly, due to the space constraints) the underlying data model used in SNEEql and the syntax and semantics of the language. Discussion of related work then follows.

3 SNEEql Data Model

The *primitive types* are `integer`, `float`, `string` and `time`. The *compound types* are `tuple` and `tagged tuple`. A `tuple` type consists of a set of typed attributes, $a_1 : t_1, \ldots, a_n : t_n$, where each a_i is an attribute name and each t_i is a primitive type. A `tagged tuple` type is a `tuple` type including two distinguished attributes: one, named `tick`, of type `integer`, and another, named `index`, of type `integer`. Values of type `tick` are drawn from a system-wide ordinal domain, those of type `index` are ordered inside the collection in which they appear. A `tick` value denotes the *timestamp* in which a tagged tuple was created, an `index` value denotes its *position* in a sequence where it was placed. The *collection types* are `window` and `stream`. A `window` type is a pair whose first element is a distinguished attribute, named `tick`, of type `integer`, denoting the *timestamp* in which the window was created, and whose second element is of type bag of tuples of the same `tuple` type. A `stream` is a potentially infinite, append-only sequence of values of the same `tagged tuple` or `window` type.

As an example SNEEql schema, consider a system with access to (1) road sensors that detect temperature and vibration levels every minute at four sites (named 1,2,3, and 4); (2) a push stream that reports, at a frequency of its choosing, the weight and class of passing traffic; and (3) a table of temperature classes. The schema could be expressed as follows:

```
road: sensed (site:integer, time:integer, temp:integer, vibration:integer)
traffic: pushed (site:integer, time:integer, weight:integer, vehicle:integer)
tempClass: stored (temp:integer, category:string)
```

Sensed extents are pull-based, i.e., associated with a declared acquisition rate (one tuple per minute per site, in this example), and for this reason can also be referred to as *acquisitional*. Streamed extents are push-based, i.e., associated with an unknown, presumed variable, arrival rate. From the viewpoint of continuous SNEEql queries, both sensed and pushed extents, like road and traffic, are streams of tagged tuples, whereas stored extents, like tempClass, are streams of windows, as described further below. Note that `tick` and `index` are implicitly-defined attributes of tagged tuples, as is `tick` for windows.

4 SNEEql Syntax

This section introduces the kinds of SNEEql queries whose semantics is described in later sections, viz., stream queries and windows queries. **Stream queries** are of the form `SELECT` $a_1 \ldots a_n$ `FROM` s `WHERE` p where $a_1 \ldots a_n$ is a project list, s denotes a stream of tagged tuples (i.e., either the name of a sensed or pushed extent, or a subquery of type `stream`), and p is a predicate. Restrictions on stream queries that are relaxed for window queries (described below) include: the `FROM` clause must reference a single stream because cross product is not well defined over infinite collections; and the projection list elements a_i cannot apply aggregate operations over values from s. Evaluating a stream query yields a stream of tagged tuples. Let **Q1** be the following acquisitional stream query:

```
SELECT road.time, road.site, road.vibration
FROM   road
WHERE  road.temp < 50 AND road.vibration > 20
```

Window queries are of the form:

SELECT $a_1 \ldots a_n$ FROM $w_1 \ldots w_m$ WHERE p

where $a_1 \ldots a_n$ is a project list, $w_1 \ldots w_m$ is a list of window definitions, and p is a predicate. Window queries can be extended with GROUP BY and HAVING clauses: these are not described here due to space constraints. Evaluating a window query yields a stream of windows. Each w_i in the FROM clause refers to either a streamed (sensed or pushed) or a stored extent, as follows.

A window on a stream is of the form s[FROM t_1 TO t_2 SLIDE int $unit$], where s denotes a stream of tagged tuples (i.e., either the name of a sensed or pushed extent, or a subquery of type stream), and both t_i are either of the form NOW or NOW $- int$, where NOW denotes the current tick or index, int is a positive integer, and $unit \in \{\texttt{SEC}, \texttt{MIN}, \texttt{HOUR}, \texttt{ROWS}\}$. The FROM and TO parameters define a window that selects all tuples in s in the range relative to when the window is created, while the SLIDE parameter determines how often a new window is created.

A window on a table is of the form t[SCAN int $timeUnit$], where t is a table, int is a positive integer, and $timeUnit \in \{\texttt{SEC}, \texttt{MIN}, \texttt{HOUR}\}$. The SCAN parameter indicates how often the table t is scanned and a window created that contains the result of the scan.

Let **Q2** be a window query in SNEEql that requests the minimum temperature and maximum vibration from the road sensors over the last 10 minutes, but only for sites and times where the traffic stream reported a vehicle passing. **Q2** returns a stream of windows and can be written as follows:

```
SELECT   MIN(temp), MAX(vibration)
FROM     road      [FROM NOW-10 TO NOW SLIDE 1 MIN],
         traffic   [FROM NOW-10 TO NOW SLIDE 1 ROW]
WHERE    road.site=traffic.site AND road.time=traffic.time
```

The result of a SNEEql window query can be converted into a stream using the CQL [2] type-conversion functions RSTREAM, ISTREAM and DSTREAM (see Section 6.3). The next query, **Q3**, shows how SNEEql allows data acquired from sensors to be combined with stored data scanned from a table. **Q3** returns a stream of tuples and can be written as follows:

```
ISTREAM( SELECT    road.site, tempClass.category
         FROM      road       [FROM NOW-5 TO NOW SLIDE 5 MIN],
                   tempClass  [SCAN 10 MIN]
         WHERE     tempClass.temp = road.temp)
```

5 SNEEql Translation to a Logical Algebra

The SNEEql semantics contributed in this paper is defined in terms of a mapping to a logical algebra, whose operators are defined using Haskell[1]. This section both introduces the algebra and describes the mapping from SNEEql. Stream queries are mapped to the algebra as follows:

[1] In the Haskell notations used, lower case is used to name variables and functions; upper case is used to name types and constructors.

```
SELECT a1 ... an FROM s WHERE p
⇒  evaluateStream (StreamProject [a1 ... an] (
      StreamSelect (p) (
        StreamAcquire (s, s.acquisitionInterval, s.sites) )))
```

where, given an instance of an operator on streams, `evaluateStream` returns the stream that results from evaluating that operator. The operators are further described in Sec. 6, but their arguments are now briefly explained. `StreamProject` takes a list of projection expressions and the stream from which their values are obtained. `StreamSelect` takes a predicate expression and the stream over which it is applied as a filter. `StreamAcquire` takes the name, the acquisition interval and the sensing sites from which the desired stream of data is obtained. For example, **Q1** from Sec. 4 maps to the following algebraic expression:

```
evaluateStream (
  StreamProject [Attr "road.time", Attr "road.site", Attr "road.vibration"] (
    StreamSelect (Predicate "road.temp>50 and road.vibration>20") (
      StreamAcquire "road" (Tick 60) [1,2,3,4])))
```

Window queries are mapped to the algebra as follows:

```
SELECT a1 ... an FROM w1 ... wm WHERE p
⇒  evaluateWindow (WindowProject [a1 ... an] (
      WindowSelect (p) (
        WindowCrossProduct (w1), ..., (wm) )))
```

where, given an instance of an operator on streams of windows, `evaluateWindow` returns the stream of windows that results from evaluating that operator. The operators are further described in Sec. 6, but their arguments are now briefly explained. `WindowProject` takes a list of projection expressions and the stream from which their values are obtained. `WindowSelect` takes a predicate expression and the stream over which it is applied as a filter. `WindowCrossProduct` takes the operators that yield the windows to which the cross product is applied. The translation of a window operator, whose argument is a window definition, depends on the form of the latter. For example, a time window definition over an acquisitional stream is mapped to the algebra as follows:

```
s[FROM t1 TO t2 SLIDE int timeUnit]
⇒  evaluateWindow (TimeWindow ( TimeScope(t1.offset,t2.offset), slide) (
      evaluateStream (StreamAcquire (s, s.acquisitionInterval, s.sites))))
```

where `TimeWindow` takes two offsets relative to `NOW`, which define the endpoints of the window on s; the slide value, which indicates how frequently windows are created; and the stream from which tuples are obtained for creating the window. For example, **Q3** from Sec. 4 maps to the following algebraic expression:

```
evaluateStream(IStream (
  WindowProject [Attr "road.site", Attr "tempClass.category"] (
      WindowSelect (Predicate "tempClass.temp=road.temp") (
        WindowCrossProduct (
          TimeWindow (TimeScope ((Tick (-300)) (Tick 0)) (Tick 300)) (
            StreamAcquire "road" (Tick 60) [1,2,3,4])) (
          Scan (Tick 600) "tempClass")))))
```

where most of the operators have been introduced except `IStream`, which returns an output stream of tagged tuples containing the tuples most recently added to its input; and `Scan`, which, given the name of a stored extent and a time interval, returns the stream of windows that results from scanning the given table with a frequency governed by the scanning interval.

6 Semantics of the SNEEql Logical Algebra

The semantics of the algebra introduced in Sec. 5 is defined in this section using Haskell. Because Haskell is a lazy, pure functional programming language, the computation of a value is deferred until it is needed. This allows us to represent streams as unbounded lists and operators as functions which take as input and return unbounded lists. The Haskell definitions of the operators aim at clarity, not space- and time-efficiency. Note that even though SNEEql was designed to run over sensor networks as a distributed query evaluation system [8], the semantics does not take distributed execution into account because, in practice, in-network query evaluation is carried out through the translation of the logical algebra into a parallel algebra, in which the semantics of operators in the logical algebra is preserved. The semantics is organized as follows: Sec. 6.1 defines operators that return streams of tuples, and which together support **stream queries**; Sec. 6.2 defines operators that return streams of windows, and which together with the operators from Sec. 6.1 support **window queries**; and Sec. 6.3 describes operators for converting from windows to tuple streams.

In the Haskell notations, a declaration `e :: t` asserts the expression `e` to be of type `t`. For example, the increment function may be declared as `inc :: Integer -> Integer`. The expression `h:t` denotes the list with head `h` and tail `t`. List concatenation is denoted by `++`, list difference by `\\`. The expression `e == ee` is true iff `e` and `ee` are equal. The expression `id = e` binds the expression `e` to the name `id`. If `h = 1` and `t = 2:3:[]`, then `h:t == [1,2,3]` is true. Local definitions use `let e in ee` blocks. Given a function f and a list L, `map` returns the list that results from applying f to each $l \in L$. Given a function f from type a to type `Bool` and a list L of elements of type a, `filter` returns $l \in L$ such that f is true of l.

6.1 Tuple Stream Operators

A tuple stream is represented as a lazily evaluated list of tagged tuples. Operations that return tuple streams fall into the following groups: *input operators*, which obtain data from sensors and from pushed streams; *filtering operators*, which select or project tuples, and *conversion operators*, which generate a tuple stream from a stream of windows (and whose definition is postponed to Sec. 6.3). The definition of `evaluateStream` is in Fig. 1.

Input The leaf operators in tuple stream queries either acquire data from sensors or receive data from pushed streams. `StreamAcquire` is defined using `acquire` (in Fig. 2). Given the name of a sensed extent, the acquisition interval, the list of associated sites, the tick at which to take a reading and the index of the next value to be read, `acquire` returns a potentially infinite list of tagged tuples. The function `lookupAttributes` returns the list of attribute names for

```
evaluateStream  :: StreamOp->[TaggedTuple]
evaluateStream  (StreamAcquire sourceName tick sites) =
  acquire sourceName tick sites (Tick 0) (Index 1)
evaluateStream  (StreamReceive source) = receive source
evaluateStream  (StreamSelect predicate streamOp) =
  filter (predicateOnTaggedTuple predicate) (evaluateStream streamOp)
evaluateStream  (StreamProject projectList streamOp) =
  map (projectOnTaggedTuple projectList) (evaluateStream streamOp)
evaluateStream (RStream windowOp) =
  rStream (Index 1) (evaluateWindow windowOp)
evaluateStream  (IStream windowOp) =
  iStream (Index 1) [] (evaluateWindow windowOp)
evaluateStream  (DStream windowOp) =
  dStream (Index 1) [] (evaluateWindow windowOp)
```

Fig. 1. Definition of `evaluateStream`

```
acquire::String->Tick->[Int]->Tick->Index->[TaggedTuple]
acquire sourceName acquisitionInterval sites now index =
        let attributeNames = lookupAttributes sourceName
  in let tuples = acquireTuples attributeNames now index sites
  in let nextTick = (now + acquisitionInterval)
  in let nextIndex = index + Index (length sites)
  in tuples ++ acquire sourceName
                 acquisourceNamerval sites nextTick nextIndex

acquireTuples::[AttributeName]->Tick->Index->[Int]->[TaggedTuple]
acquireTuples _ _ _ [] = []
acquireTuples attributeNames tick index (site:sites) =
  let tupleData = map (getData tick site) attributeNames
  in [TaggedTuple tick index (Tuple attributeNames tupleData)]
    ++ acquireTuples attributeNames tick (inc index) (sites)
```

Fig. 2. Definition of `acquire`

a sensed extent. The function `getData` abstracts away from low-level calls to physical sensors. In Haskell, this can be simulated by generating readings of individual tuples at the specified points in time. `StreamReceive` is defined using `receive::String -> [TaggedTuple]`. Given the name of a pushed extent, `receive` returns a potentially infinite list of tagged tuples. Unless a tuple arrives timestamped, it is tagged with the current tick at its arrival, and assigned an index. The function abstracts away from a port onto which an external source can write. In Haskell, this can be simulated by generating a variable number of tuples at variable intervals. The semantics of other operators accommodates multiple tuples with the same timestamp, as well as timestamps for which there are no tuples.

Filtering Filtering operators are applied to each tuple independently. `StreamProject` is defined using `map` (see Fig. 1) to apply `projectOnTaggedTuple` (in Fig. 3) to each tuple in the stream. Given an attribute list and a tagged tuple, `projectOnTaggedTuple` returns a tagged tuple that retains from the input tuple

```
projectOnTaggedTuple :: [AttributeName]->TaggedTuple->TaggedTuple
projectOnTaggedTuple attributeNames(TaggedTuple tick index tuple)=
  TaggedTuple tick index (projectOnTuple attributeNames tuple)

projectOnTuple :: [AttributeName]->Tuple->Tuple
projectOnTuple attributeNames tuple =
  Tuple attributeNames (map (getAttribute tuple) attributeNames)
```

Fig. 3. Definition of `projectOnTaggedTuple`

```
evaluateWindow :: WindowOp->[Window]
evaluateWindow (Scan tick tableName) = scan tableName tick (Tick 0)
evaluateWindow (TimeWindow windowScope tick streamOp) =
  createTimeWindow windowScope tick (Tick 0) (evaluateStream streamOp)
evaluateWindow (RowWindow windowScope index streamOp) =
  createRowWindow windowScope index index (evaluateStream streamOp)
evaluateWindow  (WindowSelect predicate windowOp) =
  map (selectOverWindow predicate) (evaluateWindow windowOp)
evaluateWindow  (WindowProject attributeNames windowOp) =
  map (projectOnWindow attributeNames) (evaluateWindow windowOp)
evaluateWindow (WindowAggregation attributeNames windowOp) =
  map (aggregateOverWindow attributeNames) (evaluateWindow windowOp)
evaluateWindow (WindowCrossProduct leftWindowOp rightWindowOp) =
  crossProduct (evaluateWindow leftWindowOp)
               (evaluateWindow rightWindowOp)
```

Fig. 4. Definition of `evaluateWindow`

the `tick`, the `index` and the attributes named in the list. `StreamSelect` is directly defined using `filter` (see Fig. 1).

6.2 Window Stream Operators

A window stream is represented as a lazily evaluated list of windows. Operations that return window streams fall into the following groups: *input operators* that obtain windows from stored tables, *creation operators* that generate windows from streams, *single-window operators* that are applied to the individual windows in a stream independently, and *multiple-window operators* that act on windows from more than one stream. The definition of `evaluateWindow` is in Fig. 4.

Input Most window streams are obtained from tuple streams, as described below. However, tables can be scanned at regular intervals, thereby allowing SNEEql queries to access stored data (e.g., in normal databases, or in persistent memory or in data loggers in sensor networks). `Scan` is defined using the function `scan` (see Fig. 5) which, given the name of a stored extent, the scanning interval specified in the query, and the current tick, returns the stream of windows that results from scanning the table. The function `scanTable` abstracts away from the access to an external table. In Haskell, this can be simulated by generating bags of tuples at the specified points in time.

Creation Window creation operators take as input a stream of tuples and output a stream of windows. Window creation involves determining when to create a

```
scan :: String->Tick->Tick->[Window]
scan tableName scanInterval now =
  let window = Window now (scanTable now tableName)
  in [window] ++ scan tableName scanInterval (now+scanInterval)
```

Fig. 5. Definition of `scan`

```
createRowWindow :: WindowScope->Index->Index->[TaggedTuple]->[Window]
createRowWindow windowScope slide index taggedTuples =
  let tuples = takeWhile (lessEqualsIndex index) taggedTuples
  in  let(TaggedTuple now lastIndex lastTuple) = last tuples
  in  [Window now (getWindowTuples windowScope now index tuples)]
  ++  createRowWindow windowScope slide (index+slide) taggedTuples

getWindowTuples::WindowScope->Tick->Index->[TaggedTuple]->[Tuple]
getWindowTuples _ _ _ [] = []
getWindowTuples windowScope@(RowScope from to)now currentIndex input =
  let passedFrom=dropWhile (lessThanIndex (currentIndex + from))input
  in let window=filter (lessEqualsIndex(currentIndex + to))passedFrom
  in map getTuple window
```

Fig. 6. Definition of `createRowWindow`

new window and which tuples to include in the window. `RowWindow` is defined using `createRowWindow` (see Fig. 6), which, given the offsets, the slide, the current index and the input tuple stream, returns a window stream in which windows are created containing tuples from the tuple stream with a frequency that respects the slide and a size that reflects the number of tuples that satisfy the offsets. As the offsets that characterize window membership are expressed relative to the current index, `getWindowTuples` simply drops tuples whose index is less than `currentIndex` adjusted by the `FROM` offset or greater than `currentIndex` adjusted by the `TO` offset. Note that since SNEEql was designed to query sensor networks, windows are not created at every tick as in [2]. Instead, windows are created at the frequency requested, through the specified slide. There can be zero, one or many windows created at each tick. `TimeWindow` differs from `RowWindow` only in that the window membership test is applied to timestamps rather than to indexes, so the definitions are omitted due to space constraints.

Single- and Multiple-Window Operators The operators `WindowSelect`, `WindowProject` and `WindowAggregation` are evaluated over each window in a stream of windows individually, as if each were a relation (see Fig. 4). The `tick` of a window is not changed by the application of a single-window operator.

Multiple-window operators relate windows from more than one stream. Only `WindowCrossProduct` is considered here (see Fig. 7), but, in our sensor network implementation [8], we rely on the optimizer to rewrite selections over cross products into joins. Many stream systems, such as CQL, map one window per stream to every tick, so that the cross product of two streams of windows is the cross product of the two windows mapped to each tick. Because in SNEEql, there may be zero, one or many windows at each tick, the cross product operator has been extended to deal with ticks for which there is not necessarily exactly one window in each stream. To avoid unnecessary work, cross product only occurs

```
crossProduct :: [Window]->[Window]->[Window]
crossProduct left right =
  let ticks = getTickUnion left right
  in tickCrossProduct ticks left right

tickCrossProduct :: [Tick]->[Window]->[Window]->[Window]
tickCrossProduct (tick:ticks) left right =
  let leftTick = findLastTick tick left
  in let rightTick = findLastTick tick right
  in let leftWindows = getWindowsAtTick leftTick left
  in let rightWindows = getWindowsAtTick rightTick right
  in let windowPairs = mapMap windowCrossProduct leftWindows rightWindows
  in windowPairs ++ tickCrossProduct ticks left right

windowCrossProduct :: Window->Window->Window
windowCrossProduct (Window leftTick leftTuples)
           (Window rightTick rightTuples) =
  let windowTick = maxTick leftTick rightTick
  in let tuples = mapMap concatTuples leftTuples rightTuples
  in Window windowTick tuples
```

Fig. 7. Definition of `windowCrossProduct`

at ticks where at least one of the streams contains a window. Given two window streams, `crossProduct` identifies these ticks and executes a cross product at each such tick. At each identified tick, the most recent window(s) in each stream at or before this tick are identified. Every such window identified in each stream is combined with those from the other stream, thereby creating a new window for each pair. The `windowCrossProduct` function takes two windows as input, removes their `tick`, concatenates every tuple from one window with every tuple from the other window, and uses the most recent timestamp found as the `tick` of the new window. A `mapMap` function (i.e., a map on a list of lists) applies `windowCrossProduct` to every pair of windows identified in `tickCrossProduct` as having timestamps that should be matched.

6.3 Window-to-Stream Converters

A SNEEql query can specify that a stream of windows is to be converted into a stream of tuples using the keywords `RSTREAM`, `ISTREAM` and `DSTREAM` from CQL [2]. In combination with the `TimeWindow` and `RowWindow` operators, the corresponding conversion operators enable window queries to be nested within the `FROM` clauses of stream queries, and *vice versa*. `RStream` is defined using the function `rStream` (see Fig. 8), which, given the index of the next tuple to be returned and the window stream from which tuples are obtained, appends to the output stream all the tuples in each window. In common with the other window to stream operators, each tuple in the output stream receives its `tick` from its source window and a running `index` unique to the stream. `IStream` is defined using `iStream` (see Fig. 8), which, given the index of the next tuple to be returned, the tuples in the previous window, and the window stream from which tuples are obtained, appends tuples into the output stream that were not in the

```
rStream :: Index->[Window]->[TaggedTuple]
rStream index ((Window tick tuples):windows) =
 (append  tuples tick index) ++
    rStream (index + Index (length tuples)) windows

append :: [Tuple]->Tick->Index->[TaggedTuple]
append [] _ _ = []
append (tuple:tuples) tick index =
  [TaggedTuple tick index tuple]++(append tuples tick (inc index))

iStream :: Index->[Tuple]->[Window]->[TaggedTuple]
iStream index previousTuples ((Window tick tuples):windows) =
  let insertTuples = tuples \\ previousTuples
  in (append  insertTuples tick index) ++
      iStream (index + Index (length insertTuples)) tuples windows
```

Fig. 8. Definition of `rStream` and `iStream`

previous window. `DStream`, which returns the tuples deleted from the window is similar to `IStream` except for the swapped arguments in the list difference line.

7 Related Work

The literature on stream data management is quite large: we focus on window-based accounts. With respect to sensor network query languages, no formal description of either TinyDB [10] or Cougar [14] has been published. The TinyDB query language uses materialization points to offer limited support for blocking operators, does not allow window specifications (other then for aggregates) and is, therefore, less expressive than SNEEql in these respects. The Cougar query language has not been sufficiently described to allow a meaningful contrast to be drawn. With respect to query languages on pushed streams, CQL [2] was given a denotational semantics [3]. While being the major inspiration behind it, CQL is less expressive than SNEEql. For example, in assuming that there is exactly one window associated with every tick (whereas in SNEEql, there can be zero, one or many), and in not supporting bag of tuples (i.e., relations) as a primitive collection type. While the denotational semantics given to CQL is exemplary, it is, by its nature, less informative from an implementer's viewpoint than the one contributed in this paper, in that query language implementation tends to build on algebras. The other previous formal treatments fail to be as exhaustive and systematic as the one given here, in that the account in [11] only applies to pushed, punctuated streams, while the account in [1] only applies to pushed streams and to a significantly more constrained notion of window.

8 Conclusions

This paper has shown that a query language over streams and relations can be given a formal semantics that clarifies the relationship between stream query processing and classical query processing. Building on the pioneering work on CQL [2], the paper shows that taking windows as a collection type obtained

by type-conversion operations on streams suffices to encompass more cases of interest in the same semantic account than previously done. Thus, the paper has shown how streams and relations, as well as push- and pull-based streams, relate to one another in the presence of both non-blocking and blocking operators. As shown, SNEEql advances on previous work in supporting windows without the requirement to do so in every query. By defining a window stream as a stream of zero, one or even many windows per tick, rather than exactly one for each tick, SNEEql avoids having to drop windows if too many tuples arrive at once and having to contend with repeated evaluations that produce repeated results

The semantics of SNEEql, as described in this paper, has been implemented in Haskell, with the resulting code, using simulated inputs, acting as a SNEEql interpreter. The subset of SNEEql that pertains to sensed extents has been fully implemented (as was TinyDB) over a nesC/TinyOS software environment, as described in [8]. Work is in progress to accommodate, as informed by the semantics contributed in this paper, pushed streams and stored extents.

Acknowledgements This work was funded by UK EPSRC WINES EP/C014774/1 DIAS-MC project. We are grateful for this support and to our collaborators in the project. C.Y.A. Brenninkmeijer thanks the School of Computer Science.

References

1. D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, et al. Aurora: A new Model and Architecture for Data Stream Management. *VLDB J.*, 12(2):120–139, 2003.
2. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.
3. A. Arasu and J. Widom. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Record*, 33(3):6–12, 2004.
4. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS*, pages 1–16, 2002.
5. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.
6. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, pages 379–390, 2000.
7. C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, pages 647–651, 2003.
8. I. Galpin, C. Y. A. Brenninkmeijer, F. Jabeen, A. A. A. Fernandes, and N. W. Paton. An Architecture for Query Optimization in Sensor Networks. In *Proc. ICDE*, 2008.
9. L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.
10. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
11. D. Maier, J. Li, P. A. Tucker, K. Tufte, and V. Papadimos. Semantics of Data Streams and Operators. In *ICDT*, pages 37–52, 2005.
12. D. Maier, P. A. Tucker, and M. Garofalakis. *Filtering, Punctuation, Windows and Synopses*, chapter 3 in StreamDataManagement: N. A. Chaudhury et al. (eds). Springer, 2005.
13. E. A. Rundensteiner, L. Ding, T. M. Sutherland, Y. Zhu, et al. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *VLDB*, 2004.
14. Y. Yao and J. Gehrke. Query Processing in Sensor Networks. In *CIDR*, 2003.