T U M

I N S T I T U T F Ü R I N F O R M A T I K

Data Stream Sharing

Richard Kuntschke Bernhard Stegmaier Alfons Kemper

TUM-I0504
April 05

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-04-I0504-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Data Stream Sharing*

Richard Kuntschke Bernhard Stegmaier Alfons Kemper

Lehrstuhl Informatik III: Datenbanksysteme
Fakultät für Informatik
Technische Universität München
Boltzmannstraße 3, D-85748 Garching bei München, Germany
{richard.kuntschke|bernhard.stegmaier|alfons.kemper}@in.tum.de

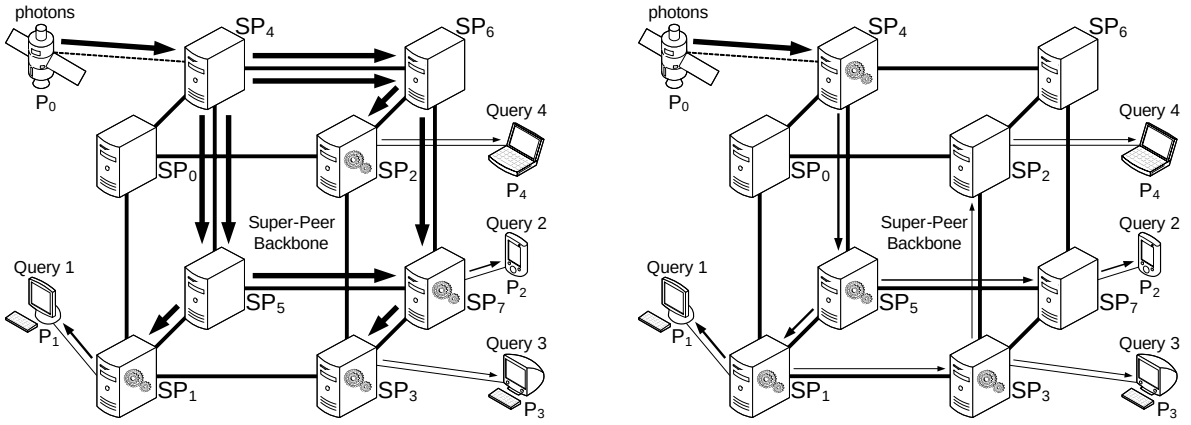
Abstract

Recent developments in novel application areas like e-science, e-health, and e-business demonstrate the increasing importance of processing data streams on-the-fly instead of storing the ever increasing amount of, e. g., sensor generated data prior to processing. Efficient processing of data streams can be achieved by employing optimization techniques like *in-network query processing* and *multi-subscription optimization*, thus enabling *data stream sharing* in data stream management systems (DSMSs). On the basis of existing infrastructures for distributed computing like *peer-to-peer (P2P) networks* and *grid computing* standards (OGSA), we present a novel approach for optimizing the integration, distribution, and execution of newly registered continuous queries over data streams in grid-based P2P networks. We introduce *Windowed XQuery (WXQuery)*, our XQuery-based subscription language for continuous queries over XML data streams supporting window-based operators. Concentrating on filtering and window-based aggregation, we present our stream sharing algorithms, describe our grid-based P2P implementation in the *StreamGlobe* project, and show experimental evaluation results from the astrophysics application domain to assess our approach.

1 Introduction

Data stream processing is currently gaining importance due to the developments in novel application areas like e-science, e-health, and e-business, e. g., considering RFID [BLHS05]. In e-science for example, it can be observed that scientific experiments and observations in many fields, e. g., in physics and astronomy, create huge volumes of data which have to be interchanged and processed. With experimental and observational data coming in particular from sensors, online simulations, etc., the data has an inherently streaming nature. The aim in e-science is to enable various researchers and research institutes to share their research data, e. g., sensor measurements of complex experiments in physics or telescope observation data in astronomy. This allows for resource sharing as well as multiple evaluation and analysis of data. Furthermore, continuing advances will result in even higher data volumes, rendering storing all of the delivered data prior to processing increasingly impractical. Also, transmitting all the data over physically limited and therefore eventually congested network connections is a problem. This is especially true if only small subsets of the data or some processing results—which usually constitute a much smaller data volume than the input data—are actually needed. To enable efficient data sharing and processing, it is imperative to reduce the huge amounts of data generated by scientific experiments and observations as early as possible and to reuse computational results if appropriate. Thus, the transmission of unnecessary data, the redundant transmission of data streams, the redundant execution of operators, and therefore network and peer overload can be prevented. Hence, in such e-science scenarios as well as in many other fields, processing and sharing of data streams will play a decisive role. It will enable new possibilities for researchers, since they will be able to subscribe to interesting data streams of other scientists without having to set up their own devices or experiments. This results in much better utilization of expensive

*This research is supported by the German Federal Ministry of Education and Research (BMBF) within the D-Grid initiative under contract 01AK804F and by Microsoft Research Cambridge (MSRC) under contract 2005-041.



(a) Without data stream sharing

(b) With data stream sharing

Figure 1: Example DSMS scenario

equipment such as telescopes, satellites, etc. Further, processing and sharing data streams on-the-fly in the network helps to reduce network traffic and to avoid network congestion. Thus, even huge streams of data can be handled efficiently by removing unnecessary parts early on, e. g., by early filtering and aggregation, and by sharing previously generated data streams and processing results.

We propose *data stream sharing* as a new optimization technique addressing these issues. Data stream sharing is based on two main optimization approaches. These are (1) *in-network query processing* for distributing and executing newly registered continuous queries in the network and (2) *multi-subscription optimization* for enabling the reuse and sharing of existing (parts of) data streams that were generated to satisfy previously registered subscriptions.[1]

These optimizations are an integral part of our *StreamGlobe* system [SKK04, KSKR05]. To enable them, we use *peer-to-peer (P2P) networking* techniques. P2P has gained a lot of attention in the context of exchanging persistent data—in particular for *file sharing*. In contrast to that, we apply P2P networks for the dissemination of individually subscribed and transformed data streams, allowing for *data stream sharing*. By using the computational capabilities of peers in the P2P network, we can push data stream transforming operators into the network, thus enabling efficient in-network query processing. This leads to increased flexibility since any kind of peer can register arbitrarily complex queries regardless of its own computational abilities. Furthermore, load balancing among peers and the reduction of network traffic by employing early filtering and aggregation close to the data sources are enabled. At the same time, multi-subscription optimization allows for data stream sharing and reuse of computational results among various peers. This yields a reduction of both, network traffic and peer load. Ultimately, these optimizations improve overall performance as more clients subscribing to data streams can be accomplished. We propose StreamGlobe as a prototype to meet the challenges described above in a generic distributed data stream management system (DSMS). The StreamGlobe system architecture is based on a P2P overlay network that is organized as a *super-peer network* [YGM03], i. e., peers are classified into *super-peers* and *thin-peers*. Super-peers are powerful servers which form a stationary super-peer backbone network. Thin-peers—often simply called peers in the following—are less powerful devices that can be registered at a super-peer and deliver data streams or register queries in the network. The StreamGlobe implementation adheres to established *grid computing* [FK04] standards like the *Open Grid Services Architecture (OGSA)* [FKNT02] and therefore fits seamlessly into existing e-science platforms. Peers in StreamGlobe are implemented as collaborating grid services in the Globus Toolkit [Glo05] grid middleware. To ensure interoperability, StreamGlobe is built on top of standards like XML and XQuery for representing data streams and specifying subscriptions.

As a motivating example for the application of StreamGlobe, we introduce an astrophysical e-science application. Consider Figure 1 which illustrates an exemplary network once without and once with data stream sharing. Here, SP_0 to SP_7 are the super-peers that constitute the super-peer backbone network

[1] The terms *query*, *continuous query*, and *subscription* are treated as synonyms throughout this paper.

```
<!ELEMENT photons  (photon*)>
<!ELEMENT photon   (coord, phc, en, det_time)>
<!ELEMENT coord    (cel, det)>
<!ELEMENT cel      (ra, dec)>
<!ELEMENT ra       (#PCDATA)>
<!ELEMENT dec      (#PCDATA)>
<!ELEMENT det      (dx, dy)>
<!ELEMENT dx       (#PCDATA)>
<!ELEMENT dy       (#PCDATA)>
<!ELEMENT phc      (#PCDATA)>
<!ELEMENT en       (#PCDATA)>
<!ELEMENT det_time (#PCDATA)>
```

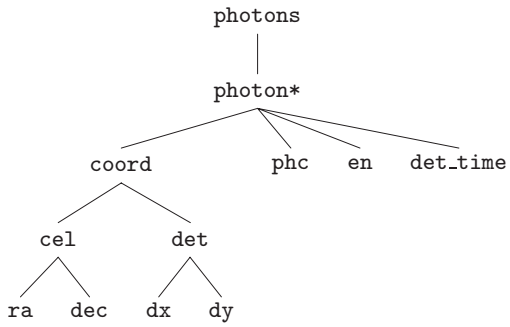


Figure 2: DTD of example data stream `photons`

and P_0 to P_4 are thin-peers. P_0 is a satellite-bound telescope that detects photons and registers a data stream called `photons` at super-peer SP_4. This data stream contains real astrophysical data collected during the ROSAT All-Sky Survey (RASS) [VAB+99] which we obtained through our cooperation partners from the Max Planck Institute for Extraterrestrial Physics [MPE05].

In StreamGlobe, we deal with streams of XML data. Stream `photons` complies to a DTD of the following structure. Each element in our example has an occurrence of exactly one. As its name implies, the data stream delivers a stream of photons detected by the telescope's photon detector. Each photon in the data stream is represented by an XML element `photon` that incorporates the coordinates of the corresponding photon (`coord`), the pulse height channel, i. e., the detector pulse caused by the photon when hitting the detector (`phc`), the photon's energy in keV (`en`), and the time of its detection in seconds since the start of the observation (`det_time`). The coordinates consist of the celestial coordinates of the position in the sky where the photon was detected (`cel`) and the coordinates of the detector pixel where the photon actually hit the detector (`det`). Celestial coordinates comprise the right ascension (`ra`) and declination (`dec`) of a point in the sky, measured in degrees. Detector pixel coordinates simply contain the two-dimensional coordinates of the respective pixel on the detector plain (`dx`, `dy`). The DTD of the example data stream `photons` is shown in Figure 2, together with its tree structure.

For simplicity, we consider only one single data stream in our example. However, multiple data streams can of course be registered at one or more super-peers in the network. Also note that while, except for the `photon` element, each element in the example DTD occurs exactly once, more complex DTDs with varying element occurrences ("`?`", "`+`", "`*`", "`|`") are also possible and can be handled accordingly.

Peers P_1 to P_4 in the example network are devices of astrophysicists used to register subscriptions in the network referencing the available data stream as input. Subscriptions are registered using *WXQuery*, our XQuery-based subscription language that will be introduced in detail in Section 3. We will only consider Queries 1 and 2 of Figure 1 here. Queries 3 and 4 will be presented in Section 3. All queries reference data stream `photons` as their single input. Query 1 (Q1) is shown in Figure 3 below.

```
<photons>
  { for $p in stream("photons")/photons/photon
    where $p/coord/cel/ra >= 120.0
      and $p/coord/cel/ra <= 138.0
      and $p/coord/cel/dec >= -49.0
      and $p/coord/cel/dec <= -40.0
    return
      <vela>
        { $p/coord/cel/ra } { $p/coord/cel/dec }
        { $p/phc } { $p/en } { $p/det_time }
      </vela> }
</photons>
```

Figure 3: Query 1 (Q1)

This query selects an area in the sky that contains the *vela supernova remnant* and delivers the

celestial coordinates, the pulse height channel, the energy, and the detection time of all the photons detected in that area. The `stream` function was newly introduced by us and indicates a possibly infinite data stream used as input to the query. Query 2 (Q2) is shown in Figure 4 below and filters a smaller section of the sky.

```
<photons>
  { for $p in stream("photons")/photons/photon
    where $p/en >= 1.3
      and $p/coord/cel/ra >= 130.5
      and $p/coord/cel/ra <= 135.5
      and $p/coord/cel/dec >= -48.0
      and $p/coord/cel/dec <= -45.0
    return
      <rxj>
        { $p/coord/cel/ra } { $p/coord/cel/dec }
        { $p/en } { $p/det_time }
      </rxj> }
</photons>
```

Figure 4: Query 2 (Q2)

This query selects the area of the *RXJ0852.0-4622 supernova remnant* [Asc98] which is situated within the area of *vela*. Note that the section of the sky selected by Query 2 is completely contained in the section selected by Query 1. Also, Query 2 is only interested in photons having an energy value of at least 1.3 keV.

We first consider Figure 1(a) which shows the traditional scenario of answering queries in the network. The thickness of the arrows associated with the various network connections indicates the size of the data streams transmitted over those connections. Each of the four queries in the system only needs a certain part of the original data stream. However, in each case, the whole stream gets transmitted from the data source to the peer that registered the query, leading to the transmission of unnecessary data. Since query execution for each subscription takes place at the super-peer that the subscribing peer is connected to, queries that perform the same operations on the same input data streams cause redundant execution of operators. Note that this scenario already performs a basic form of data stream sharing by transmitting a stream only once to a peer and forwarding it multiple times to various other peers if required.

Figure 1(b) shows the situation when using our stream sharing approach which answers newly registered subscriptions using (parts of) data streams already present in the network. These data streams have been generated for satisfying previously registered continuous queries. We assume that Queries 1 to 4 have been registered one after another in ascending order in our example. Obviously, network traffic and processing overhead can be significantly reduced by avoiding redundant transmissions and computations through sharing previously generated data streams. For example, when Query 1 is registered, its execution can be pushed into the network and computed at SP_4 instead of SP_1. The result is then routed to P_1 via SP_5 and SP_1. When Query 2 is registered afterwards, it can reuse the stream constituting the answer for Query 1 at SP_5 because the result of Query 2 is completely contained in the answer for Query 1. The result data stream of Query 1 is duplicated at SP_5, yielding two identical streams. One is used to answer Query 1, the other is filtered using the selection and projection of Query 2. This results in a new stream that constitutes the result of Query 2 which is subsequently routed to P_2 via SP_7. More details on this and the registration of Queries 3 and 4 will be set forth in Section 4.

The contributions presented in this paper comprise the following. First, we introduce *Windowed XQuery (WXQuery)*, our XQuery-based subscription language for continuous queries over XML data streams enabling the formulation of queries including window-based aggregation operators. Then, we develop a properties representation of data streams and subscriptions in the network. This representation forms the basis for finding reusable streams enabling data stream sharing. Finally, we introduce a cost model and algorithms for optimizing the evaluation of newly registered continuous queries in a P2P data stream management system by sharing possibly preprocessed data streams.

The paper is organized as follows. In Section 2, we describe the problems of matching subscriptions and data streams, and of suitably placing query operators in the network. Futhermore, we introduce

a formal notion of data streams and data windows. In Section 3, we introduce WXQuery. Our new data stream sharing approach is presented in Section 4. Section 5 gives an overview of our prototype implementation and Section 6 shows some evaluation results. In Section 7, we present related work. Section 8 concludes and states future work.

2 Preliminaries

In this section, we introduce and describe the problem dealt with in this paper as well as our notion of data streams and data windows.

2.1 Problem Statement

Our goal is to efficiently integrate, distribute, and execute newly registered continuous queries over data streams in P2P networks, thus reducing network traffic and peer load, avoiding network congestion and peer overload, enabling load balancing among peers and network connections, and increasing flexibility in terms of which peers can register which kinds of subscriptions. We employ a local optimization approach to incrementally include new subscriptions in an existing network. Note that static multi-query optimization on a set of subscriptions is a different problem that is not covered in this paper. However, if appropriate, it could be used for periodic or event-based global reoptimization to complement our approach. The core problem that has to be solved in order to achieve our goal is the discovery of reusable (parts of) data streams. Solving this problem requires taking into account schema- or structure-based information, e. g., projections, as well as content-based information, e. g., selections, about subscriptions and data streams. In order to enable the efficient comparison of subscriptions and data streams, we abstract from the textual representation of the subscription and the data stream schema. Instead, we use a properties approach introduced in Section 4.1 to gather the relevant properties of subscriptions and data streams. On this basis, it is possible to compare the properties of a new subscription with those of existing data streams in the network. In our approach, the contents of a data stream are represented by the properties of the subscription generating the respective stream. Therefore, a subscription corresponds to a data stream, i. e., the result data stream of that subscription, and vice versa. This implies that a subscription and its corresponding result data stream are represented by the same properties. Consequently, subscriptions and data streams are treated symmetrically in the following.

The super-peers of the overlay backbone network are connected according to some topology. The contributions presented in this paper are independent of the actual topology used. Therefore, we do not elaborate on this issue any further. For the validity of our approach it is, however, important that there is a reasonable relation between the P2P overlay network we consider and the underlying physical network. Optimization of network traffic on the overlay network must yield a corresponding optimization benefit on the physical network. This implies, for example, that topological neighbor peers should also be geographically close. An important question in this context is how to construct the overlay network on the basis of an existing physical network in order to achieve such a reasonable relation. Similar problems have already been examined [RHKS02]. However, solving this issue is not the scope of this paper.

In general, the above mentioned comparison of subscriptions and data streams will identify more than one shareable stream for a given subscription. This leads to multiple possible evaluation plans. The choice for one of those plans is made according to a cost function taking into account additional network traffic and peer load caused by the new operators. The cost function will be introduced in Section 4.3.

2.2 Data Streams

Before dealing with data stream sharing, we first introduce our notion of a data stream in the context of this work.

Definition 2.1 (Data Stream) A data stream S is an ordered sequence $\langle s_1, s_2, \ldots, s_n \rangle$ of data items s_i with $1 \leq i \leq n$ and $n \in \mathbb{N} \cup \{+\infty\}$. A value of $n = +\infty$ indicates an infinite data stream. Only the next data item arriving on the stream can be read from the stream at any time. After reading a data item s_i from the stream, access to data item s_j with $j \in \mathbb{N}$ and $j \leq i$ is not possible any more. □

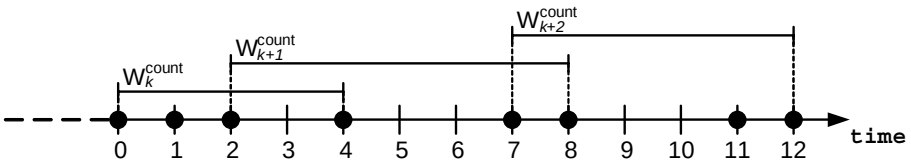


Figure 5: Example of an item-based data window with window size 4 and step size 2

Data streams in our context are possibly infinite streams of XML data. Each stream consists of a sequence of XML elements called *data stream items*. The sequence of data stream items is enclosed in a *data stream root element*. The data stream root element can only have one single subelement in the schema or DTD of the stream, namely the root element of a data stream item. This element can and generally will have multiple occurrences. The structure of a data stream item is arbitrary. In the DTD of our example data stream `photons` of Figure 2, the data stream root element is `photons` and the data stream items are the XML subtrees rooted at the `photon` elements. Note that the opening `photons` tag marks the beginning of the corresponding data stream while the closing `photons` tag marks its ending. The stream contains a possibly infinite sequence of `photon` elements. A data stream can be referenced via a *stream node*, corresponding to a document node in standard XML.

The order of the data objects in a data stream depends on the sorting of the stream. A data stream can deliver its data stream elements either sorted according to a certain order or unsorted.

Definition 2.2 (Sorting) A data stream $S = \langle s_1, s_2, \ldots, s_n \rangle$ is called sorted according to an order $\trianglelefteq$, if and only if:

$$\forall i, j \in \mathbb{N} \cup \{+\infty\} : i < j \Leftrightarrow s_i \trianglelefteq s_j$$

□

Note that the order of data stream items as they are produced and sent out on the stream by the data source implies a *stream order* corresponding to the document order of persistent XML files.

2.3 Data Windows

For being able to execute stateful operators like aggregations over possibly infinite data streams, we employ a window-based approach. The contents of a data stream are partitioned into a sequence of data windows and the contents of each data window can subsequently be processed, e. g. by computing an aggregate value over the window contents. In accordance with the literature, we distinguish two variants of data windows. These are *item-based* data windows and *time-based* data windows.

2.3.1 Item-based Data Windows

Item-based data windows have a fixed size in terms of the number of items contained in the window. Items are inserted into the window as they arrive in the data stream. As soon as the window is full, the processing of the window contents starts. This can be as simple as writing the window contents to the output stream, which basically corresponds to grouping the data stream. However, it can also involve a more complex computation like an aggregation. After processing the window contents, the window is updated. The update is performed by sliding the window, leading to the removal of some items from the window contents and to the addition of some new items arriving in the data stream. In an item-based data window, the step size of the window, i. e., the amount by which the window slides along, is given in terms of the number of items that need to be removed from the window contents and replaced by new ones read from the data stream during an update. Data items are removed from the window contents according to a FIFO strategy. An example for an item-based data window with window size 4 and step size 2 is shown in Figure 5. The bullets in the figure indicate data items arriving on the stream.

Definition 2.3 (Item-based Data Window) Formally, the k-th item-based data window W_k^{count} with window size Δ and step size μ on a data stream S is defined as follows:

$$W_k^{\text{count}}(\Delta, \mu) := \{s_i \in S \mid 1 + (k-1) \cdot \mu \leq i \leq \Delta + (k-1) \cdot \mu\}$$

□

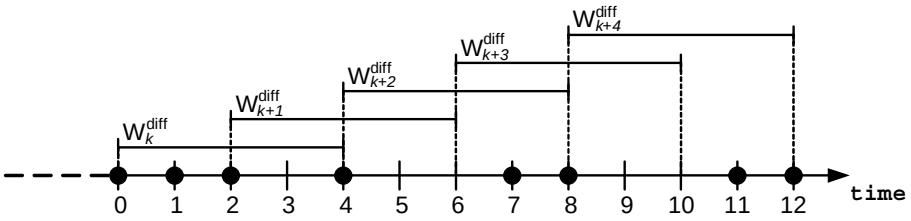


Figure 6: Example of a time-based data window with window size 4 and step size 2

2.3.2 Time-based Data Windows

Time-based data windows are not organized by the number of items contained in them but by the value of a certain subelement which is called the *reference element* of the window. A data item is contained in the window if and only if its reference element value is greater than or equal to the lower bound and less than the upper bound of the window. This implies that there must be a total order defined on the values of the reference element and that the lower and upper bounds of the window must be defined in terms of the data type of the reference element. In this paper, we will always assume integer values for the window bounds and the reference element value. Note that the reference element value does not necessarily need to be a real time value. Rather, it suffices if it is an abstract time value which basically can be any value of a totally ordered set of values. Also note that, for time-based windows to work properly on data streams, the reference element values of subsequent data stream items in a data stream must be monotonically increasing. The step size of a time-based data window then indicates the amount by which the lower and upper bounds of the window need to be increased during updating the window. All elements with reference element values less than the new lower bound are subsequently removed from the window while all new elements arriving in the data stream with reference element values greater than or equal to the new lower bound and less than the new upper bound are inserted into the window. When the data window is completely filled, i. e., all data items with reference element values satisfying the above condition have been inserted, the window contents can be processed and the window can be updated again. An example for a time-based data window with window size 4 and step size 2 is given in Figure 6. Again, the bullets in the figure indicate data items arriving on the stream.

Definition 2.4 (Time-based Data Window) Formally, the k-th time-based data window W_k^{diff} with reference element r, window size Δ, and step size μ on a data stream S is defined as follows:

$$W_k^{\text{diff}}(r, \Delta, \mu) := \{s_i \in S \mid s_1.r + (k-1) \cdot \mu \leq s_i.r < s_1.r + \Delta + (k-1) \cdot \mu\}$$

Reference element values r are monotonically increasing:

$$\forall s_i, s_j \in S : i < j \Rightarrow s_i.r \leq s_j.r$$

□

3 The WXQuery Subscription Language

In StreamGlobe, subscriptions over XML data streams are registered using a subscription language called *Windowed XQuery (WXQuery)*. WXQuery is a fragment of XQuery [W3C05c] that has been augmented with support for window-based operators.

In Definition 3.1 below, α and β are WXQuery expressions and χ denotes a condition. A tag name is denoted by t. Further, $\$x$ and $\$y$ are variables representing XML trees, where $\$y$ can also start with a function call to reference a document node or the stream node of a data stream like `stream("photons")` in the example subscriptions. A variable representing the result of a window-based aggregation operation is denoted by $\$a$. The variable $\$z$ can represent any of the three kinds of variables $\$x$, $\$y$, or $\$a$ as described above. We use $\overline{\pi}$ to denote a relative path that only employs the child axis ("`/`"). It does not include wildcards ("`*`"), conditions ("`[`p`]`"), or other axes (e. g., "`//`"). A relative path π differs from $\overline{\pi}$ in that it can also contain conditions. An aggregation operator is denoted by Φ, i. e., $\Phi \in \{\texttt{min}, \texttt{max}, \texttt{sum}, \texttt{count}, \texttt{avg}\}$.

Expressions enclosed in $[\![\,]\!]^?$, $[\![\,]\!]^*$, or $[\![\,]\!]^+$ in the definition are optional, can occur zero or more times, or can occur one or more times, respectively. A vertical bar (|) indicates an alternation. An expression of the form $\alpha_{i_1,\ldots,i_n}$ represents a WXQuery expression from a restricted set of expressions. For example, $\alpha_{1,2}$ stands for any one of the two element constructor expressions numbered 1 and 2 in the definition below and $\alpha_{3,4,5,6,7}$ stands for any one of the remaining expressions numbered 3 to 7.

Definition 3.1 (WXQuery) The WXQuery subscription language comprises all subscriptions that consist only of the following expressions:

1. <t/>
   (empty direct element constructor)

2. <t> $[\![\alpha_{1,2} \mid \{\alpha_{3,4,5,6,7}\}]\!]^*$ </t>
   (direct element constructor)

3. $[\![$`for` $\$x$ `in` $\$y[\![/\pi]\!]^?[\![|$`count` Δ $[\![$`step` $\mu]\!]^?|$ $\mid$ $|\overline{\pi}$ `diff` Δ $[\![$`step` $\mu]\!]^?|]\!]^?$ $\mid$
   `let` $\$a$ `:=` Φ(`$`$y[\![/\pi]\!]^?$)$]\!]^+$
   $[\![$`where` $\chi]\!]^?$
   `return` α
   (FLWR expression)

4. `if` χ `then` α `else` β
   (conditional)

5. $\$y/\pi$
   (output of subtrees reachable from node $\$y$ through path π)

6. $\$z$
   (output of subtree rooted at node $\$z$)

7. (`(`$[\![\alpha[\![$`,`$\beta]\!]^*]\!]^?$`)`)
   (sequence)

□

The WXQuery EBNF grammar is shown in Appendix B on page 40. The FLWR expression in the WXQuery definition introduces our new syntax for expressing data windows, e. g., for use with window-based aggregation operators. Query 3 (Q3) in the network of Figure 1 is an example for the use of such an operator. The query is shown in Figure 7 below.

```
<photons>
  { for $w in stream("photons")/photons/photon
      [coord/cel/ra >= 120.0 and
       coord/cel/ra <= 138.0 and
       coord/cel/dec >= -49.0 and
       coord/cel/dec <= -40.0]
      |det_time diff 20 step 10|
    let $a := avg($w/en)
    return <avg_en> { $a } </avg_en> }
</photons>
```

Figure 7: Query 3 (Q3)

Query 4 (Q4) employs a different window and is shown in Figure 8 below.

```
<photons>
  { for $w in stream("photons")/photons/photon
      [coord/cel/ra >= 120.0 and
       coord/cel/ra <= 138.0 and
       coord/cel/dec >= -49.0 and
       coord/cel/dec <= -40.0]
      |det_time diff 60 step 40|
    let $a := avg($w/en)
    where $a >= 1.3
    return <avg_en> { $a } </avg_en> }
</photons>
```

Figure 8: Query 4 (Q4)

The definition of a data window is enclosed in "|" characters. Item-based windows—indicated by the keyword `count`—contain a fixed number of items given by the numeric value of Δ. Optionally, a step size μ determining the update interval of the data window can be specified. For example, the window `|count 20 step 10|` defines a data window that always contains 20 data items and, during each update, removes the 10 oldest entries from the window while adding the next 10 new data items arriving in the stream. If omitted, the step size defaults to the value of Δ, meaning the contents of the window are completely replaced by new ones during each update.

The situation is analogous for time-based windows, except that Δ indicates the size of the window in time units and the step size indicates the time interval between two successive data windows. Again, the step size defaults to Δ if omitted. Time-based windows can only be applied on data streams that are sorted according to the values of particular *reference element* that is used to control the window. This premise could be somewhat relaxed to a fuzzy order by requiring that a fixed sized buffer is sufficient to derive the total order. An example for a time-based window is `|det_time diff 60 step 40|` in Query 4. Note that the path inside the window is not meant to be evaluated yielding a sequence as defined by the conventional XQuery semantics. Rather, it specifies the reference element controlling the window. The path to the reference element is specified relative to the context node of the data window.

It is worth pointing out that data windows as introduced above could also be expressed using conventional XQuery syntax. Compare for example the WXQuery specifying an item-based data window in Figure 9 with a possible equivalent formulation in standard XQuery in Figure 10. The window construction in standard XQuery is handled by a recursive function *cwin* that returns the next window each time the function is called. An example for a query with a time-based data window is shown in Figures 11 and 12, respectively. Again, a recursive function *dwin* is used for window construction in XQuery. The reasons why we introduced a new window syntax in WXQuery are twofold.

- First, as can be seen from the examples, the new syntax is much less verbose and easier to read than the standard syntax.
- Second, the semantics of the recursive function in standard XQuery requires reading the whole input data before starting to build the first window. This blocking behavior is not applicable when dealing with possibly infinite data streams. Therefore, the new window syntax in WXQuery is also meant to express the streaming nature of the query and of query processing.

Note that in the standard XQuery syntax, an explicit root element for each data window is introduced. It shows up as a direct element constructor in the recursive functions of Figures 10 and 12, constructing an element `cw` or `dw` enclosing the window contents, respectively.

```
for $w in doc("data.xml")/a/b|count 4 step 2|
return
  <result>
    <win> { $w } </win>
  </result>
```

Figure 9: WXQuery with item-based data window

```
declare function local:cwin($count as xs:integer,
                            $step as xs:integer,
                            $data as node()*) as node()*
{
  let $cwin := fn:subsequence($data, 1, $count)
  let $tail := fn:subsequence($data, $step + 1)
  return
    if (fn:count($data) < $count) then
      ()
    else
      if (fn:count($data) = $count) then
        (<cw> { $cwin } </cw>)
      else
        (<cw> { $cwin } </cw>, local:cwin($count, $step, $tail))
};

for $x in doc("data.xml")/a
return
  <result>
    { for $w in local:cwin(4, 2, $x/b)
      return
        <win> { $w/* } </win> }
  </result>
```

Figure 10: XQuery with item-based data window

```
for $w in doc("data.xml")/a/b|c diff 4 step 2|
return
  <result>
    <win> { $w } </win>
  </result>
```

Figure 11: WXQuery with time-based data window

The query formulations in standard XQuery also make explicit the behavior when reaching the end of a finite input stream. The end of a stream is indicated by an end of stream message, i. e. a closing stream root element tag. Three different semantics are possible in such a case.

Cut The *cut* semantics only returns data windows that are guaranteed to contain all relevant data. In the case of an item-based data window, this means that the final data window is not returned if it contains less data elements than specified by the window size. In the case of a time-based data window, processing terminates whenever the final data item arriving in the input stream enters the current data window. The corresponding data window is not returned. This implies that the cut semantics can lead to some items at the end of the data stream never being returned as the contents of a window. The XQueries of Figures 10 and 12 yield this semantics.

Gather The *gather* semantics gathers all remaining data items at the end of a data stream in one final window and returns this window before terminating. In the case of an item-based window, this can cause the final window to contain less elements than specified by the window size. In the case of a time-based window, the final window returned is the first window containing the final data item of the input data stream. The XQueries of Figures 28 and 30 in Appendix A on page 37 yield this semantics.

Run The *run* semantics continues to construct and return data windows until the final data item from the input data stream leaves the current data window during updating the window contents. All non-empty data windows up to that point are returned before terminating processing. This causes the windows to run empty when reaching the end of the input data stream. Therefore, in the case of item-based data windows, this semantics can lead to the final $\lceil(\Delta - 1)/\mu\rceil$ windows containing less

```
declare function local:dwin($start as xs:integer,
                            $diff as xs:integer,
                            $step as xs:integer,
                            $data as node()*,
                            $refs as node()*) as node()*
{
  let $dwin := for $i in $data
               let $ds := for $d in $i/descendant-or-self::node()
                          where some $r in $refs satisfies $r is $d
                          return $d
               where $ds >= $start and $ds < $start + $diff
               return $i
  let $tail := for $i in $data
               let $ds := for $d in $i/descendant-or-self::node()
                          where some $r in $refs satisfies $r is $d
                          return $d
               where $ds >= $start + $step
               return $i
  return
    if (fn:count($dwin) = fn:count($data)) then
      ()
    else
      (<dw> { $dwin } </dw>, local:dwin($start + $step, $diff, $step, $tail, $refs))
};

for $x in doc("data.xml")/a
return
  <result>
    { for $w in local:dwin(0, 4, 2, $x/b, $x/b/c)
      return
        <win> { $w/* } </win> }
  </result>
```

Figure 12: XQuery with time-based data window

data items than specified by the window size. The XQueries of Figures 29 and 31 in Appendix A on page 37 yield this semantics.

We use the cut semantics for item-based data windows and the run semantics for time-based data windows in our StreamGlobe implementation. Note that the handling of the end of a finite data stream is an issue that is dealt with here for the sake of completeness. It does, however, not affect the processing of a running stream before reaching the end of the stream.

The `let` construct of WXQuery is constrained compared to ordinary XQuery as it is only used to assign to a variable a singleton aggregation result value. Conditions in our context, whether they appear in a `where` clause ("χ") or within a path ("`[`p`]`"), are *predicates* that consist of *atomic predicates*. A predicate is either a single atomic predicate or a conjunction of atomic predicates. Atomic predicates can be of the form $\$v\,\theta\,c$ or $\$v\,\theta\,\$w + c$, where $\$v$ and $\$w$ represent paths of the form $\overline{\pi}$, c represents a constant value, and $\theta \in \{=, <, \leq, >, \geq\}$. Constant values can be negative and are either integer values or decimal values with a finite number of decimal places.

Since we concentrate on filtering, i. e., selection and projection, and window-based aggregation operators, the subscriptions we consider in this paper always have one single input data stream. Furthermore, we restrict ourselves to queries with a single `for` loop in the context of this paper. Support for more complex queries is part of future work. Restructuring, e. g., introducing new elements, reordering or renaming output elements, etc., is done in a post-processing step at the super-peer that is connected to the peer that registered the subscription. The result of the post-processing is delivered to the final destination and is not considered for reuse in the network. Therefore, in the case of subscriptions employing only selection and projection operators, the schema of a data stream generated during in-network query

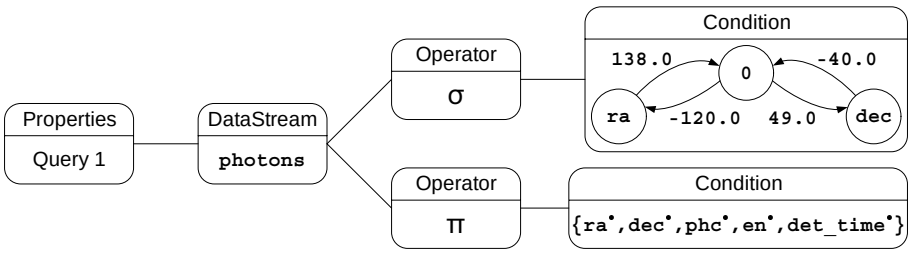


Figure 13: Abstract properties of Query 1

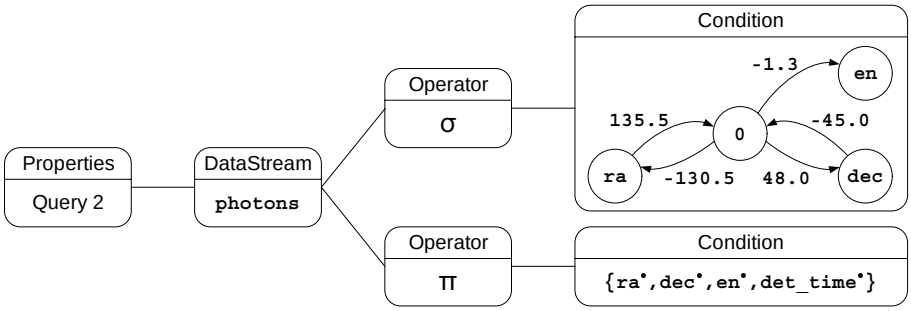


Figure 14: Abstract properties of Query 2

processing can differ from the schema of the corresponding original data stream only by some missing elements which have been removed by a projection operator. Selection operators do not affect the data stream schema at all. Any other more complex data stream schema transformations are postponed to the post-processing step. The only exception are subscriptions containing aggregate operators. In this case, a result data stream with a generic schema is produced by in-network query processing. The generic schema consists of a generic enclosing element for each data stream item in the result data stream and one generic subelement for each aggregate value computed in the subscription. Since attributes in XML data can be converted into corresponding elements, we restrict ourselves to dealing with elements.

4 Data Stream Sharing

This section introduces our properties approach for representing subscriptions and data streams, shareability and dependency relations between the properties of subscriptions and data streams, our cost model, and the algorithms for finding, comparing, and choosing an appropriate stream for satisfying a new subscription. Furthermore, the handling of window-based aggregation operators and some optimizations and extensions that improve the effectiveness and applicability of our approach are presented.

4.1 Properties

Subscriptions and data streams are treated symmetrically in our context. This is due to the fact that a subscription can always be seen as producing a result data stream and a data stream can always be seen as the result of a subscription. Therefore, subscriptions and data streams are also represented by the same properties data structure.

The properties of subscriptions and data streams consist of three parts and describe how the associated (result) data stream was generated. Simplified schematic illustrations of the properties of Queries 1 to 4 from Section 1 are shown in Figures 13 to 16. A subscription or data stream is described by a set of original input data streams, a set of operators for each input data stream used to transform the respective input data stream into the represented (result) data stream and, for each operator, a set of conditions specifying the operator, i. e., selection predicates, projection elements, data window specifications, or aggregation operators together with the identifier of the corresponding aggregated element. Predicates, e. g., selection

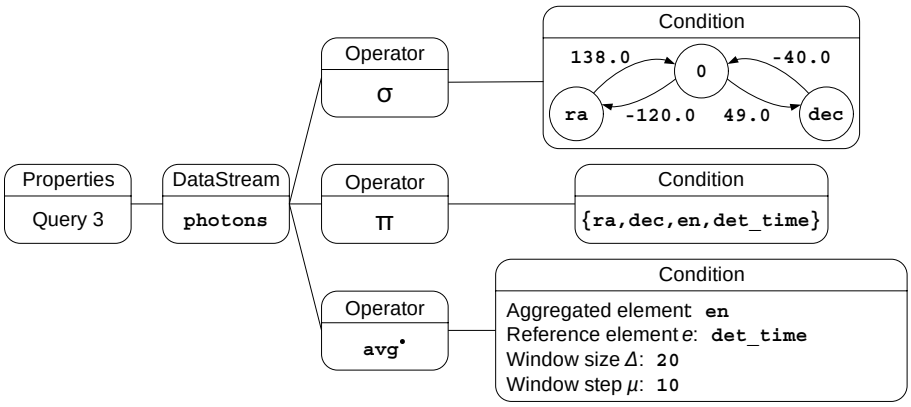


Figure 15: Abstract properties of Query 3

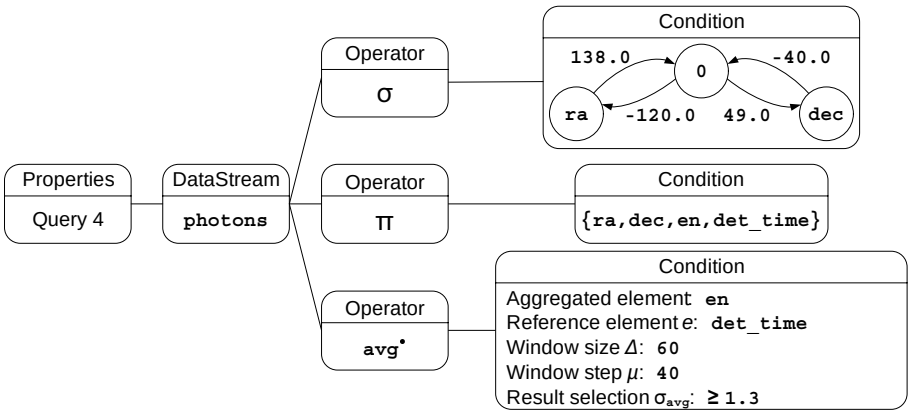


Figure 16: Abstract properties of Query 4

predicates, are stored using a graph representation as shown in Figures 13 to 16. This representation will be introduced in more detail in Section 4.4.3. Data windows for window-based aggregation operators are also stored in a specific format that contains the ordered reference element (only for time-based windows), the window type (`count` or `diff`), the window size (Δ) and the step size (μ).

The properties of a newly registered subscription are obtained by parsing the (W)XQuery subscription and translating it into its corresponding (W)XQueryX [W3C05b] representation. From this representation, which is a standard XML file, the necessary information to be stored in the properties data structure is extracted using XPath [W3C05a]. The properties approach as described here supports queries with multiple input data streams and without nesting. An extended and more flexible properties structure which supports nested queries and enables an even more effective optimization is part of future work.

Note that the properties as described above serve two purposes. First, they represent the parts of the originally queried input data streams that are needed by the corresponding subscription. Second, they describe the contents—relative to the contents of the input data streams—of the data stream produced as a result of that subscription. Also note that properties do not need to represent transformation details like the exact structure of query results as stated in a query's return clause.

4.2 Shareability and Dependency Relations

In order for a subscription to be able to reuse an existing data stream in the network, the data stream to be reused must fulfill certain properties, i. e., it must contain all the necessary information needed for satisfying the subscription.

Definition 4.1 (Shareability Relation) The *shareability relation* $\sqsubset_{sr}$ is defined on a set of subscription and data stream properties. For two subscription or data stream properties p and p', $p' \sqsubset_{sr} p$ indicates that the data stream represented by p can be used as input to satisfy the subscription represented by p'. For the shareability relation to be fulfilled, both properties must reference the same original input data stream. Furthermore, for selection operators, the selection predicate σ'_p of p' must imply the selection predicate σ_p of p, i. e., $\sigma'_p \Rightarrow \sigma_p$. For projection operators, the set R' of elements referenced in p' must be a subset of the set of output elements $\overline{R}$ of p, i. e., $R' \subseteq \overline{R}$. Finally, for window-based aggregation operators, the aggregation operator and any pre-aggregation selection predicates must be equal. Furthermore, any selection on the aggregation result must fulfill the same condition as described for selection operators above. Eventually, the data windows defined in p and p' must be compatible. Let W be the data window defined in p with a window size of Δ and a step size of μ. Let W' be the data window defined in p' with a window size of Δ' and a step size of μ'. Then, W and W' are compatible according to our definition if they have the same window type (item-based or time-based), are defined on the same element, use the same reference element (only for time-based windows), and the following three conditions hold:

- $\Delta' \bmod \Delta = 0$
- $\Delta \bmod \mu = 0$
- $\mu' \bmod \mu = 0$

□

Theorem 4.1 *The shareability relation defines a strict partial order.* □

Proof We need to prove that $\sqsubset_{sr}$ is an irreflexive, antisymmetric, and transitive relation on a set of properties. Let p, p', and p'' be the properties of three subscriptions or data streams.

Irreflexivity ($\neg(p \sqsubset_{sr} p)$):
The irreflexivity of the shareability relation arises from the fact that the set R of elements referenced in p is not a subset of the set $\overline{R}$ of output elements in p, i. e. $\neg(R \subseteq \overline{R})$. Instead, $\overline{R} \subseteq R$ always holds. Furthermore, the window condition $\Delta \bmod \mu = 0$ does not necessarily hold for an arbitrary p. Note that the shareability relation would become reflexive and therefore a weak partial order if only subscriptions with $R = \overline{R}$ and, for each data window defined in the subscription, $\Delta \bmod \mu = 0$ were allowed.

Antisymmetry ($p \sqsubset_{sr} p' \wedge p' \sqsubset_{sr} p \Rightarrow p = p'$):
The antisymmetry of the shareability relation readily follows from its irreflexivity and transitivity. Nevertheless, we present a detailed proof here. First, $p \sqsubset_{sr} p' \wedge p' \sqsubset_{sr} p$ implies that p and p' reference the same input data stream. For selection operators, it yields that $\sigma_p \Rightarrow \sigma_{p'} \wedge \sigma_{p'} \Rightarrow \sigma_p$, i. e., $\sigma_p \Leftrightarrow \sigma_{p'}$. For projection operators, we have $R \subseteq \overline{R'} \wedge R' \subseteq \overline{R}$. Since we also have $\overline{R} \subseteq R \wedge \overline{R'} \subseteq R'$, it follows that $R \subseteq \overline{R'} \subseteq R' \subseteq \overline{R} \subseteq R$ and therefore $R = \overline{R} = \overline{R'} = R'$. Finally, for window-based aggregation operators, it remains to be shown that $W = W'$. Since $\Delta \bmod \Delta' = 0 \wedge \Delta' \bmod \Delta = 0$, we have $\Delta = \Delta'$. Similarly, from $\mu \bmod \mu' = 0 \wedge \mu' \bmod \mu = 0$ follows $\mu = \mu'$. Therefore, we immediately have $W = W'$, since the remaining properties of both windows must be equal due to $p \sqsubset_{sr} p' \wedge p' \sqsubset_{sr} p$.

Transitivity ($p \sqsubset_{sr} p' \wedge p' \sqsubset_{sr} p'' \Rightarrow p \sqsubset_{sr} p''$):
From $p \sqsubset_{sr} p' \wedge p' \sqsubset_{sr} p''$ follows that p, p', and p'' all reference the same input data stream. For selection operators, the transitivity of predicate implication yields $(\sigma_p \Rightarrow \sigma_{p'} \wedge \sigma_{p'} \Rightarrow \sigma_{p''}) \Rightarrow (\sigma_p \Rightarrow \sigma_{p''})$. For projection operators, we have $R \subseteq \overline{R'} \wedge R' \subseteq \overline{R''}$. Because of $\overline{R'} \subseteq R'$, the transitivity of the subset relation yields $R \subseteq \overline{R''}$. Finally, it remains to be shown that a data window W in p with window size Δ and step size μ can reuse a data window W'' in p'' with window size Δ'' and step size μ''. Because of $p \sqsubset_{sr} p' \wedge p' \sqsubset_{sr} p''$ we know that $\Delta \bmod \Delta' = 0 \wedge \Delta' \bmod \Delta'' = 0$. This immediately yields $\Delta \bmod \Delta'' = 0$. Similarly, $\mu \bmod \mu' = 0 \wedge \mu' \bmod \mu'' = 0$ immediately yields $\mu \bmod \mu'' = 0$. Furthermore, because of $p' \sqsubset_{sr} p''$, the condition $\Delta'' \bmod \mu'' = 0$ holds for the data window defined in p''.

The shareability relation $\sqsubset_{sr}$ is not a total relation. For a counter-example, consider Queries 2 and 3 of Sections 1 and 3, respectively. Their properties are incomparable according to $\sqsubset_{sr}$, i. e., $\neg(p_{Q2} \sqsubset_{sr} p_{Q3}) \wedge \neg(p_{Q3} \sqsubset_{sr} p_{Q2})$. This is due to the fact that Query 3 is an aggregating query that returns an aggregate value which cannot be shared by the non-aggregating Query 2. Furthermore, the selection

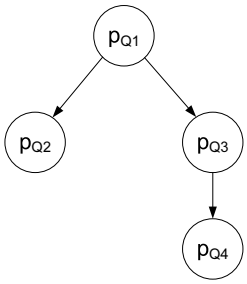


Figure 17: Shareability graph for Queries 1 to 4

predicates of the selection operators in Query 2 are more strict than those in Query 3, causing the result data stream of Query 2 not to contain all the necessary data for Query 3. ■

The shareability relation can be visualized as a *shareability graph*.

Definition 4.2 (Shareability Graph) A shareability graph G_{sg} is a directed graph $G_{sg} = (V_{sg}, E_{sg})$ with a set of vertices V_{sg} and a set of edges E_{sg}. A vertex in the graph represents the properties of a subscription or a data stream, respectively. A directed edge from one vertex to another indicates that the (result) data stream represented by the source vertex of this edge can be shared to satisfy the query represented by the target vertex of this edge. □

The shareability graph for Queries 1 to 4 in the example network of Figure 1(b) is shown in Figure 17. Queries 2, 3, and 4 can share the result of Query 1. Furthermore, Query 4 can also share the result of Query 3. Note that the shareability relation between Queries 1 and 4 is not shown as an explicit edge between p_{Q1} and p_{Q4} in the graph since it is implied by the transitivity of the shareability relation.

Definition 4.3 (Dependency Relation) The *dependency relation* $\sqsubset_{dr}$ is a restriction of the shareability relation including only those pairs of properties that depend on each other in an actual system state. For two subscription or data stream properties p and p', $p' \sqsubset_{dr} p$ indicates that the data stream represented by p is actually used as input to satisfy the subscription represented by p'. □

Theorem 4.2 *The dependency relation defines a strict partial order.* □

Proof The proof follows directly from the definition of the dependency relation and Theorem 4.1. ■

The dependency relation can be visualized as a *dependency graph*.

Definition 4.4 (Dependency Graph) A dependency graph G_{dg} is a directed graph $G_{dg} = (V_{dg}, E_{dg})$ with a set of vertices V_{dg} and a set of edges E_{dg}. A vertex in the graph represents the properties of a subscription or a data stream, respectively. A directed edge from one vertex to another indicates that the (result) data stream represented by the source vertex of this edge is actually shared to satisfy the query represented by the target vertex of this edge. □

The set of vertices of a dependency graph is the same as the set of vertices of the corresponding shareability graph. The set of edges of a dependency graph is a subset of the set of edges of the corresponding shareability graph. For the example network state of Figure 1(b), the dependency graph is identical with the shareability graph shown in Figure 17.

4.3 Cost Model

We now introduce the cost model used in our optimizations. The cost function C focuses on the amount of additional network traffic and peer load caused by answering a new subscription. Other parameters, e. g., latency of network connections, could easily be added. To define C, we need to introduce some notation. Let p be the properties of a new continuous query q that is to be registered in the network. Then $\overline{size}(p)$ denotes the average size of one data stream item (e. g., one `photon`) of the stream represented

by p. Let P_q be the set of properties of all input data streams of q, $\overline{occ}(n_s)$ the average occurrence and $\overline{size}(n_s)$ the average size of element n_s in the input stream represented by properties s, and Π_{p_s} the set of projection elements of p concerning the input stream represented by s. Then, for a subscription that only contains selection and projection operators, $\overline{size}(p)$ is calculated using the following formula:

$$\overline{size}(p) := \sum_{s \in P_q} \left(\overline{size}(s) - \sum_{n_s \notin \Pi_{p_s}} \left(\overline{occ}(n_s) \cdot \overline{size}(n_s) \right) \right)$$

Note that, in the above formula, $\overline{size}(p)$ denotes the average size of one data stream item in the stream represented by p, e. g., one `photon` element in stream `photons`, whereas $\overline{size}(n_s)$ denotes the average size of one subelement n_s, e. g., the `phc` subelement of a `photon`. For aggregate queries, the result data stream is a stream of aggregate result values. The average result data stream size is therefore independent of the input stream size in this case and is computed as the size of the computed aggregate values and their surrounding element tags. For queries returning the contents of data windows, the average size of a data window needs to be determined. For item-based data windows this can be done by multiplying the window size with the average size of the items contained in the window and adding the sizes of the enclosing window tags. For time-based data windows this works analogously except that the average number of data items contained in a window must be estimated as the product of the input stream frequency and the window size.

The average frequency of data items in the stream represented by p is denoted by $\overline{freq}(p)$. With $sel(\sigma_p)$ denoting the selectivity of the subscription represented by p, $\overline{freq}(p)$ can be computed as follows:

$$\overline{freq}(p) := sel(\sigma_p) \cdot \sum_{s \in P_q} \overline{freq}(s)$$

Note that the expression $\sum_{s \in P_q} \overline{freq}(s)$ in this formula depends on the semantics of the employed operators in q. The above formula is valid for selection operators. Projection operators do not influence $\overline{freq}(p)$. For window-based queries, $\overline{freq}(p)$ depends on the step defined for the data window and the average frequency of the input data stream. For item-based data windows, $\overline{freq}(p)$ corresponds to the frequency of the corresponding input data stream divided by the step size μ of the data window. For time-based data windows, $\overline{freq}(p)$ depends on the distribution of the values of the reference element. To be able to estimate the frequency of the result data stream in such a case, we keep track of the average increment of the reference element value between two successive data items arriving in the stream. Dividing the step size μ of the time-based data window by this average increment yields the average number of data items that need to be read from the stream before the window update is complete. Then, as with item-based data windows, the frequency of the input data stream is divided by this estimated number of data items to obtain the estimated average frequency of the result data stream.

Introducing $b(e)$ as the maximum bandwith of a network connection e, we can characterize the relative amount $u_b(e)$ of bandwith of e used by the additional data streams routed over e for answering q using the following formula:

$$u_b(e) := \frac{\sum_{p \in P_e} \left(\overline{size}(p) \cdot \overline{freq}(p) \right)}{b(e)}$$

Here, P_e denotes the set of properties of all additional data streams added over e to answer q.

The average computational load caused by an operator o on a peer v with a set of input stream properties P_o is denoted $\overline{load}(o, v, P_o)$. The maximum load of a peer v is represented by $l(v)$. The relative amount $u_l(v)$ of computational load on a peer v caused by the additional operators in O_v installed at v for answering a new subscription can be computed as follows:

$$u_l(v) := \frac{\sum_{o \in O_v} \overline{load}(o, v, P_o)}{l(v)}$$

Cost function inputs like average frequencies of data stream items, average sizes and occurences of elements, and selectivities of operators are obtained from statistics and selectivity estimations. The average load $\overline{load}(o, v, P_o)$ of an operator o on a peer v with input stream properties P_o depends on the

performance of the executing peer, expressed by a performance index ($pindex(v)$), and the characteristics of the operator itself. For example, assuming a linear dependency of the load caused by a selection operator σ from the frequency $\overline{freq}(s)$ of its only input stream s, the average load caused by σ on a peer v can be defined as $\overline{load}(\sigma, v, s) := bload(\sigma) \cdot pindex(v) \cdot \overline{freq}(s)$. Here, $bload(\sigma)$ represents a base load factor for the selection operator. Factors like base loads of operators and performance indices of peers as well as formulas for combining these factors yielding realistic load estimations have to be determined, e. g., on the basis of reference values.

The cost function C is then defined as follows:

$$C(\mathcal{P}) := \gamma \cdot \left(\sum_{e \in E_\mathcal{P}} \left(u_b(e) + \max(0, (u_b(e) - a_b(e))) \cdot e^{(u_b(e) - a_b(e))} \right) \right) + (1 - \gamma) \cdot \left(\sum_{v \in V_\mathcal{P}} \left(u_l(v) + \max(0, (u_l(v) - a_l(v))) \cdot e^{(u_l(v) - a_l(v))} \right) \right)$$

In this function, $\mathcal{P}$ denotes the evaluation plan of the new subscription, i. e., the operators that have to be installed, the peers on which they have to be installed, and the additional data streams that are generated and routed through the network. Furthermore, $E_\mathcal{P}$ is the set of network connections and $V_\mathcal{P}$ is the set of peers affected by plan $\mathcal{P}$. A weighting factor $\gamma \in [0, 1]$ determines, which part of the cost function should be more dominant—network traffic or peer load. An exponential penalty is given for overload situations on peers and network connections. The relative amount of available bandwith on network connection e and of available computational load on peer v is represented by $a_b(e)$ and $a_l(v)$, respectively. A plan $\mathcal{P}$ is better than another plan $\mathcal{P}'$ according to cost function C, expressed by $\mathcal{P} \prec_C \mathcal{P}'$, if and only if $C(\mathcal{P}) < C(\mathcal{P}')$.

4.4 Stream Sharing Algorithms

We now describe our stream sharing algorithms for registering and efficiently satisfying new continuous queries in P2P data stream management systems. The algorithms search for shareable data streams in the network, compare the properties of new subscriptions with those of existing data streams, and decide which streams to reuse at which peers.

4.4.1 Query Registration

The algorithm for continuous query registration searches for shareable data streams in the network and decides if a certain available data stream can actually be shared for answering a new query by comparing the corresponding properties. Further, it decides whether a newly found evaluation plan for the new query is better than the previously best plan.

The algorithm is divided into four parts. The OptimizeQuery algorithm, which is shown in Algorithm 1, describes the discovery of shareable data streams and the generation of corresponding query evaluation plans. The MatchProperties and MatchPredicates algorithms which are detailed in Algorithms 2 and 3 handle the matching of properties and predicates, respectively. Finally, the matching of aggregation operators is dealt with in the MatchAggregations algorithm shown in Algorithm 4. Beginning with Algorithm 1, the inputs p_q and v_q are the properties of the new subscription q and the network node where q is registered, respectively. The output of the algorithm is the evaluation plan $\mathcal{P}$, describing how the network has to be changed in terms of installed operators and routed data streams in order to satisfy q. Note that there will always be at least one plan that is suitable for answering q—provided that q refers to existing inputs—namely the plan using the originally registered versions of q's input streams. The goal of our approach is to find possibly transformed versions of these streams—generated by projection, selection, or aggregation operators in the network for answering other continuous queries—that can also be used to answer q, possibly by applying some further transformations.

Algorithm 1 starts with an empty plan $\mathcal{P}$ (line 1) and iterates over all input data streams of q (line 2). For each such input data stream, the algorithm performs some initialization tasks (lines 3–6). First, a FIFO queue L_V for network nodes (peers) and another queue L_P for properties are initialized. Then, the properties p_s of the currently considered input data stream s and the network node where this input data

Algorithm 1 OptimizeQuery

Input: The properties p_q of the subscription q to be registered and the node v_q where q is to be registered.
Output: A query evaluation plan $\mathcal{P}$.

```
1: P ← ∅;
2: for all p_s ∈ getInputDS(p_q) do
3:    L_V ← ∅; L_P ← ∅;
4:    p_b ← p_s; v_b ← getTNode(p_b);
5:    P_s ← generatePlan(p_b, v_b, v_q);
6:    add(L_V, v_b);
7:    while L_V ≠ ∅ do
8:       v ← dequeue(L_V); mark(v);
9:       for all data streams available at v that are variants of p_s do
10:         enqueue all associated properties in L_P;
11:      end for
12:      while L_P ≠ ∅ do
13:         p ← dequeue(L_P);
14:         if MatchProperties(p, p_s) then
15:            n ← getTNode(p);
16:            if (¬(isMarked(n)) ∧ (n ∉ L_V)) then
17:               add(L_V, n);
18:            end if
19:            P'_s ← generatePlan(p, v, v_q);
20:            if P'_s ≺_C P_s then
21:               p_b ← p; v_b ← v; P_s ← P'_s;
22:            end if
23:         end if
24:      end while
25:   end while
26:   unmark all nodes;
27:   add(P, P_s);
28: end for
29: return P;
```

stream is registered are stored in p_b and v_b, respectively. The variables p_b and v_b represent the properties of the currently best solution for the data stream chosen as input for satisfying q and the network node where to find and reuse that stream. Installing the whole new subscription at the super-peer at which it is registered and using the original input streams, routed to the subscription via shortest paths in the network, is set as the initial evaluation plan. Therefore, the part of the query evaluation plan that deals with input stream s, called $\mathcal{P}_s$, is initially set to routing s from the peer where s is registered to the peer where q is registered via a shortest path in the network and performing any query evaluation tasks on s at the target peer. This plan is generated by means of the *generatePlan* function that takes as inputs the properties p_b of the data stream chosen for reuse, the node v_b where to reuse that stream, and the node v_q where the query to be answered is registered and where the query result is needed. At each time during the remaining execution of the algorithm, the current best plan for input data stream s is represented by $\mathcal{P}_s$. Note that the initial plan does not reuse any existing data streams in the network. Finally, the start node v_b of the search in the network is added as first node to L_V.

If a subscription references more than one input stream, each stream is handled individually by the subscription algorithm. The algorithm assures that at least the relevant parts of each input stream are delivered to the super-peer connected to the peer that registered q. Any combination of input data streams as demanded by the subscription is performed at this peer during the final post-processing step and the result of this combination is not considered for reuse in the network. This is the same as with any restructuring of the query result as described in Section 3.

After the initialization, the algorithm basically performs a breadth-first search in the network graph for each input stream, starting at the node that corresponds to the super-peer at which the corresponding original input stream of q is registered. Using LIFO queues for L_V and L_P instead of FIFO queues would cause the algorithm to perform depth-first search which would be equally possible. The peers in L_V are dequeued one after another (line 8). Each peer in L_V is marked in order to handle circles in the network graph, i. e., consider each node at most once. For each dequeued peer, all unmarked properties of data streams that are available at the currently handled peer and that are variants of p_s are subsequently inserted into L_P (lines 9–11). These properties are then consecutively taken out of the queue and matched against the properties p_q of q using Algorithm 2 (lines 12–14). This will be described in detail in Section 4.4.2 below. Network connections that do not have any associated properties because they do not carry any data streams are ignored during the breadth-first search. Also, non-matching properties do not add any peers to L_V since following these paths cannot yield a reusable data stream. Pruning the search in this way leads to the breadth-first search traversing only the relevant part of the network instead of the whole network.

If a property p has been successfully matched, its corresponding stream can be reused for answering q. If the target peer of p, i. e., the peer to which the stream corresponding to p is delivered, is still unmarked, it is added to L_V to be processed later on during the breadth-first search (lines 15–18). For any found solution, a new plan $\mathcal{P}'_s$ is generated, again using the *generatePlan* function (line 19). Then, the value of the cost function C for the plan reusing the found data stream is computed and compared against the current best solution (line 20). Only if the new solution is better according to C, it replaces the current best solution and is stored along with its cost function value for future comparisons (lines 20–22). When there are no properties left in queue L_P, the next node of L_V is considered. If there are no more nodes left in L_V, the best plan $\mathcal{P}_s$ found for input stream s is added to the overall plan $\mathcal{P}$ for evaluating q (line 27). When all input streams of q have been considered, the algorithm terminates and returns the current best solution for plan $\mathcal{P}$ as the final result.

The termination of the algorithm is guaranteed since there is only a finite number of input data streams of q and of nodes and data streams in the network. For each input data stream, each node can be added to L_V at most once, and each time through the while-loop in line 7 of the algorithm one node gets dequeued from L_V. Similar considerations apply to properties of data streams and L_P.

4.4.2 Matching Properties

Next, we explain how Algorithm 2 matches properties. For each input data stream of a subscription, the properties of the subscription reflect what operators and operator conditions are employed to transform the respective input stream into the subscription result. These properties have to be matched with the properties of data streams already present in the network to find shareable streams for each input stream of the new subscription. The inputs for the properties matching algorithm are the properties p of the data stream that is considered for reuse and the properties p' of the newly registered subscription. The algorithm returns true if these properties match and false otherwise. If the input streams of both properties match—checked in lines 1–4 of Algorithm 2—the operators used to transform the inputs are fetched from the properties data structures (line 5) and assigned to O and O', respectively. For each operator in O, there must be a corresponding operator in O'. For example, if O contains a selection operator, the data stream represented by p is only considered for reuse if p' also contains a corresponding selection. Otherwise, the stream of p would not contain all the data needed by q. If a corresponding operator is found in O', it has to be assured that the conditions of the two operators, which are fetched from the properties data structures in line 10 of the algorithm, are compatible. We distinguish four cases (lines 11–30), i. e., selection, projection, window-based aggregation, and unknown operators. If the corresponding operators are selection operators (lines 11–15), the algorithm retrieves the graphs representing the selection predicates (line 12) and tries to match them using Algorithm 3. In case of a projection (lines 16–20), the set $\overline{R}$ of elements that are actually returned in the result data stream of the query represented by p—these are the projection elements marked with bullets in the properties of the example queries in Figures 13 to 16—has to be a superset of the set R' of all the elements referenced in the query—these appear as unmarked elements in the projection operator conditions of properties—in order for the stream represented by p to be reusable. If o and o' are one of the window-based aggregation operators `min`, `max`, `sum`, `count`, or `avg`, it has to be assured that the conditions and data windows are

Algorithm 2 MatchProperties

Input: The properties p of a data stream to be reused and p' of a subscription to be registered.
Output: true if p and p' match, false otherwise.

```
 1: s ← getDS(p); s' ← getDS(p');
 2: if s ≠ s' then
 3:   return false;
 4: end if
 5: O ← getOps(s); O' ← getOps(s');
 6: for all o ∈ O do
 7:   match ← false;
 8:   for all o' ∈ O' do
 9:     if o = o' then
10:       C ← getConds(o); C' ← getConds(o');
11:       if o = σ then
12:         G ← getPGraph(C); G' ← getPGraph(C');
13:         if MatchPredicates(G, G') then
14:           match ← true; break;
15:         end if
16:       else if o = Π then
17:         R̄ ← getOutElems(C); R' ← getRefElems(C');
18:         if R̄ ⊇ R' then
19:           match ← true; break;
20:         end if
21:       else if o ∈ {min,max,sum,count,avg} then
22:         if MatchAggregations(C, C') then
23:           match ← true; break;
24:         end if
25:       else
26:         i⃗ ← getParams(C); i⃗' ← getParams(C');
27:         if i⃗ = i⃗' then
28:           match ← true; break;
29:         end if
30:       end if
31:     end if
32:   end for
33:   if match = false then
34:     return false;
35:   end if
36: end for
37: return true;
```

compatible (lines 21–24). This is done by the MatchAggregations algorithm described further below. All other operators are handled in the fourth and final case (lines 25–30). These are unknown operators, in particular user defined functions. Nothing is known about the semantics of these operators. We only require them to be deterministic, meaning that the same operators applied on the same inputs must always yield the same results. The algorithm then demands that not only the operators but also their input vectors, i. e., their parameter lists retrieved in line 26 of the algorithm, are the same for reusability. More sophisticated techniques for identifying reusable user defined operators involve the development of suitable operator descriptions and are the subject of future work.

Algorithm 3 MatchPredicates

Input: The predicate graphs G of a data stream considered for reuse and G' of a new subscription to be registered.

Output: true if the predicates of G match the predicates of G', false otherwise.

```
1: for all v ∈ V do
2:   vmatch ← false;
3:   for all v' ∈ V' do
4:     if v ≙ v' then
5:       vmatch ← true;
6:       for all x ∈ {e ∈ E|e connected to v} do
7:         ematch ← false;
8:         for all y ∈ {e' ∈ E'|e' connected to v'} do
9:           if ζ(x) ⇚ ζ(y) then
10:            ematch ← true; break;
11:          end if
12:        end for
13:        if ematch = false then
14:          return false;
15:        end if
16:      end for
17:      break;
18:    end if
19:  end for
20:  if vmatch = false then
21:    return false;
22:  end if
23: end for
24: return true;
```

4.4.3 Matching Predicates

A predicate is represented by a weighted directed graph $G = (V, E)$ within the corresponding properties. The construction and representation of predicate graphs are an extension of related work on the processing of conjunctive predicates [RH80]. In addition to integer valued variables and constants, we also allow decimal values with a finite number of decimal places. First, predicates are normalized to contain only comparisons of the form $\$v \geq c$, $\$v \leq c$ and $\$v \leq \$w + c$ where $\$v$ and $\$w$ represent variables and c represents a constant integer or decimal value. Each variable in the predicate becomes a node in V and an atomic predicate of the form $\$v \leq \$w + c$ is represented by a weighted directed edge in E from node $\$v$ to node $\$w$ with weight c. Further, V contains a node for the constant zero. An atomic predicate of the form $\$v \leq c$ is represented by an edge from node $\$v$ to node zero with weight c and an atomic predicate of the form $\$v \geq c$, which can be expressed as $0 \leq \$v - c$, by an edge from node zero to node $\$v$ with weight $-c$. As illustrating examples, consider Figures 13 to 16 which contain the predicate graphs of the selections in Queries 1 to 4. After the construction of G, the predicate can be checked for satisfiability and is minimized using techniques introduced in earlier related work [RH80]. If an operator's predicate is unsatisfiable, the corresponding subscription can be rejected. A minimized predicate does not contain any redundant atomic predicates. Note that the construction of the properties together with all the steps described in this paragraph is performed only once for each new subscription during registration.

The MatchPredicates algorithm shown in Algorithm 3 can match any predicates in the described graph representation, e. g., selection and join predicates. In this paper, it is used to match the predicates of selection operators. The algorithm takes the data structures G and G' of the weighted directed graphs representing the selection predicates of the existing data stream and the new subscription which are to be compared and returns true if the predicates of G' imply those of G, i. e., reusability of the data stream is not prevented by the predicates. One prerequisite for the possibility of data stream sharing is that, for

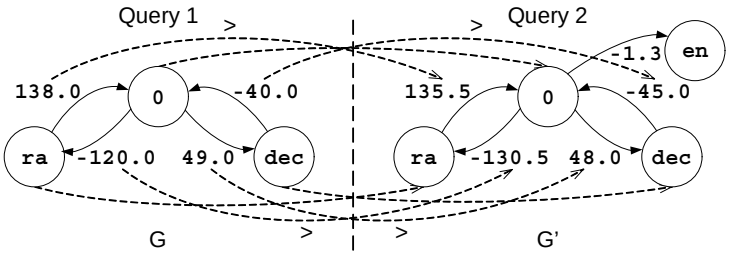


Figure 18: Matching Predicates

each node v in the node set V of G, there exists an equivalent node v' in the node set V' of G', denoted $v \hat{=} v'$ in line 4 of Algorithm 3. Nodes are equivalent if the variables represented by them refer to the same element. Furthermore, if two equivalent nodes v and v' have been found, for each edge x connected to v there must be an edge y connected to v' such that the atomic predicate represented by x, denoted $\zeta(x)$, is compatible with the atomic predicate represented by y, denoted $\zeta(y)$. In our algorithm, this is the case if $\zeta(x) \Leftarrow \zeta(y)$ in line 9. An example matching for the predicate graphs of Queries 1 and 2 is shown in Figure 18. For brevity, only the variable names instead of the full paths are shown as node labels in the figure. The definition of $\zeta(e)$ for any edge e in a predicate graph G can be formally expressed as

$$\zeta(e) := (sourcelabel(e) \leq targetlabel(e) + weight(e))$$

where $sourcelabel(e)$ and $targetlabel(e)$ denote the absolute path to the variable represented by the source and the target node of edge e, respectively, and $weight(e)$ denotes the weight of edge e.

4.4.4 Query Evaluation Plans

The query evaluation plans that are constructed during optimization are represented as hierarchical XML documents specifying which actions, e. g. installations of operators, have to be performed at which peers in the network. The installation of the best plan at the end of the optimization starts at the peer that registered the query by distributing the relevant subplans to all other affected peers in the network asynchronously. After that, the installation of the operators specified in the subplans on their respective peers starts in the opposite direction, i. e., first the operator closest to the data source is installed. When a peer has installed the new local operators of its new subplan it notifies the next peer in the plan to install its local operators in a synchronous fashion, eventually leading to a completely installed and executing plan. An example query evaluation plan for installing Query 2 from Section 1 is shown in Appendix C on page 43.

4.4.5 Examples

Let us now consider Queries 1 and 2 of Section 1 as illustrative examples for the above described algorithms. We start with the network topology of Figure 1. We further assume, that stream `photons` has already been registered in the network and is available at super-peer SP_3. Note that it suffices to consider the super-peer backbone network in the algorithm as the thin-peers are only the start and end points of data streams but do not transform any streams.

Example 4.1 (Query 1) When Query 1 is registered, the algorithm first constructs the corresponding properties of the query including the minimized weighted directed graphs of the selection predicates. The only peer in the network that has a reusable stream is SP_4 and the only reusable stream is the originally registered stream `photons`. Consequently, the selection and projection operators of Query 1 are installed at SP_4 and the result is routed to P_1 using the shortest path which is via SP_5 and SP_1. □

Example 4.2 (Query 2) Query 2 is registered at P_2 after Query 1 has been registered. At this point in time, the original stream `photons` is available at SP_4 and the stream filtered by the projection and selection of Query 1 is available at SP_4, where it is generated, at SP_5, and at SP_1. The algorithm finds out that the filtered stream is suitable for answering Query 2 because the set of projection elements that are

returned by Query 1 is a superset of the set of elements referenced in Query 2 and the atomic predicates of the minimized selection predicates in Query 1 are all implied by corresponding atomic predicates of the minimized selection predicates in Query 2, as can be seen from Figure 18. Note that, in this example, the original predicates are already minimized since they do not contain any redundant atomic predicates. Altogether, four possible solutions for reusing a stream to answer Query 2 are identified by the algorithm. These include the original stream `photons` at SP_4 as well as the filtered stream generated for answering Query 1 at SP_4, at SP_5, and at SP_1. Reusing the filtered stream at SP_5 yields the lowest value for cost function C. Therefore, this stream is duplicated at SP_5 and a copy, after the necessary additional filtering is done by installing the projection and selection operators of Query 2 at SP_5, is routed to P_2, again using a shortest path which is via SP_7. □

4.5 Window-based Aggregation

Reusing results of window-based aggregation operators has been studied before [AW04]. Our approach differs from this previous solution in two ways. First, we introduce a step in our windows which allows us to explicitly specify when a new aggregate value shall be computed. Second, we consider existing results of other subscriptions for reuse instead of precomputing aggregation results that might never be used. As usual, we categorize aggregation operators using three classes. These classes are distributive (e. g., `min`, `max`, `sum`, `count`), algebraic (e. g., `avg`), and holistic aggregates (e. g., `quantile`). We concentrate on the above mentioned distributive and algebraic aggregation operators here.

The MatchAggregations algorithm shown in Algorithm 4 is used in Algorithm 2 to compare the conditions of window-based aggregation operators. Such operators are compared by examining their input data, their results, and their data windows as follows. First, MatchAggregations checks whether the aggregate considered for reuse and the new aggregate employ the same aggregation operator, are based on the same input data, and aggregate the same element. Furthermore, selections in aggregate subscriptions have to be handled more strictly than in other subscriptions. It has to be assured that any selection performed on the aggregated data stream prior to the aggregation is the same in both the reused and the new aggregate subscription. Second, it is checked whether the aggregation result which is considered for reuse has been filtered in any way. As an example consider Query 4 which filters its aggregation result `$a` using the predicate `$a > 1.3`. Reusing such aggregate values for computing more coarse-grained window aggregates is not possible in general since a part of the necessary data might have been filtered

Algorithm 4 MatchAggregations

Input: The conditions C of the operators contained in the properties of a data stream to be reused and the conditions C' of the operators contained in the properties of a new subscription to be registered.
Output: true if the aggregations of C match the aggregations of C', false otherwise.

1: $o \leftarrow getAggOp(C)$; $o' \leftarrow getAggOp(C')$;
2: $s \leftarrow getDS(C)$; $s' \leftarrow getDS(C')$;
3: $e \leftarrow getAggElem(C)$; $e' \leftarrow getAggElem(C')$;
4: $W \leftarrow getWindow(C)$; $W' \leftarrow getWindow(C')$;
5: **if** $o = o'$
$\wedge\, s = s'$
$\wedge\, e = e'$
$\wedge\, \sigma(g) = \sigma(g')$
$\wedge\, ref(W) = ref(W')$
$\wedge\, \Delta' \; mod \; \Delta = 0$
$\wedge\, \Delta \; mod \; \mu = 0$
$\wedge\, \mu' \; mod \; \mu = 0$
$\wedge$ MatchPredicates($getPGraph(C), getPGraph(C')$) **then**
6: **return** true;
7: **end if**
8: **return** false;

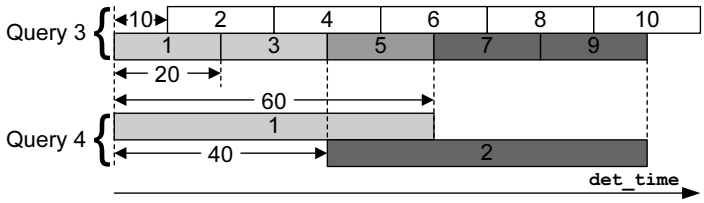


Figure 19: Reusing window-based aggregates

out. However, they can still be reused for aggregates that apply the same or a more restrictive filter on the aggregation result as long as all other prerequisites for reusability are fulfilled.

Eventually, the data windows of both operators are examined. For time-based windows, reuse is only possible if both windows have the same ordered reference element, e. g., `det_time` in Queries 3 and 4. For both, time-based and item-based windows, we require the window size and the step size of the windows to be compatible for being able to reuse existing aggregate values without any further complex optimizations or transformations. One requirement for this is that the window size of the new subscription is a multiple of the window size of the data stream considered for reuse. This guarantees that a fixed number of reused windows fits into one new window. Furthermore, the window size of a reused aggregate's data window must be a multiple of its step size. This assures that a sequence of non-overlapping windows, i. e., aggregate values, covering the whole input data can be obtained—possibly by ignoring some windows. Note that ignored aggregate values might have to be temporarily buffered to be reused for computing subsequent values of the new aggregate. The situation for the step sizes of both windows is analogous to their window sizes as described above, guaranteeing that the reused aggregate delivers an aggregate value at least each time the new aggregate has to produce one. Formally, these three conditions for data window reusability can be formulated as follows.

- $\Delta' \bmod \Delta = 0$
- $\Delta \bmod \mu = 0$
- $\mu' \bmod \mu = 0$

Note that for the values of `avg` aggregates to be shareable, we internally represent such aggregates by their appropriate `sum` and `count` values. These values are actually transmitted in the super-peer network. The final aggregate value is computed at the super-peer at which the corresponding subscription is registered by evaluating `(sum/count)`. The described internal representation of `avg` aggregates also enables their reuse for computing `sum` and `count` aggregates, i. e., the requirement of equal aggregate operators for shareability introduced above can be relaxed.

Example 4.3 (Queries 3 and 4) As an example illustrating how window-based aggregates are handled by our algorithm consider Queries 3 and 4 as introduced in Section 3. We assume the network of Figure 1 with Queries 1 and 2 already registered as described earlier. Query 3, which can reuse the result data stream of Query 1, is registered at peer P_3 in the network and computes the average energy of all photons detected in a certain area of the sky. The time-based data window has a size of 20 time units and every 10 time units a new aggregate value is computed. Furthermore, Query 3 does not filter the result values of the aggregation in any way. Query 4 is another aggregate query that employs the same aggregation operator, references the same input data stream, aggregates the same element, and uses the same selection predicate as Query 3.

Obviously, in terms of cost function C, reusing the result data stream of Query 3 at SP_3 is the best solution for answering Query 4, provided that reuse is possible. In order to determine shareability, the data windows of both subscriptions need to be compared. The situation is illustrated in Figure 19. We compute $60 \bmod 20 = 0$ for the window size and $40 \bmod 10 = 0$ for the step size of the windows as well as $20 \bmod 10 = 0$ for the window size and the step size of the result data stream of Query 3, meaning that reuse is possible. Since $60 \operatorname{div} 20 = 3$ holds, three consecutive non-overlapping windows of Query 3 are needed to form a window of Query 4. Because of $20 \operatorname{div} 10 = 2$, only every second aggregate value

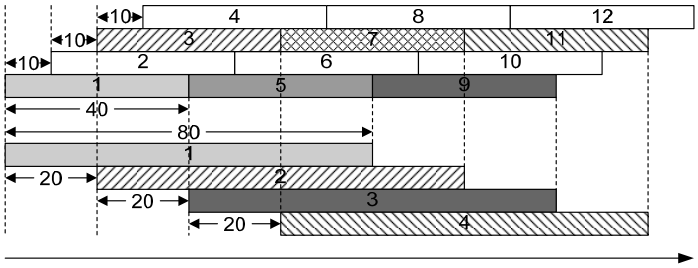


Figure 20: Window selection for reusing window-based aggregates

of Query 3 is to be reused for Query 4. Eventually, 40 div 10 = 4 indicates that each time four values of Query 3 have been seen, only two of which have been reused, a new aggregate value of Query 4 is computed. □

For being able to reuse aggregate values of previously computed window-based aggregation operators to compute more coarse-grained aggregates, we have developed an operator for selecting the appropriate values in the course of query evaluation in the FluX [KSSS04] query engine. FluX is a query engine for efficiently processing (W)XQueries on XML data streams. Since, depending on the window definition, not all aggregate values of an existing aggregate are needed and the values are not necessarily needed in the same order as they appear in the reused stream, the appropriate elements have to be selected and also buffered and reordered if necessary. The algorithm for selecting aggregate values is shown in Algorithm 5. Consider the following example.

Example 4.4 (Reusing Aggregate Values) We assume a stream data window with window size $\Delta = 40$ and step size $\mu = 10$. The query window has a window size of $\Delta' = 80$ and a step size of $\mu' = 20$. The two data windows are illustrated in Figure 20. Algorithm 5 starts with buffering the first $((\Delta' - \Delta) \text{ div } \mu) + 1 = ((80 - 40) \text{ div } 10) + 1 = 5$ aggregate values arriving in the stream. It then sends the aggregate values at buffer positions $i \cdot (\Delta \text{ div } \mu)$ for $0 \leq i < (\Delta' \text{ div } \Delta)$ to FluX. Since $\Delta' \text{ div } \Delta = 80 \text{ div } 40 = 2$ and $\Delta \text{ div } \mu = 40 \text{ div } 10 = 4$, these are the values at buffer positions $0 \cdot 4 = 0$ and $1 \cdot 4 = 4$. After that, the first $\mu' \text{ div } \mu = 20 \text{ div } 10 = 2$ values are removed from the buffer and the next 2 values are read from the stream and added to the buffer. After updating the buffer, the values needed for computing the next window aggregate value reusing the values in the buffer can be determined as above. □

The reusing query that is executed in FluX uses the values delivered by the above algorithm as input. It aggregates these input values using the appropriate aggregate function and an item-based data window with equal window and step size, both set to $\Delta' \text{ div } \Delta$.

Note that Algorithm 5 buffers all aggregate values arriving in the input stream. This can be avoided by exactly identifying the aggregate values that need to be buffered and immediately discarding the

Algorithm 5 SelectAggregateValues

Input: Window sizes Δ and Δ' as well as step sizes μ and μ' of a data window to be reused and a new data window, respectively.

Output: The correct sequence of aggregate values for reuse.

1: buffer first $((\Delta' - \Delta) \text{ div } \mu) + 1$ aggregate values arriving in the stream;
2: $i \leftarrow 0$;
3: **while** $i < (\Delta' \text{ div } \Delta)$ **do**
4: send value at buffer position $i \cdot (\Delta \text{ div } \mu)$ to the FluX query engine;
5: $i \leftarrow i + 1$;
6: **end while**
7: remove first $(\mu' \text{ div } \mu)$ values from buffer and read next $(\mu' \text{ div } \mu)$ values from stream into buffer;
8: continue in line 2 above until buffer contains no more values;

others. The resulting alternative algorithm is shown in Appendix D on page 46. Since the alternative algorithm is more complex to evaluate, we use the algorithm introduced above in StreamGlobe.

4.6 Extensions and Optimizations

On the basis of the algorithms of Section 4.4, we now introduce some further extensions and general optimizations improving the quality and the efficiency of our solution. All of these extensions and optimizations have been implemented in our StreamGlobe prototype.

4.6.1 Bypassing

The result of any subscription evaluation in the network is routed towards the receiving peer via a shortest path in the network. In order to avoid congested network connections and overloaded peers, we introduce a simple bypassing mechanism, thus increasing the search space of our algorithm. Whenever a plan is discovered to cause an overload situation on any network connections or peers, a new internal network graph representing the original network without the overloaded connections and peers is constructed. Then the plan is modified to route its data over shortest paths within this reduced network. This can be repeated multiple times until no overload occurs or the reduced network does not contain any valid paths to the target peer any more. Each plan found during this procedure is compared against the current best plan as described above.

A disadvantage of this solution is that the shortest path algorithm needs to be executed multiple times if an overload situation is predicted. Furthermore, the approach can lead to network partitioning in the reduced network, making it impossible to find an overload-free evaluation plan although one exists. This can be avoided by using an alternative bypassing scheme which computes appropriate weights for each network connection and then uses a shortest path algorithm to find the weighted shortest path between two peers in the network. In this case, the shortest path algorithm needs to be executed only once during the generation of a query evaluation plan. The weight of a network connection can be computed by determining weights based on the current amount of network traffic and peer load on the respective network connection and its two incident peers, adding the peer weights to the weight of the network connection. A disadvantage of this scheme is, however, that the weights of network connections and peers need to be updated each time the network state changes. If the number of iterations needed to find a plan without overload in the first approach is low, i. e., only one iteration on average, then the first approach is supposed to be more efficient than the second approach. In both bypassing solutions, if no plan without overloaded network connections and peers can be found, the corresponding query can be rejected by the system. We have implemented both bypassing schemes in StreamGlobe and use the first one by default.

4.6.2 Optimized Loop Computation

The loops in the algorithms of Section 4.4 iterate over sets of peers, vertices, edges, properties, etc. Some of these sets contain rather few items in practice, e. g., number of operators in a query, vertices and edges in a predicate graph, and therefore also yield a small number of loop iterations. Additionally, many loops can be exited early, e. g., as soon as a match is found—indicated by the `break` statements in the algorithms. Some loop computations can be optimized by employing an execution similar to merge-joins. For example, the first two loops in Algorithm 3 can be executed in a merge-join fashion if the vertices in V and V' are sorted lexicographically according to their labels, i. e., according to the paths they represent.

4.6.3 Caching Matching Results

Routing a data stream through the network via several peers without transforming it leads to identical data streams and data stream properties being available at many different peers in the network. The basic algorithm of Section 4.4 does not take this into account when searching for shareable streams and matches each of the identical properties anew. Furthermore, the algorithms make no difference between incoming and outgoing data streams at a certain peer. This leads to each data stream property being considered twice, once at the source peer and once at the target peer of the corresponding stream. Both

problems can be avoided by identifying identical versions of already matched properties and reusing the corresponding cached matching result.

4.6.4 Exploiting Local Matches

A special case occurs when two or more—in terms of our properties data structure—identical subscriptions are registered at peers that are connected to the same super-peer in the network. This might easily occur in a multi-user network where several users have the same interests and register their continuous queries at the same point in the network. Using the basic algorithm, each of those queries would be matched starting at the super-peers where the input data streams of the query are registered and then traversing the network using breadth-first or depth-first search as described in Section 4.4.1. However, in each case, the result would be to reuse the already present answer to the subscription at the super-peer connected to the subscribing client as this will obviously yield the lowest value for cost function C. The situation can easily be improved by checking a new subscription's super-peer for the presence of reusable streams prior to executing the actual query subscription algorithm. The approach could even be extended to checking the properties of data streams available at peers in the neighborhood of a subscribing peer. This could be done either by checking the neighboring peers or, for larger networks where a larger neighborhood should be considered, by flooding the network with a data stream request and using an adaptable horizon for the flooding depth.

5 Implementation

In this section, we will give a brief overview of the StreamGlobe system architecture, which is depicted in Figure 21.

Basically, all components of StreamGlobe which are represented by rounded rectangles in Figure 21 are implemented as collaborating grid services on top of the Globus Toolkit [Glo05]. According to the classification of peers, the provided services are shown individually for each type. The grid services are divided into two groups as follows. *Interface services* represent the different peers in the network. These services constitute the interfaces between peers and users as well as between peers and other parts of the StreamGlobe system. Users register subscriptions and data streams at these interfaces. Non-XML data streams are fed into the system by means of wrappers which are executed at the corresponding peer to convert the given data format into XML. Furthermore, these interface services receive control messages from other peers and forward them to the appropriate *core service*. The set of core services of a peer comprises all the functionality of that peer. Every peer runs exactly one instance of an interface service and a set of core services according to its capabilities.

Thin-peers are peers with limited functionality. They publish data streams in the network and/or receive the results of their subscriptions, but do not carry out complex query processing. Since even such a peer has to provide metadata, e. g., statistics of a data stream needed for optimization, it mainly runs the metadata management service. Beyond that, only the plan distribution service and a minimum query execution service are provided. The plan distribution service is needed for all peers, since this component is responsible for correctly setting up data communication with other peers and instantiating the queries in the query execution service. The query execution service of a thin-peer is only able to display results of subscriptions, to publish—and possibly wrap—data streams, and to maintain statistics, if needed. In addition to the components of thin-peers, super-peers provide extensive query processing capabilities by enabling full-fledged query processing for data streams. For carrying out query processing tasks, the query execution service employs the extensible *FluX* [KSSS04] query engine for efficiently processing XML data streams, which is installed on every super-peer. FluX is an event-based query engine that achieves buffer minimization through optimizations based on schema information of data streams. It is therefore applicable for efficient stream processing. Furthermore, user defined stream operators which are implemented as mobile code can be executed as mobile extensions of the FluX engine. This provides for a large amount of flexibility in specifying subscriptions. Finally, speaker-peers are basically super-peers with the additional role of optimizing and managing their subnets. Hence, speaker-peers provide an optimization service for carrying out the optimization tasks as described in Section 4.

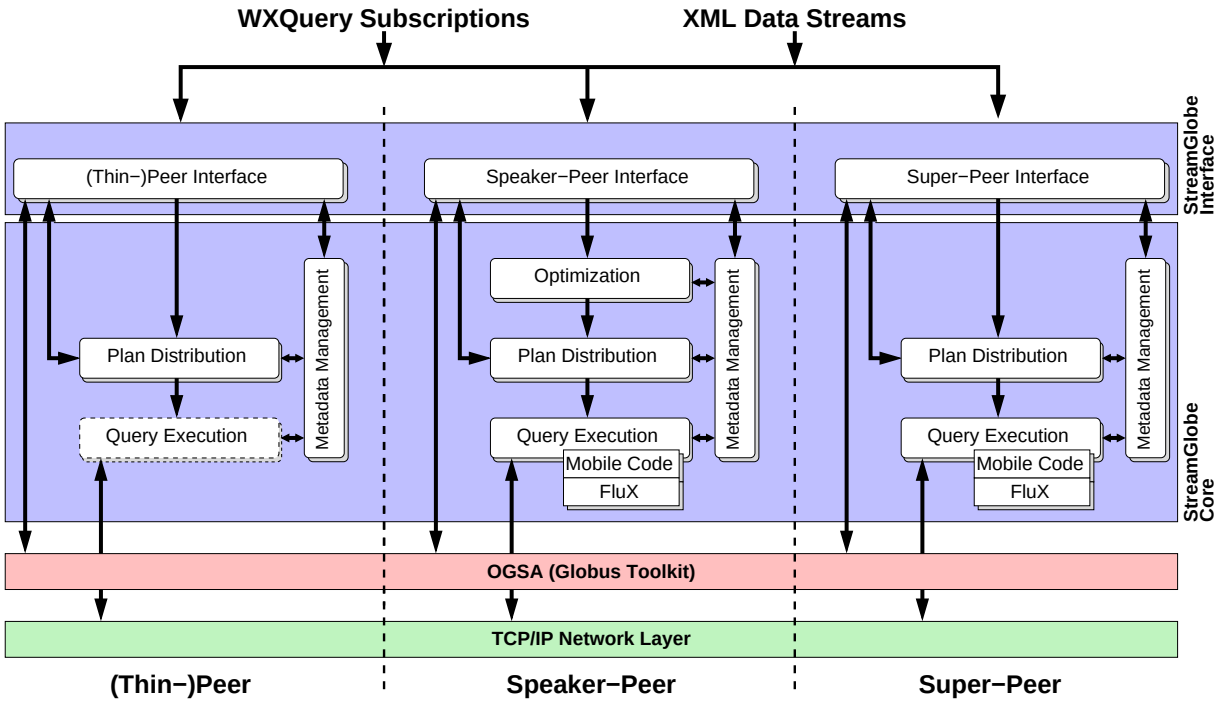


Figure 21: Peer architecture overview

In Figure 21, the communication paths are depicted by arrows. Different StreamGlobe components and services on a single peer communicate directly via mechanisms provided by the Globus Toolkit. Inter-peer communication takes place between the interface services of the two communicating peers using the RPC mechanisms of Globus. As the OGSA framework does not yet provide any suitable means for data stream transfer, we realized our own protocols based on TCP/IP networking techniques for direct data exchange between query execution services.

The algorithms as introduced in Section 4, like all other parts of the StreamGlobe system, are implemented in Java. The query registration sequence used when registering a new continuous query is shown in Figure 22. A newly registered WXQuery is parsed and transformed into an equivalent WXQueryX [W3C05b] representation. From this representation, which is a standard XML file, the necessary information to be stored in the query's properties can easily be extracted using XPath [W3C05a].

Optimization starts with the matching of properties data structures as described in this paper and leads to the generation of a query evaluation plan to be installed in the network. The final query evaluation plan is distributed and installed in the network by the StreamGlobe plan distribution component. Eventually, the queries and operators in the plan are executed on the appropriate peers using the FluX query engine.

Figure 23 shows a screenshot of the StreamGlobe monitoring GUI. The GUI is used to continuously monitor and visualize the current network state. The network in the figure is that of Figure 1 with the example data stream of Figure 2 and 25 randomly generated queries registered. The screenshot shows the network state without data stream sharing, leading to overloaded network connections indicated in

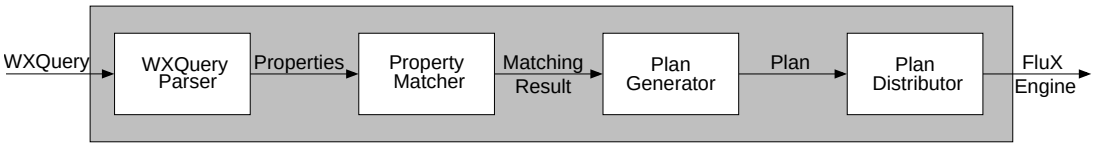


Figure 22: Query registration sequence

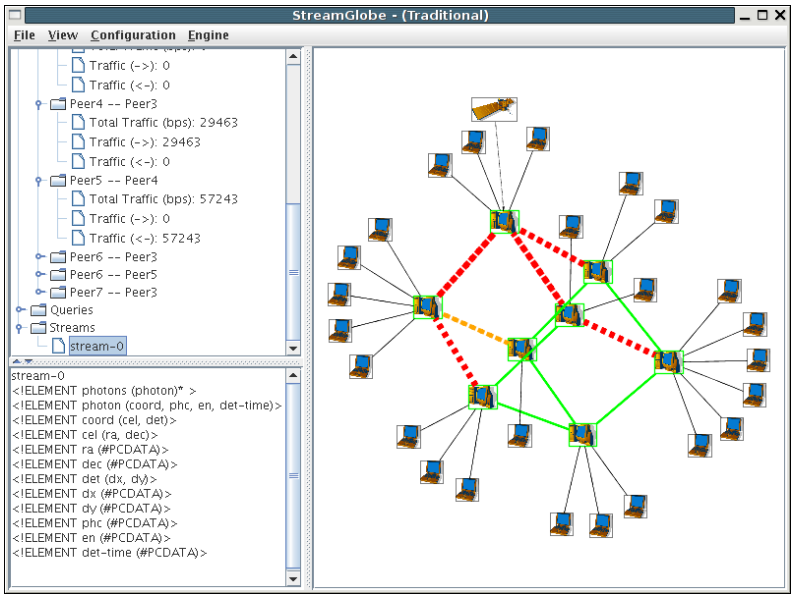


Figure 23: StreamGlobe GUI

orange and red colors in the network graph. Using data stream sharing in this scenario causes all network connections to stay green, i. e., no overload situations occur.

6 Evaluation

This section presents the results of some performance evaluations that we conducted using our prototype implementation in the StreamGlobe system. For the evaluation, the system was installed on a blade server. Each super-peer ran on one blade. The blades had a 2.8 GHz Xeon Processor and 1 GB of main memory each. They were interconnected by a 100 MBit/s LAN. We report on four scenarios here. The first one is the example scenario of Section 1 with 8 super-peers, 1 data stream, and 4 queries. The second scenario is based on the same network topology like the first but registers 25 queries in the system. The third scenario is a small scenario using 4 super-peers, 1 data stream, and 4 queries. Three of the super-peers form a triangle in this scenario and the fourth, which is the one where the data stream is registered at, is connected to one of the three super-peers in the triangle. The fourth scenario uses a 4×4 grid topology with 16 super-peers, 2 data streams, and 100 queries. All data streams and queries are based on real astrophysical data. The queries were generated using query templates for selection, projection, and aggregation queries. Constant values, e. g., in selection predicates or data window definitions, were chosen uniformely from a predefined set of values to enable a certain degree of shareability. The four benchmark scenarios are summarized in Table 1.

For each scenario, we compare three strategies. *Data shipping* simply transmits the whole input data stream for each query from the data source to the target super-peer using a shortest path in the network. The whole query evaluation takes place at the target super-peer. *Query shipping* evaluates each query

Scenario	Network Topology	# Peers	# Data Streams	# Queries
Example	3-dimensional hypercube	8	1	4
Extended Example	3-dimensional hypercube	8	1	25
Small	irregular	4	1	4
Grid	4×4 grid	16	2	100

Table 1: Benchmark scenarios

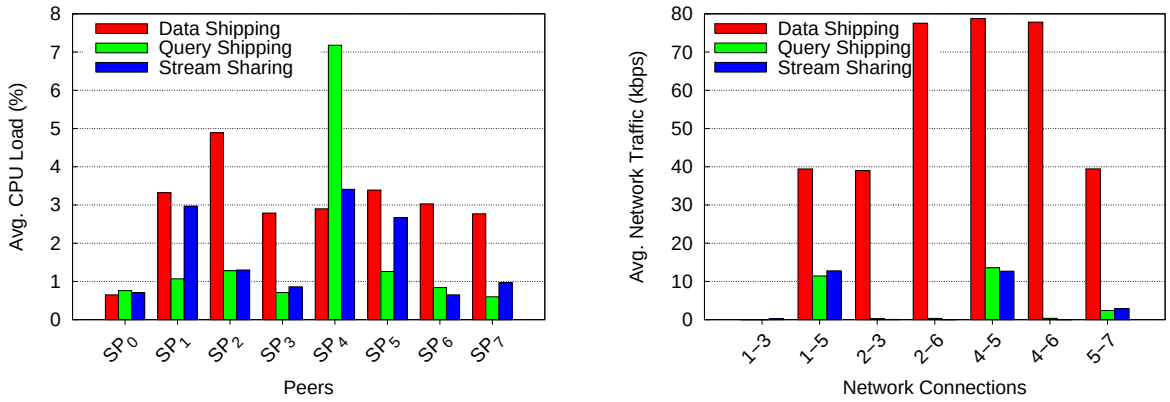


Figure 24: Example scenario: 8 super-peers, 1 data stream, 4 queries

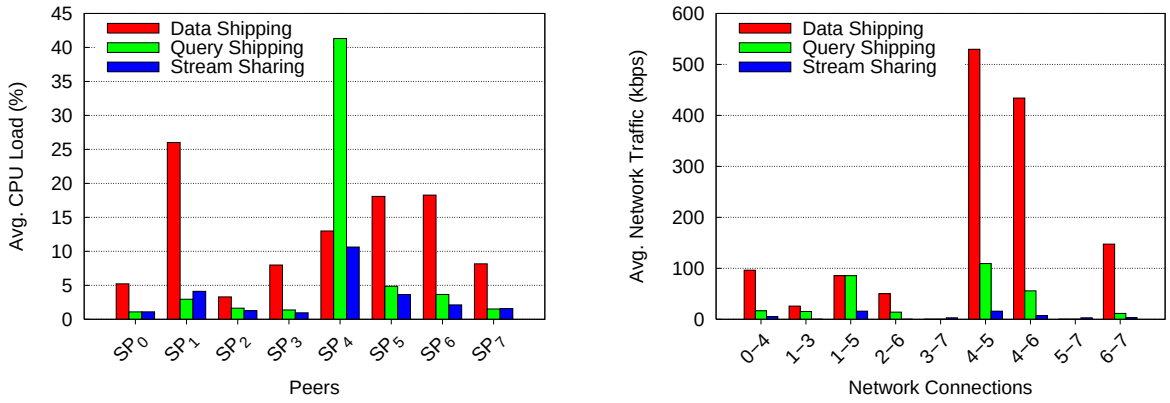


Figure 25: Extended example scenario: 8 super-peers, 1 data stream, 25 queries

completely at the super-peer that the data source is registered at. The query result is transmitted to the target peer again using a shortest path in the network. This of course only works for queries that reference a single input data stream, which is the case in our example queries used here. Finally, *stream sharing* uses our previously described optimization algorithms.

Benchmark results in terms of average CPU load in percent and average network traffic on network connections in kbps are shown in Figures 24 and 25 for the example and the extended example scenarios. As can be clearly seen from the diagrams, query shipping leads to massive peaks of CPU load at data stream source peers since all computation on the respective stream is executed there. On the other hand, network traffic caused by this strategy is comparatively low. Data shipping, as expected, causes much more network traffic but also relatively high CPU load over the whole range of super-peers in the network, since all the data needs to be forwarded over many peers and network connections, often even multiple times. Stream sharing distributes computational load much better over the peers in the network than query shipping and causes less overall CPU load than data shipping. Furthermore, network traffic is also greatly reduced compared to the other two strategies due to the effects of reusing streams for multiple queries.

The results for the remaining two scenarios in terms of average CPU load in percent and accumulated network traffic in MBit including both, incoming and outgoing traffic for each super-peer are shown in Figures 26 and 27, respectively. The results, especially for the larger grid scenario, clearly show, that our approach significantly reduces network traffic at single peers as well as overall in the network. Note that, while data shipping transmits the whole original data stream through the network multiple times, once for each subscription referencing the stream as input, query shipping already significantly reduces network traffic by means of early filtering at the data stream source. However, like data shipping, query shipping still transmits one distinct data stream for each query through the network. Stream sharing

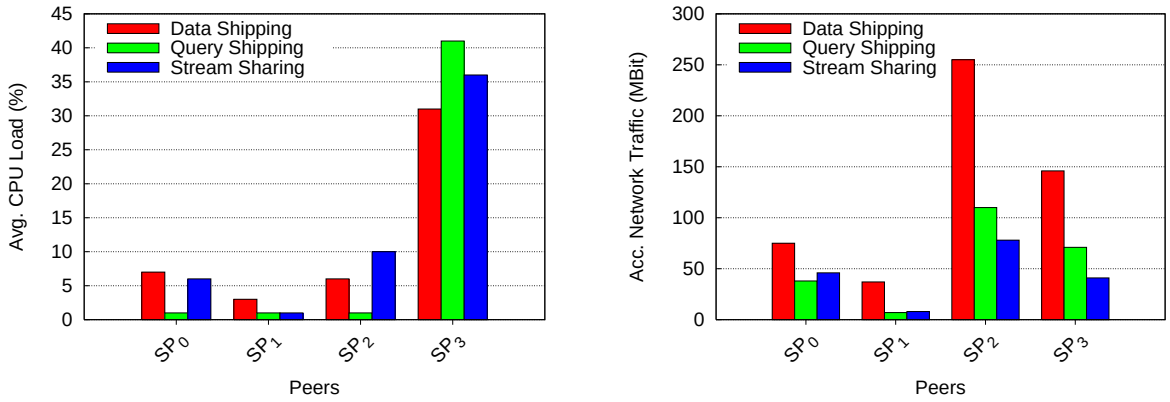


Figure 26: Small scenario: 4 super-peers, 1 data stream, 4 queries

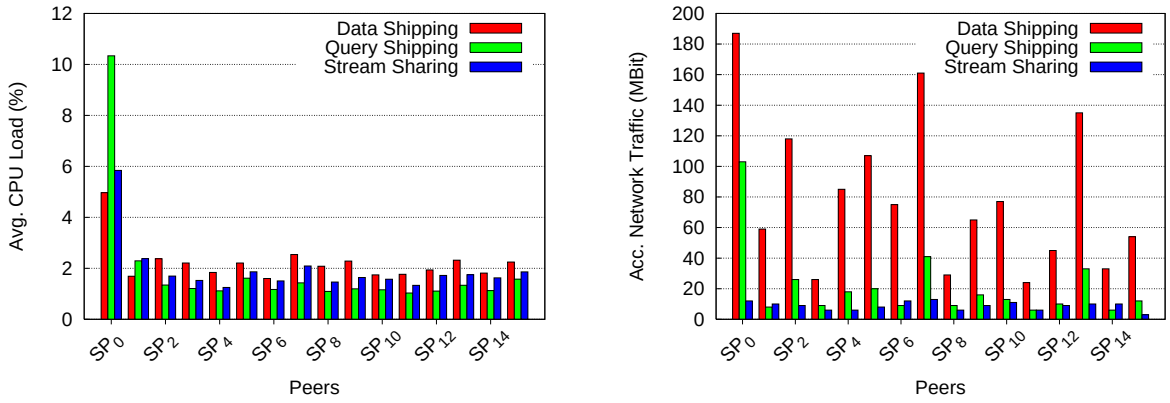


Figure 27: 4 × 4 grid scenario: 16 super-peers, 2 data streams, 100 queries

is able to further reduce network traffic greatly by using multi-subscription optimization, transmitting data streams through the network only once and sharing them for satisfying multiple similar or equal queries. CPU load is comparable to the other approaches on most peers in the shown scenarios, except for the peak at the data stream source nodes for query shipping. We expect our approach to distribute load better over peers in larger scenarios than the other two approaches. This expectation is confirmed by the results of an additional test where we limited the maximum CPU load of peers to 10 % of their actual capacity and the maximum bandwith of network connections between peers to 1 MBit/s. We then used the second scenario and determined how many queries had to be rejected by the system because no query evaluation plan without causing overload on peers or network connections could be found. While query shipping had to reject 35 and data shipping had to reject 47 out of the 100 queries that we tried to register, our stream sharing approach only rejected 2 queries.

Of course, stream sharing does not come for free. Tables 2 and 3 show the times a query took from the beginning of its registration until it was successfully installed and executed in the network in our extended example and our large benchmark scenario, respectively. The stream sharing approach stays within a factor of 3 of the other two much simpler approaches. This is acceptable, since we are dealing with continuous queries that usually remain registered over long periods of time. The difference between the query registration times of data stream sharing and the other two approaches is expected to grow for increasing network sizes and numbers of queries. This is due to the larger effort invested in query optimization. However, many real application scenarios, e. g., e-science collaboration networks, are not supposed to grow far beyond the dimensions of our largest benchmark scenarios. Also, if query registration time should not exceed a certain threshold, the optimization could be stopped early returning the best query evaluation plan found so far.

Time (ms)	Data Shipping	Query Shipping	Stream Sharing
Average	931	890	2153
Minimum	390	284	509
Maximum	2078	2032	5025

Table 2: Query registration times (extended example scenario)

Time (ms)	Data Shipping	Query Shipping	Stream Sharing
Average	1363	1287	3558
Minimum	265	250	672
Maximum	4953	4802	11855

Table 3: Query registration times (4×4 grid scenario)

7 Related Work

The techniques described in this paper can be incorporated into any data stream management system. Numerous DSMSs have been proposed in recent years. Among them are STREAM [ABB+03] which uses the Continuous Query Language (CQL) [ABW03] for registering subscriptions. STREAM processes data streams by transforming them into relations and the query results back into streams again. TelegraphCQ [CCD+03] adaptively processes data streams using, among other things, the Eddy [AH00] approach for adaptive tuple routing. NiagaraCQ [CDTW00] optimizes query processing by sharing common computations among continuous queries through appropriately grouping queries according to similar structures. Aurora is basically a centralized data flow system that processes tuple streams. A decentralized version of Aurora is Aurora*. Finally, Medusa is a distributed infrastructure that supports federated operation of nodes [CBB+03]. Further development lead to the Borealis [AAB+05] system. PIPES [KS04] is a public infrastructure for processing and exploring data streams. The needs of high fan-in systems are being addressed in the HiFi [FJK+05] system. The Cougar [YG02] approach deals with in-network query processing in sensor networks. All of these systems more or less focus on certain aspects of data stream handling and processing. The contributions presented in this paper can be used to augment existing DSMSs to support efficient integration of incrementally subscribed continuous queries. In [DRF04] ONYX, an architecture and techniques for content-based dissemination of XML data in large-scale distributed publish&subscribe systems, is introduced. Like our approach, this system is based on an overlay network. Another example for an astrophysical application in the computer science domain can be found in [LF04].

Our project employs the FluX [KSSS04] query engine for processing continuous queries over XML data streams. Other examples of streaming XQuery implementations include Raindrop [SJR03] and XQRL [FHK+03]. These can be used to process standard XQueries. For being able to execute window-based aggregation operators, they have to be augmented with support for our WXQuery extensions introduced in Section 3.

The approach of optimizing query execution by computing identical or similar parts of queries only once and reusing them multiple times for various queries is similar to multi-query optimization [Sel88]. However, instead of optimizing a set of queries all at once, we incrementally optimize queries one after another when they are registered in the network, based on the current network state. Sharing of work between queries over streams has also been addressed in previous work [MSHR02, KFHJ04]. Our solution differs from these approaches in that we can adaptively distribute subscription evaluation among peers in a network.

The question of which previously generated data stream should be reused for answering a newly subscribed continuous query is similar to the problems of view materialization and view selection in the context of persistent data [TS97, TS99]. In view materialization, however, data is materialized before queries are posed, whereas in our scenario, reusable data streams are generated by previously registered queries in the network.

Even closer related is semantic caching [DFJ+96, CR02], where reusable data also consists of previously

computed query results. Semantic caching differs from query subscription in streaming environments mainly by the difference between processing persistent data and processing data streams. In DSMSs, the cached data corresponds to the data streams travelling through the network.

Another related field is data integration. Instead of matching new subscriptions with existing data streams as is the case in our domain, data integration in peer data management systems (PDMS) [TH04] uses schema matchings in order to match a new query with various data sources that have different, however similar schemas.

Schema matching is one possible approach for comparing newly registered subscriptions with existing data streams in the network. Many different solutions for this problem have been proposed [RB01, DLD+04], also in the context of XML data [DDL00, DDH01]. However, generic schema matchers only match static schema information. This is sufficient for structural filters like projection, but not for content-based filters like selection operators. Supporting this would demand an appropriate extension of the matcher. Furthermore, generic ontology-based schema matchers do not work without user interaction. Since we do not need the matching power of such ontology-based matchers in our context but we do need to match the results of structural and content-based filters alike, we have taken a different approach based on subscription and data stream properties instead.

Of further interest is the problem of query containment, which has also been discussed in the context of XML queries with nesting [DHT04]. Query containment, especially for XML queries, is a difficult problem. We were able to make it manageable by exploiting the properties of our distributed system architecture.

Multicast [DC90] routes data towards receiving ends in a way that reduces network traffic by transmitting the same message or document only once for multiple recipients. It is important to point out that our work differs from this technique in a major way. Instead of merely reusing existing messages or documents needed in identical versions at various network sites, our approach is able to perform expressive in-network transformations. Therefore, it can dynamically create data streams that best fit the queries to be answered while at the same time reducing network traffic and peer load.

8 Conclusion and Future Work

In this paper, we have presented a subscription language, a properties approach, a cost model, and algorithms for registering continuous queries over data streams in P2P networks using data stream sharing. Our approach takes three steps. First, the properties of a newly registered subscription are constructed. Second, shareable data streams generated for answering previously registered subscriptions in the network are identified by matching properties. An appropriate stream for answering the new subscription is chosen according to a cost model that focuses on the reduction of network traffic and peer load, and on load balancing aspects. Finally, operators are placed in the network to execute the new subscription.

We are currently working on an enhanced version of the approach presented in this paper that is able to handle nested queries and to widen data streams. This enables the system to consider data streams for sharing that initially do not contain all the necessary data for a new query but can be altered to do so by changing some operators in the network. Apart from that, there are numerous opportunities for future work. Dynamic optimization on the set of registered subscriptions can be introduced to retain an optimized data flow in the network even if network conditions or data stream statistics change over time. We intend to address the issue of scalability by introducing a hierarchical network organization with several interconnected subnets where each subnet is optimized separately. Alternatively, a fully distributed network architecture could also be realized, completely eliminating the need for specialized speaker-peers. Further, additional stream processing operators like joins will be considered.

References

[AAB+05] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the Conf. on Innovative Data Systems Research*, pages 277–289, Asilomar, CA, USA, January 2005.

[ABB+03] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, March 2003.

[ABW03] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Proc. of the Intl. Workshop on Database Programming Languages*, pages 1–19, Potsdam, Germany, September 2003.

[AH00] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, Dallas, TX, USA, May 2000.

[Asc98] B. Aschenbach. Discovery of a young nearby supernova remnant. *Nature*, 396(6707):141–142, November 1998.

[AW04] A. Arasu and J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 336–347, Toronto, Canada, August 2004.

[BLHS05] C. Bornhövd, T. Lin, S. Haller, and J. Schaper. Integrating Smart Items with Business Processes – An Experience Report. In *Proc. of the Hawaii Intl. Conf. on System Sciences*, page 227.3, Waikoloa, HI, USA, January 2005.

[CBB+03] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable Distributed Stream Processing. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, Dallas, TX, USA, May 2000.

[CR02] L. Chen and E. A. Rundensteiner. ACE-XQ: A CachE-aware XQuery Answering System. In *Proc. of the Intl. Workshop on the Web and Databases*, pages 31–36, Madison, WI, USA, June 2002.

[DC90] S. E. Deering and D. R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Trans. on Computer Systems*, 8(2):85–110, May 1990.

[DDH01] A. Doan, P. Domingos, and A. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 509–520, Santa Barbara, CA, USA, May 2001.

[DDL00] A. Doan, P. Domingos, and A. Levy. Learning Source Descriptions for Data Integration. In *Proc. of the Intl. Workshop on the Web and Databases*, pages 81–86, Dallas, TX, USA, May 2000.

[DFJ+96] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 330–341, Mumbai (Bombay), India, September 1996.

[DHT04] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of Nested XML Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 132–143, Toronto, Canada, August 2004.

[DLD+04] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Semantic Matches between Database Schemas. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 383–394, Paris, France, June 2004.

[DRF04] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 612–623, Toronto, Canada, August 2004.

[FHK+03] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 997–1008, Berlin, Germany, September 2003.

[FJK+05] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-in Systems: The HiFi Approach. In *Proc. of the Conf. on Innovative Data Systems Research*, pages 290–304, Asilomar, CA, USA, January 2005.

[FK04] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 2nd edition, 2004.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, June 2002. `http://www.globus.org/alliance/publications/papers/ogsa.pdf`.

[Glo05] The Globus Alliance, November 2005. `http://www.globus.org`.

[KFHJ04] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The Case for Precision Sharing. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 972–986, Toronto, Canada, August 2004.

[KS04] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 925–926, Paris, France, June 2004.

[KSKR05] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 1259–1262, Trondheim, Norway, August 2005.

[KSSS04] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization on Structured Data Streams. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 228–239, Toronto, Canada, August 2004.

[LF04] D. T. Liu and M. J. Franklin. GridDB: A Data-Centric Overlay for Scientific Grids. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 600–611, Toronto, Canada, August 2004.

[MPE05] Max Planck Institute for Extraterrestrial Physics, November 2005. `http://www.mpe.mpg.de`.

[MSHR02] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, Madison, WI, USA, June 2002.

[RB01] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001.

[RH80] D. J. Rosenkrantz and H. B. Hunt. Processing Conjunctive Predicates and Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 64–72, Montreal, Canada, October 1980.

[RHKS02] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proc. of IEEE INFOCOM*, pages 1190–1199, New York, NY, USA, June 2002.

[Sel88] T. K. Sellis. Multiple-Query Optimization. *ACM Trans. on Database Systems*, 13(1):23–52, March 1988.

[SJR03] H. Su, J. Jian, and E. A. Rundensteiner. Raindrop: A Uniform and Layered Algebraic Framework for XQueries on XML Streams. In *Proc. of the ACM Intl. Conf. on Information and Knowledge Management*, pages 279–286, New Orleans, LA, USA, November 2003.

[SKK04] B. Stegmaier, R. Kuntschke, and A. Kemper. StreamGlobe: Adaptive Query Processing and Optimization in Streaming P2P Environments. In *Proc. of the Intl. Workshop on Data Management for Sensor Networks*, pages 88–97, Toronto, Canada, August 2004.

[TH04] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 539–550, Paris, France, June 2004.

[TS97] D. Theodoratos and T. Sellis. Data Warehouse Configuration. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 126–135, Athens, Greece, August 1997.

[TS99] D. Theodoratos and T. Sellis. Dynamic Data Warehouse Design. In *Proc. of the Intl. Conf. on Data Warehousing and Knowledge Discovery*, pages 1–10, Florence, Italy, August 1999.

[VAB+99] W. Voges, B. Aschenbach, Th. Boller, H. Bräuninger, U. Briel, W. Burkert, K. Dennerl, J. Englhauser, R. Gruber, F. Haberl, G. Hartner, G. Hasinger, M. Kürster, E. Pfeffermann, W. Pietsch, P. Predehl, C. Rosso, J. H. M. M. Schmitt, J. Trümper, and H. U. Zimmermann. The ROSAT All-Sky Survey Bright Source Catalogue. *Astronomy and Astrophysics*, 349(2):389–405, July 1999.

[W3C05a] W3C. XML Path Language (XPath) 2.0 (W3C Candidate Recommendation, November 3rd, 2005), November 2005. `http://www.w3.org/TR/xpath20/`.

[W3C05b] W3C. XML Syntax for XQuery 1.0 (XQueryX) (W3C Candidate Recommendation, November 3rd, 2005), November 2005. `http://www.w3.org/TR/xqueryx/`.

[W3C05c] W3C. XQuery 1.0: An XML Query Language (W3C Candidate Recommendation, November 3rd, 2005), November 2005. `http://www.w3.org/TR/xquery/`.

[YG02] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3):9–18, September 2002.

[YGM03] B. Yang and H. Garcia-Molina. Designing a Super-Peer Network. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 49–60, Bangalore, India, March 2003.

A Alternative XQuery Window Implementations

This section presents alternative XQuery implementations of item-based and time-based data windows.

A.1 Item-based Data Windows

Alternative XQuery implementations of item-based data windows are shown in Figures 28 and 29. In the implementation of Figure 28, the remaining elements are gathered in a final window at the end of the stream. In the implementation of Figure 29, the window slides along until no more elements are contained in it.

```
declare function local:cwin($count as xs:integer,
                            $step as xs:integer,
                            $data as node()*) as node()*
{
  let $cwin := fn:subsequence($data, 1, $count)
  let $tail := fn:subsequence($data, $step + 1)
  return
    if (fn:count($data) <= $count) then
      (<cw> { $cwin } </cw>)
    else
      (<cw> { $cwin } </cw>, local:cwin($count, $step, $tail))
};

for $x in doc("data.xml")/a
return
  <result>
    { for $w in local:cwin(4, 2, $x/b)
      return
        <win> { $w/* } </win> }
  </result>
```

Figure 28: Gathering remaining elements in final window

```
declare function local:cwin($count as xs:integer,
                            $step as xs:integer,
                            $data as node()*) as node()*
{
  let $cwin := fn:subsequence($data, 1, $count)
  let $tail := fn:subsequence($data, $step + 1)
  return
    if (fn:empty($tail)) then
      (<cw> { $cwin } </cw>)
    else
      (<cw> { $cwin } </cw>, local:cwin($count, $step, $tail))
};

for $x in doc("data.xml")/a
return
  <result>
    { for $w in local:cwin(4, 2, $x/b)
      return
        <win> { $w/* } </win> }
  </result>
```

Figure 29: Sliding windows until no elements remain

A.2 Time-based Data Windows

Alternative XQuery implementations of time-based data windows are shown in Figures 30 and 31. In the implementation of Figure 30, the remaining elements are gathered in a final window at the end of the stream. In the implementation of Figure 31, the window slides along until no more elements are contained in it.

```
declare function local:dwin($start as xs:integer,
                            $diff as xs:integer,
                            $step as xs:integer,
                            $data as node()*,
                            $refs as node()*) as node()*
{
  let $dwin := for $i in $data
               let $ds := for $d in $i/descendant-or-self::node()
                          where some $r in $refs satisfies $r is $d
                          return $d
               where $ds >= $start and $ds < $start + $diff
               return $i
  let $tail := for $i in $data
               let $ds := for $d in $i/descendant-or-self::node()
                          where some $r in $refs satisfies $r is $d
                          return $d
               where $ds >= $start + $step
               return $i
  return
    if (fn:count($dwin) = fn:count($data)) then
      (<dw> { $dwin } </dw>)
    else
      (<dw> { $dwin } </dw>, local:dwin($start + $step, $diff, $step, $tail, $refs))
};

for $x in doc("data.xml")/a
return
  <result>
    { for $w in local:dwin(0, 4, 2, $x/b, $x/b/c)
      return
        <win> { $w/* } </win> }
  </result>
```

Figure 30: Gathering remaining elements in final window

```
declare function local:dwin($start as xs:integer,
                            $diff as xs:integer,
                            $step as xs:integer,
                            $data as node()*,
                            $refs as node()*) as node()*
{
  let $dwin := for $i in $data
               let $ds := for $d in $i/descendant-or-self::node()
                          where some $r in $refs satisfies $r is $d
                          return $d
               where $ds >= $start and $ds < $start + $diff
               return $i
  let $tail := for $i in $data
               let $ds := for $d in $i/descendant-or-self::node()
                          where some $r in $refs satisfies $r is $d
                          return $d
               where $ds >= $start + $step
               return $i
  return
    if (fn:empty($tail)) then
      (<dw> { $dwin } </dw>)
    else
      (<dw> { $dwin } </dw>, local:dwin($start + $step, $diff, $step, $tail, $refs))
};

for $x in doc("data.xml")/a
return
  <result>
    { for $w in local:dwin(0, 4, 2, $x/b, $x/b/c)
      return
        <win> { $w/* } </win> }
  </result>
```

Figure 31: Sliding windows until no elements remain

B WXQuery EBNF Grammar

The notation of the following WXQuery EBNF grammar is based on the notation of the XQuery EBNF grammar in [W3C05c]. In particular, it uses a special notation to reference externally defined parts of the grammar via URLs.

```
[1]   QueryBody              ::=  Expr

[2]   Expr                   ::=  ExprSingle ( "," ExprSingle )*

[3]   ExprSingle             ::=  FLWRExpr
                                | PathExpr
                                | ElementConstructor
                                | ParenthesizedExpr
                                | IfExpr

[4]   FLWRExpr               ::=  ( ForClause | LetClause )+
                                  WhereClause?
                                  "return" ExprSingle

[5]   ForClause              ::=  "for" VarRef "in" WindowedPathExpr

[6]   VarRef                 ::=  "$" VarName

[7]   VarName                ::=  QName

[8]   QName                  ::=  [http://www.w3.org/TR/REC-xml-names/#NT-QName]Names

[9]   WindowedPathExpr       ::=  PathExpr ( "|" WindowSpec "|" )?

[10]  PathExpr               ::=  ( ( PrimaryExpr "/" )? RelativePathExpr )
                                | ( "/" ( RelativePathExpr )? )

[11]  PrimaryExpr            ::=  VarRef | XMLFunctionCall

[12]  XMLFunctionCall        ::=  ( "stream"
                                | "doc"
                                | "collection" )
                                  "(" ( StringLiteral )? ")"

[13]  StringLiteral          ::=  ( '"' ( PredefinedEntityRef
                                | CharRef
                                | EscapeQuot
                                | [^"&] )* '"' )
                                | ( "'" ( PredefinedEntityRef
                                | CharRef
                                | EscapeApos
                                | [^'&] )* "'" )

[14]  PredefinedEntityRef    ::=  "&" ( "<" | ">" | "amp" | "quot" | "apos" ) ";"

[15]  CharRef                ::=  [http://www.w3.org/TR/REC-xml#NT-CharRef]XML

[16]  EscapeQuot             ::=  '""'

[17]  EscapeApos             ::=  "''"

[18]  RelativePathExpr       ::=  StepExpr ( "/" StepExpr )*
```

```
[19]  StepExpr                ::=  ContextItemExpr | ( QName ( Predicate )? )

[20]  ContextItemExpr         ::=  "."

[21]  Predicate               ::=  "[" WherePredicate "]"

[22]  WherePredicate          ::=  PredicateOrExpr

[23]  PredicateOrExpr         ::=  PredicateAndExpr ( "or" PredicateAndExpr )*

[24]  PredicateAndExpr        ::=  ( PredicateComparisonExpr | ParenPredOrExpr )
                                   ( "and" ( PredicateComparisonExpr | ParenPredOrExpr ) )*

[25]  PredicateComparisonExpr ::=  PredicateValue ( ComparisonOperator PredicateValue )?

[26]  PredicateValue          ::=  ( ( "-" | "+" )? Literal )
                                   | PredicatePath
                                   | PredicateVariablePath

[27]  Literal                 ::=  StringLiteral | NumericLiteral

[28]  NumericLiteral          ::=  IntegerLiteral
                                   | DecimalLiteral
                                   | DoubleLiteral

[29]  IntegerLiteral          ::=  Digits

[30]  Digits                  ::=  [0-9]+

[31]  DecimalLiteral          ::=  ( "." Digits ) | ( Digits "." [0-9]* )

[32]  DoubleLiteral           ::=  ( ( "." Digits )
                                   | ( Digits ( "." [0-9]* )? ) ) [eE] [+-]? Digits

[33]  PredicatePath           ::=  QName ( "/" QName )*

[34]  PredicateVariablePath   ::=  VarRef ( "/" PredicatePath )*

[35]  ComparisonOperator      ::=  "=" | "!=" | "<" | "<=" | ">" | ">="

[36]  ParenPredOrExpr         ::=  "(" PredicateOrExpr ")"

[37]  WindowSpec              ::=  ( ( "count" IntegerLiteral )
                                   | ( RelPathExprNoPred "diff" IntegerLiteral ) )
                                   ( "step" IntegerLiteral )?

[38]  RelPathExprNoPred       ::=  StepExprNoPredicates ( "/" StepExprNoPredicates )*

[39]  StepExprNoPredicates    ::=  ContextItemExpr | QName

[40]  LetClause               ::=  "let" VarRef ":=" AggFunctionCall

[41]  AggFunctionCall         ::=  AggFunctionName "(" PathExpr ")"

[42]  AggFunctionName         ::=  "min" | "max" | "sum" | "count" | "avg"

[43]  WhereClause             ::=  "where" WherePredicate

[44]  ElementConstructor      ::=  "<" QName
```

```
                                    ( "/>" | ( ">" ( ElementContent )* "</" QName ">" ) )

[45]  ElementContent          ::=   ElementConstructor | EnclosedExpr

[46]  EnclosedExpr            ::=   "{" Expr "}"

[47]  ParenthesizedExpr       ::=   "(" ( Expr )? ")"

[48]  IfExpr                  ::=   "if" "(" PredicateOrExpr ")"
                                    "then" ExprSingle "else" ExprSingle
```

C Example Query Evaluation Plan

```
<?xml version="1.0" encoding="UTF-8"?>
<plan atPeer="http://127.0.0.1:8080/ogsa/services/streamglobe/p2p/Peer/hash
    -23968266-1140103654840"
      id="query-2:plan-1"
      xmlns="urn:streamglobe.in.tum.de/pdc"
      xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <add>
    <streamoperator id="query-2:plan-1:query-2:q1#"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:type="queryStreamoperatorType">
      <dependencies>
        <streamreference id="query-2:plan-1"/>
      </dependencies>
      <source>
        <![CDATA[
          for $p in /photons/photon
          return
            <rxj>
              {$p/coord/cel/ra} {$p/coord/cel/dec}
              {$p/en} {$p/det_time}
            </rxj>
        ]]>
      </source>
      <input-dtd>
        <![CDATA[
          <!ELEMENT photons (photon)* >
          <!ELEMENT photon (coord, en, det_time) >
          <!ELEMENT coord (cel) >
          <!ELEMENT cel (ra, dec) >
          <!ELEMENT ra (#PCDATA) >
          <!ELEMENT dec (#PCDATA) >
          <!ELEMENT en (#PCDATA) >
          <!ELEMENT det_time (#PCDATA) >
        ]]>
      </input-dtd>
      <output-dtd>
        <![CDATA[
          <!ELEMENT photons (photon)* >
          <!ELEMENT photon (coord, en, det_time) >
          <!ELEMENT coord (cel) >
          <!ELEMENT cel (ra, dec) >
          <!ELEMENT ra (#PCDATA) >
          <!ELEMENT dec (#PCDATA) >
          <!ELEMENT en (#PCDATA) >
          <!ELEMENT det_time (#PCDATA) >
        ]]>
      </output-dtd>
    </streamoperator>
    <streamoperator id="query-2:plan-1"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:type="counterStreamoperatorType">
      <dependencies>
        <streamreference id="query-2:plan-0"/>
      </dependencies>
      <ds-item>photon</ds-item>
      <neighbor>
```

```
      http://127.0.0.1:8080/ogsa/services/streamglobe/p2p/Peer/hash
          -12582949-1140103654931
    </neighbor>
  </streamoperator>
</add>
<plan atPeer="http://127.0.0.1:8080/ogsa/services/streamglobe/p2p/Peer/hash
   -12582949-1140103654931"
      id="query-2:plan-0"
      xmlns="urn:streamglobe.in.tum.de/pdc"
      xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <add>
    <streamoperator id="query-2:plan-0"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:type="counterStreamoperatorType">
      <dependencies>
        <streamreference id="query-2"/>
      </dependencies>
      <ds-item>photon</ds-item>
      <neighbor>
        http://127.0.0.1:8080/ogsa/services/streamglobe/p2p/Peer/hash
            -6292125-1140103654995
      </neighbor>
    </streamoperator>
  </add>
  <plan atPeer="http://127.0.0.1:8080/ogsa/services/streamglobe/p2p/Peer/hash
     -6292125-1140103654995"
        id="query-2"
        xmlns="urn:streamglobe.in.tum.de/pdc"
        xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <add>
      <streamoperator id="query-2:q0"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      xsi:type="queryStreamoperatorType">
        <dependencies>
          <stream id="photons"/>
        </dependencies>
        <source>
          <![CDATA[
            <photons>
              {for $p in /photons/photon
               where $p/en >= 1.3
                 and $p/coord/cel/ra >= 130.5
                 and $p/coord/cel/ra <= 135.5
                 and $p/coord/cel/dec >= -48.0
                 and $p/coord/cel/dec <= -45.0
              return
                <photon>
                  <coord>
                    <cel>
                      {$p/coord/cel/ra}
                      {$p/coord/cel/dec}
                    </cel>
                  </coord>
                    {$p/en}
                    {$p/det-time}
                </photon>
              }
```

```
            </photons>
          ]]>
        </source>
        <input-dtd>
          <![CDATA[
            <!ELEMENT photons (photon)* >
            <!ELEMENT photon (coord, phc, en, det_time)>
            <!ELEMENT coord (cel, det)>
            <!ELEMENT cel (ra, dec)>
            <!ELEMENT ra (#PCDATA)>
            <!ELEMENT dec (#PCDATA)>
            <!ELEMENT det (dx, dy)>
            <!ELEMENT dx (#PCDATA)>
            <!ELEMENT dy (#PCDATA)>
            <!ELEMENT phc (#PCDATA)>
            <!ELEMENT en (#PCDATA)>
            <!ELEMENT det_time (#PCDATA)>
          ]]>
        </input-dtd>
        <output-dtd>
          <![CDATA[
            <!ELEMENT photons (photon)* >
            <!ELEMENT photon (coord, en, det_time) >
            <!ELEMENT coord (cel) >
            <!ELEMENT cel (ra, dec) >
            <!ELEMENT ra (#PCDATA) >
            <!ELEMENT dec (#PCDATA) >
            <!ELEMENT en (#PCDATA) >
            <!ELEMENT det_time (#PCDATA) >
          ]]>
        </output-dtd>
      </streamoperator>
    </add>
  </plan>
 </plan>
</plan>
```

D Alternative Aggregate Value Selection Algorithm

Algorithm 6 is an alternative to the aggregate value selection algorithm introduced in Section 4.5. This alternative is computationally more intensive but would save memory for large data items by only buffering items that are actually needed later on. However, our data items are only aggregate values and are therefore comparatively small. Memory usage for small data items tends to be higher in the alternative algorithm than in the algorithm presented in Section 4.5 due to the additional use of sequence numbers and index structures like bit vectors for identifying correct buffer entries. The buffer for the data items is larger in the algorithm of Section 4.5 in general but the algorithm can do without additional sequence numbers and index structures.

Algorithm 6 SelectAggregateValues

Input: Window sizes Δ and Δ' as well as step sizes μ and μ' of a data window to be reused and a new data window, respectively.

Output: The correct sequence of aggregate values for reuse.

1: read first $((\Delta' - \Delta) \operatorname{div} \mu) + 1$ values v_0 to $v_{(\Delta' - \Delta) \operatorname{div} \mu}$ from input stream and assign sequence numbers from 0 to $(\Delta' - \Delta) \operatorname{div} \mu$ to them;
2: **for** value v_n with sequence number n contained in buffer or read from input stream **do**
3: **if** $n \in \{i \cdot (\mu' \operatorname{div} \mu) \mid i > 0 \wedge i \leq \Delta' \operatorname{div} \Delta\}$ **then**
4: insert v_n into buffer;
5: **end if**
6: **if** $n \in \{i \cdot (\Delta \operatorname{div} \mu) \mid i \geq 0 \wedge i < \Delta' \operatorname{div} \Delta\}$ **then**
7: send v_n to output;
8: **end if**
9: **end for**
10: remove values v_0 to $v_{(\mu' \operatorname{div} \mu) - 1}$ from buffer if present;
11: decrease sequence number of each value contained in buffer by $\Delta' \operatorname{div} \Delta$;
12: read next $\Delta' \operatorname{div} \Delta$ values from stream and assign increasing sequence numbers starting from $((\Delta' - \Delta) \operatorname{div} \mu) - (\Delta' \operatorname{div} \Delta) + 1$;
13: continue in line 2 above until buffer contains no more values;

E Example Scenario

```
<?xml version="1.0" encoding="UTF-8"?>
<scenario name="example" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.in.tum.de/projects/StreamGlobe/scenario">

  <statistix dbPath="${user.home}/statistiX" reportType="file" />

  <graph>
    <vertex vid="0" label="010" />
    <vertex vid="1" label="100" />
    <vertex vid="2" label="110" />
    <vertex vid="3" label="001" />
    <vertex vid="4" label="011" />
    <vertex vid="5" label="111" />
    <vertex vid="6" label="101" />
    <vertex vid="7" label="000" />
    <edge source="4" target="0" />
    <edge source="5" target="2" />
    <edge source="6" target="3" />
    <edge source="1" target="7" />
    <edge source="0" target="7" />
    <edge source="5" target="4" />
    <edge source="2" target="0" />
    <edge source="2" target="1" />
    <edge source="4" target="3" />
    <edge source="5" target="6" />
    <edge source="6" target="1" />
    <edge source="3" target="7" />
  </graph>

  <streams>
    <kindDefinition>
      <kind name="file" class="streamglobe.client.p2p.FileContentServer" />
    </kindDefinition>
    <stream sid="photons" type="file">
      <dtd filename="etc/schemas/vela_nested.dtd" />
      <param name="stream.filename">
        /home/strglobe/data/vela/vela_demo.xml
      </param>
      <param name="stream.server.port">
        4711
      </param>
      <param name="stream.sleep.time">
        100
      </param>
    </stream>
  </streams>

  <queries>
    <query qid="1">
      <![CDATA[
        <photons>
        {
          for $p in stream("photons")/photons/photon
          where $p/coord/cel/ra >= 120.0
            and $p/coord/cel/ra <= 138.0
            and $p/coord/cel/dec >= -49.0
            and $p/coord/cel/dec <= -40.0
```

```
      return
        <vela>
          {$p/coord/cel/ra} {$p/coord/cel/dec}
          {$p/phc} {$p/en} {$p/det-time}
        </vela>
    }
    </photons>
  ]]>
</query>

<query qid="2">
  <![CDATA[
    <photons>
    {
      for $p in stream("photons")/photons/photon
      where $p/en >= 1.3
        and $p/coord/cel/ra >= 130.5
        and $p/coord/cel/ra <= 135.5
        and $p/coord/cel/dec >= -48.0
        and $p/coord/cel/dec <= -45.0
      return
        <rxj_photon>
          {$p/coord/cel/ra} {$p/coord/cel/dec}
          {$p/en} {$p/det-time}
        </rxj_photon>
    }
    </photons>
  ]]>
</query>

<query qid="3">
  <![CDATA[
    <photons>
    {
      for $w in stream("photons")/photons/photon
        [coord/cel/ra >= 120.0
         and coord/cel/ra <= 138.8
         and coord/cel/dec >= -49.0
         and coord/cel/dec <= -40.0]
        |det_time diff 20 step 10|
      let $a := avg($w/photon/en)
      return
        <avg_en>
          {$a}
        </avg_en>
    }
    </photons>
  ]]>
</query>

<query qid="4">
  <![CDATA[
    <photons>
    {
      for $w in stream("photons")/photons/photon
        [coord/cel/ra >= 120.0
         and coord/cel/ra <= 138.0
         and coord/cel/dec >= -49.0
         and coord/cel/dec <= -40.0]
```

```
            |det_time diff 60 step 40|
          let $a := avg($w/photon/en)
          return
            <avg_en>
              {$a}
            </avg_en>
        }
        </photons>
      ]]>
    </query>

  </queries>

  <injectorMapping>
    <mapping peer="4" stream="photons" />
  </injectorMapping>

  <queryMapping>
    <mapping peer="0" query="1" />
    <mapping peer="2" query="2" />
    <mapping peer="1" query="3" />
    <mapping peer="6" query="4" />
  </queryMapping>

</scenario>
```