Data Stream Sharing[⋆]

Richard Kuntschke and Alfons Kemper

Institut für Informatik, Technische Universität München, Munich, Germany
{richard.kuntschke|alfons.kemper}@in.tum.de

Abstract. Recent research efforts in the fields of data stream processing and data stream management systems (DSMSs) show the increasing importance of processing data streams, e. g., in the e-science domain. Together with the advent of peer-to-peer (P2P) networks and grid computing, this leads to the necessity of developing new techniques for distributing and processing continuous queries over data streams in such networks. In this paper, 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 as well as experimental evaluation results from the astrophysics application domain to assess our approach.

1 Introduction

Over the past few years, data stream processing and data stream management systems (DSMSs) have been very active research areas. This trend is promoted by the increasing need to process streaming data on-the-fly whenever possible, instead of storing intermediate results or buffering whole input data sets before processing. Newly upcoming and evolving fields, such as e-science applications in physics and astronomy, deal with huge volumes of data and render storing all of the delivered data 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.

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 of existing (parts of) data streams that were generated to satisfy previously registered subscriptions.[1]

[⋆] This research is supported by the German Federal Ministry of Education and Research within the D-Grid initiative under contract 01AK804F and by Microsoft Research Cambridge under contract 2005-041.

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

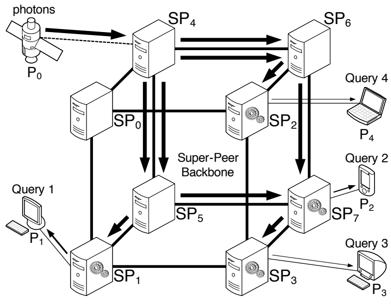


Fig. 1. No stream sharing

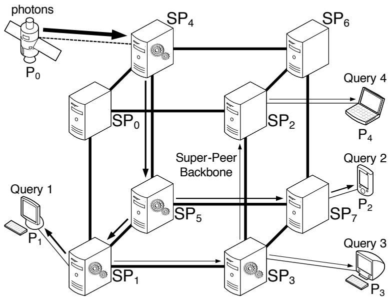
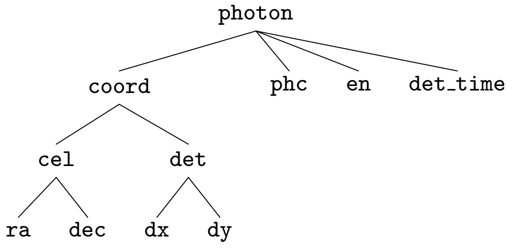


Fig. 2. Stream sharing

These optimizations are an integral part of our *StreamGlobe* system [1, 2]. To enable them, we use *peer-to-peer (P2P) networking* techniques. In contrast to the conventional use of P2P networks for *file sharing*, StreamGlobe uses P2P-based networks for *data stream sharing*. The system architecture is based on a P2P overlay backbone network that is organized as a *super-peer network* [3], 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* standards (OGSA) and therefore fits seamlessly into existing e-science platforms.

As a motivating example, we introduce an astrophysical e-science application. Consider Figures 1 and 2 which both illustrate the same exemplary network. Here, SP_0 to SP_7 are the super-peers that constitute the super-peer backbone network and P_0 to P_4 are thin-peers. Peer 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) which we obtained through our cooperation partners from the Max Planck Institute for Extraterrestrial Physics (MPE).

In our scenario, we deal with streams of XML data. The data items in stream `photons` comply to a DTD with the tree structure shown below. As its name implies, the data stream delivers a stream of photons detected by the telescope's photon detector. Each photon contains its celestial and detector pixel coordinates, its detector pulse, its energy, and its detection time.



We assume that 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 2. We will only consider Queries 1 and 2 of Figures 1 and 2 here. Queries 3 and 4 will be presented in Section 2. All queries reference data stream `photons` as their single input. Query 1 (Q1) is shown below.

```
Q1:  <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>
```

This query selects the area of the *vela supernova remnant.* The `stream` function was newly introduced by us and indicates a possibly infinite data stream used as input to the query. Query 2 (Q2) below filters a smaller section of the sky.

```
Q2:  <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>
```

This query selects the area of the *RX J0852.0-4622 supernova remnant* 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 which shows the traditional scenario of data shipping. The thickness of the arrows associated with the various network connections indicates the size of the data streams transmitted over these 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 data sink 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.

Figure 2 shows the benefits of using our stream sharing approach which answers newly registered subscriptions using (parts of) data streams already present in the network. This includes data streams which have been generated earlier 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 specified by 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.

The contributions presented in this paper are as follows. 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. Second, we present a properties representation of data streams and subscriptions, a cost model, and algorithms for optimizing the evaluation of newly registered continuous queries in a data stream management system by sharing possibly preprocessed data streams. Eventually, we show experimental evaluation results to assess our approach.

The paper is organized as follows. In Section 2, we introduce WXQuery. Our new data stream sharing approach is presented in Section 3. Section 4 shows some evaluation results. Related work is presented in Section 5. Finally, Section 6 concludes and states ongoing and future work.

2 Subscription Language

In StreamGlobe, subscriptions over XML data streams are registered using *Windowed XQuery (WXQuery)*. WXQuery is a fragment of XQuery [4] that has been augmented with support for window-based operators.

In Definition 2.1 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 $\varPhi$, i. e., $\varPhi \in \{\texttt{min},\texttt{max},\texttt{sum},\texttt{count},\texttt{avg}\}$.

Expressions enclosed in $\llbracket\,\rrbracket^{?}$, $\llbracket\,\rrbracket^{*}$, or $\llbracket\,\rrbracket^{+}$ 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 2.1 (WXQuery). *The WXQuery subscription language comprises all subscriptions that consist only of the following expressions:*

1. $\texttt{<}t\texttt{/>}$ (empty direct element constructor)
2. $\texttt{<}t\texttt{>}\ \llbracket\alpha_{1,2} \mid \texttt{\{}\alpha_{3,4,5,6,7}\texttt{\}}\rrbracket^{*}\ \texttt{</}t\texttt{>}$ (direct element constructor)
3. $\llbracket\texttt{for }\$x\texttt{ in }\$y\llbracket\texttt{/}\pi\rrbracket^{?}\llbracket\texttt{|count }\varDelta\ \llbracket\texttt{step }\mu\rrbracket^{?}\texttt{|} \mid \texttt{|}\overline{\pi}\texttt{ diff }\varDelta\ \llbracket\texttt{step }\mu\rrbracket^{?}\texttt{|}\rrbracket^{?} \mid$ $\texttt{let }\$a\texttt{ := }\varPhi\texttt{(}\$y\llbracket\texttt{/}\pi\rrbracket^{?}\texttt{)}\rrbracket^{+}\ \llbracket\texttt{where }\chi\rrbracket^{?}\ \texttt{return }\alpha$ (FLWR expression)

4. `if` χ `then` α `else` β (conditional expression)
5. $\$y/\pi$ (output of subtrees reachable from node $\$y$ through path π)
6. $\$z$ (output of subtree rooted at node $\$z$)
7. $(\llbracket\alpha\llbracket,\beta\rrbracket^{*}\rrbracket^{?})$ (sequence)

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 Figures 1 and 2 is an example for the use of such an operator.

```
Q3:  <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>
```

Query 4 (Q4) employs a different window.

```
Q4:  <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>
```

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 a 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. The value of the reference element of a time-based data window can either be a real or an abstract timestamp. 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 XPath semantics. Rather, it specifies the reference element controlling the window.

Conditions in our context, whether they appear in a `where` clause ("χ") or within a path ("`[`p`]`"), are conjunctions of *atomic predicates*. Atomic predicates are 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.

Restructuring (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. Since attributes in XML data can always be converted into corresponding elements, we restrict ourselves to dealing with elements.

3 Data Stream Sharing

This section introduces our properties-based approach for representing subscriptions and data streams, our cost model, and the algorithms for searching, identifying, and choosing appropriate streams for satisfying new subscriptions.

3.1 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. An abstract schematic illustration of the properties of Query 1 from Section 1 is shown in Figure 3. 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 predicates, are stored using a graph representation as shown in Figure 3. Data window specifications 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 ($\varDelta$) and the step size (μ). This approach supports flat WXQueries without nesting. An advanced approach supporting nested queries 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.

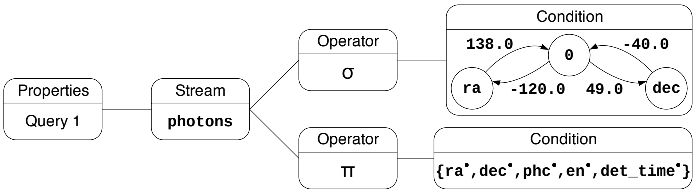


Fig. 3. Abstract properties of Query 1

3.2 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 $\mathit{\Pi}_{p_s}$ 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:

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

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 sum of the sizes of the 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 the 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 size 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 respective 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 a set of 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 represented by properties 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 $C(\mathcal{P}) < C(\mathcal{P}')$.

3.3 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.

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 Subscribe algorithm 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. 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 previously registered continuous queries—that can also be used to answer q, possibly by applying some further transformations.

The SUBSCRIBE algorithm starts with an empty evaluation plan $\mathcal{P}$ (line 1 in Algorithm 1) and iterates over the properties of 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 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 it is registered to the peer where q is registered via the shortest path in the network and performing any query evaluation tasks on data stream 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 $\mathcal{P}_s$. Finally, the start node v_b of the search 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 2.

After the initialization, the algorithm 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 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 below. Network connections that do not have any associated properties because they do not carry any data streams are ignored

Algorithm 1 SUBSCRIBE

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: $\mathcal{P} \leftarrow \emptyset$;
 2: **for all** $p_s \in getInputDS(p_q)$ **do**
 3:     $L_V \leftarrow \emptyset$; $L_P \leftarrow \emptyset$;
 4:     $p_b \leftarrow p_s$; $v_b \leftarrow getTNode(p_b)$;
 5:     $\mathcal{P}_s \leftarrow generatePlan(p_b, v_b, v_q)$;
 6:     $add(L_V, v_b)$;
 7:     **while** $L_V \neq \emptyset$ **do**
 8:         $v \leftarrow dequeue(L_V)$; $mark(v)$;
 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** $L_P \neq \emptyset$ **do**
13:             $p \leftarrow dequeue(L_P)$;
14:             **if** MATCHPROPERTIES$(p, p_s)$ **then**
15:                 $n \leftarrow getTNode(p)$;
16:                 **if** $(\neg(isMarked(n)) \wedge (n \notin L_V))$ **then**
17:                     $add(L_V, n)$;
18:                 **end if**
19:                 $\mathcal{P}'_s \leftarrow generatePlan(p, v, v_q)$;
20:                 **if** $\mathcal{P}'_s \prec_C \mathcal{P}_s$ **then**
21:                     $p_b \leftarrow p$; $v_b \leftarrow v$; $\mathcal{P}_s \leftarrow \mathcal{P}'_s$;
22:                 **end if**
23:             **end if**
24:         **end while**
25:     **end while**
26:     unmark all nodes;
27:     $add(\mathcal{P}, \mathcal{P}_s)$;
28: **end for**
29: **return** $\mathcal{P}$;
```

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.

Matching Properties. Next, we explain how Algorithm 2 matches properties. For each input data stream of a subscription, the properties of the subscription reflect which 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 sets of 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 sharing if p' also contains a corresponding selection. Otherwise, the stream represented by p would not contain all the necessary data for the query represented by p'. 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 projection elements of p 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 Query 1 in Figure 3—has to be a superset of the set R' of all the elements referenced by the query, marked as well as unmarked, 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 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 operator applied to the same inputs must always yield the same result. 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 shareability. More sophisticated techniques for identifying shareable user defined operators involve the development of suitable operator descriptions providing the necessary meta data. Developing such techniques and operator descriptions is part of future work.

Algorithm 2 MatchProperties

Input: The properties p of a data stream considered for sharing and p' of a subscription to be registered.

Output: true if p and p' match, false otherwise.

```
$s \leftarrow getDS(p);\ s' \leftarrow getDS(p');$
if $s \neq s'$ then
    return false;
end if
$O \leftarrow getOps(s);\ O' \leftarrow getOps(s');$
for all $o \in O$ do
    $match \leftarrow$ false;
    for all $o' \in O'$ do
        if $o = o'$ then
            $C \leftarrow getConds(o);\ C' \leftarrow getConds(o');$
            if $o = \sigma$ then
                $G \leftarrow getPGraph(C);\ G' \leftarrow getPGraph(C');$
                if MatchPredicates$(G, G')$ then
                    $match \leftarrow$ true; break;
                end if
            else if $o = \Pi$ then
                $\overline{R} \leftarrow getOutElems(C);\ R' \leftarrow getRefElems(C');$
                if $\overline{R} \supseteq R'$ then
                    $match \leftarrow$ true; break;
                end if
            else if $o \in \{\texttt{min},\texttt{max},\texttt{sum},\texttt{count},\texttt{avg}\}$ then
                if MatchAggregations$(C, C')$; then
                    $match \leftarrow$ true; break;
                end if
            else
                $\vec{i} \leftarrow getParams(C);\ \vec{i'} \leftarrow getParams(C');$
                if $\vec{i} = \vec{i'}$ then
                    $match \leftarrow$ true; break;
                end if
            end if
        end if
    end for
    if $match =$ false then
        return false;
    end if
end for
return true;
```

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 [5]. 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

Algorithm 3 MatchPredicates

Input: The predicate graphs $G = (V, E)$ of a data stream considered for sharing and $G' = (V', E')$ of a subscription to be registered.

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

```
for all $v \in V$ do
  $vmatch \leftarrow$ false;
  for all $v' \in V'$ do
    if $v \hat{=} v'$ then
      $vmatch \leftarrow$ true;
      for all $x \in \{e \in E | e$ connected to $v\}$ do
        $ematch \leftarrow$ false;
        for all $y \in \{e' \in E' | e'$ connected to $v'\}$ do
          if $\zeta(x) \Leftarrow \zeta(y)$ then
            $ematch \leftarrow$ true; break;
          end if
        end for
        if $ematch =$ false then
          return false;
        end if
      end for
      break;
    end if
  end for
  if $vmatch =$ false then
    return false;
  end if
end for
return true;
```

$\$v \leq \$w + c$ 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 an illustrating example, consider Figure 3 which contains the predicate graph of the selection in Query 1. After the construction of G, the predicate can be checked for satisfiability and is minimized using techniques introduced in earlier related work [5]. 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 the registration process.

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

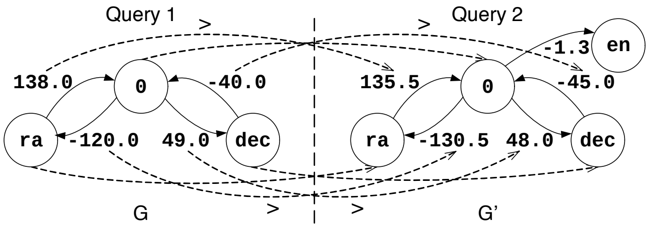


Fig. 4. Matching predicates

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 each node v in the node set V of G, there exists an equivalent node v' in the node set V' of G', denoted by $v \hat{=} v'$ 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 by $\zeta(x)$, is compatible with the atomic predicate represented by y, denoted by $\zeta(y)$. 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 4. 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

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

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

Window-based Aggregation. Sharing results of window-based aggregation operators has been studied before [6]. Our approach differs from this previous solution in two ways. First, we introduce a step size 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 sharing 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 operation 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

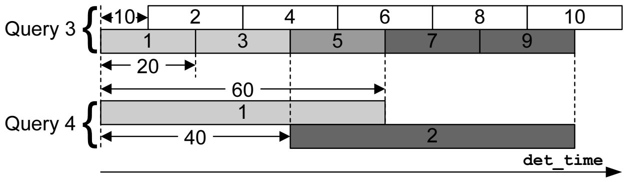


Fig. 5. Reusing window-based aggregates

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 sharing 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 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 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. These three conditions for data window shareability can be expressed as $\Delta' \bmod \Delta = 0$, $\Delta \bmod \mu = 0$, and $\mu' \bmod \mu = 0$. The sharing of result data streams of window-based aggregation operators is illustrated in Figure 5, using Queries 3 and 4 of Section 2 as examples.

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 as introduced above can be relaxed.

4 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 two scenarios here. The first one is based on the network topology of the example scenario of Section 1 and involves 8 super-peers, 1 data stream, and 25 queries. The second 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.

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 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 Figure 6 for the first scenario. 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 second scenario 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 Figure 7. The results clearly indicate, 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

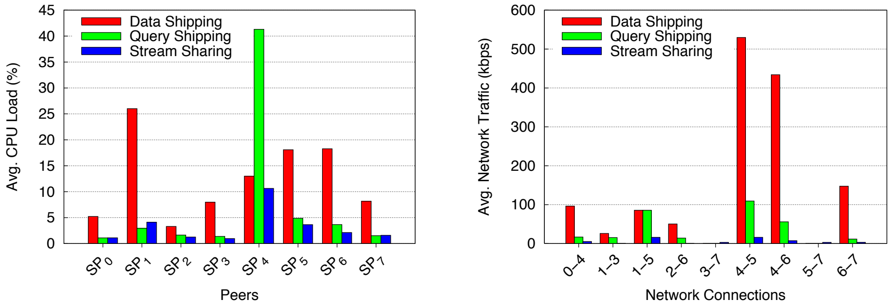


Fig. 6. Extended example scenario: 8 super-peers, 1 data stream, 25 queries

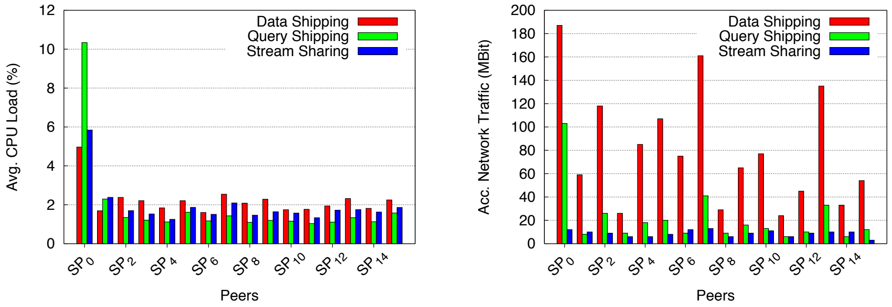


Fig. 7. 4 × 4 grid scenario: 16 super-peers, 2 data streams, 100 queries

network traffic by means of early filtering at the data stream source. However, like data shipping, query shipping still transmits one data stream per query in the network. Stream sharing is able to further reduce network traffic greatly by using multi-subscription optimization, i. e., transmitting data streams in 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 this scenario, except for the peak at the data stream source node 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 data shipping had to reject 47 and query shipping had to reject 35 out of the 100 queries that we tried to register, our stream sharing approach only rejected 2 queries in this scenario.

Of course, stream sharing does not come for free. Table 1 shows the times in milliseconds a query took from the beginning of its registration until it was successfully installed and executed in the network in the two benchmark scenarios. 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.

Table 1. Query registration times

TIME (ms)	AVERAGE		MINIMUM		MAXIMUM	
SCENARIO	1	2	1	2	1	2
Data Shipping	931	1363	390	265	2078	4953
Query Shipping	890	1287	284	250	2032	4802
Stream Sharing	2153	3558	509	672	5025	11855

5 Related Work

Numerous DSMSs have been proposed in recent years [7–12]. The contributions presented in this paper can be used to augment existing DSMSs to support the efficient integration of incrementally subscribed continuous queries.

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 [13]. 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 [14, 15]. Our solution differs from these approaches in that we can adaptively distribute subscription evaluation among peers in a network.

Of further interest is the problem of query containment, which has also been discussed in the context of XML queries with nesting [16]. 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.

Finally, for more details on data stream sharing, we refer to an extended version of this paper [17].

6 Conclusion

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. Finally, operators are placed in the network to execute the new subscription. An experimental evaluation confirms the effectiveness of our approach.

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. One is to address the issue of scalability by introducing a hierarchical network organization with several interconnected subnets where each subnet is optimized separately.

References

1. Stegmaier, B., Kuntschke, R., Kemper, A.: StreamGlobe: Adaptive Query Processing and Optimization in Streaming P2P Environments. In: Proc. of the Intl. Workshop on Data Management for Sensor Networks, Toronto, Canada (2004) 88–97
2. Kuntschke, R., Stegmaier, B., Kemper, A., Reiser, A.: StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures. In: Proc. of the Intl. Conf. on Very Large Data Bases, Trondheim, Norway (2005) 1259–1262
3. Yang, B., Garcia-Molina, H.: Designing a Super-Peer Network. In: Proc. of the IEEE Intl. Conf. on Data Engineering, Bangalore, India (2003) 49–60
4. W3C: XQuery 1.0: An XML Query Language (W3C Candidate Recommendation, November 3rd, 2005) (2005) http://www.w3.org/TR/xquery/.
5. Rosenkrantz, D.J., Hunt, H.B.: Processing Conjunctive Predicates and Queries. In: Proc. of the Intl. Conf. on Very Large Data Bases, Montreal, Canada (1980) 64–72
6. Arasu, A., Widom, J.: Resource Sharing in Continuous Sliding-Window Aggregates. [18] 336–347
7. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: Proc. of the Conf. on Innovative Data Systems Research, Asilomar, CA, USA (2005) 277–289
8. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., Widom, J.: STREAM: The Stanford Stream Data Manager. IEEE Data Engineering Bulletin **26**(1) (2003) 19–26
9. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. [19]
10. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Dallas, TX, USA (2000) 379–390
11. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S.B.: Scalable Distributed Stream Processing. [19]
12. Yao, Y., Gehrke, J.: The Cougar Approach to In-Network Query Processing in Sensor Networks. ACM SIGMOD Record **31**(3) (2002) 9–18
13. Sellis, T.K.: Multiple-Query Optimization. ACM Trans. on Database Systems **13**(1) (1988) 23–52
14. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously Adaptive Continuous Queries over Streams. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Madison, WI, USA (2002) 49–60
15. Krishnamurthy, S., Franklin, M.J., Hellerstein, J.M., Jacobson, G.: The Case for Precision Sharing. [18] 972–986
16. Dong, X., Halevy, A.Y., Tatarinov, I.: Containment of Nested XML Queries. [18] 132–143
17. Kuntschke, R., Stegmaier, B., Kemper, A.: Data Stream Sharing. Technical Report TUM-I0504, Technische Universität München (2005)
18. Proc. of the Intl. Conf. on Very Large Data Bases. In: Proc. of the Intl. Conf. on Very Large Data Bases, Toronto, Canada (2004)
19. Proc. of the Conf. on Innovative Data Systems Research. In: Proc. of the Conf. on Innovative Data Systems Research, Asilomar, CA, USA (2003)