Data Stream Sharing*

Richard Kuntschke and Alfons Kemper

Technische Universität München, Munich, Germany
⟨*firstname.lastname*⟩@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.

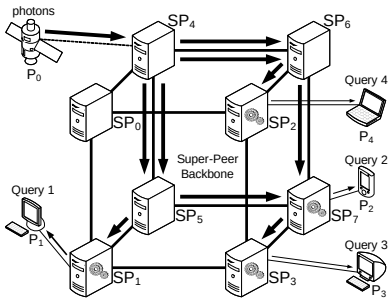


Fig. 1. No Stream Sharing

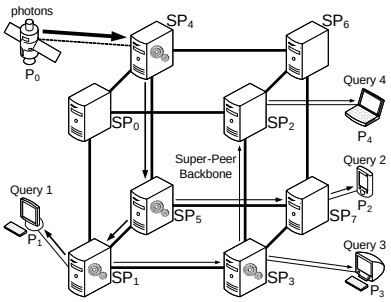
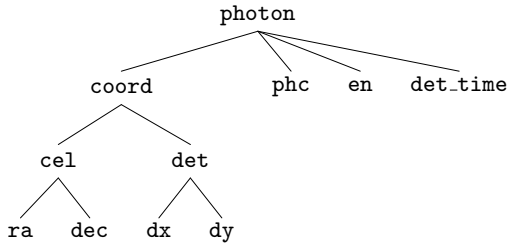


Fig. 2. Stream Sharing

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 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.

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 describes some related work. Section 5 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 that has been augmented with support for window-based operators.

In Definition 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 represent a reference to the root 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 $\overline{\psi}$ 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 ψ differs from $\overline{\psi}$ in that it can also contain conditions. An aggregation operator is denoted by Φ, i. e., $\Phi \in \{$`min,max,sum,count,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 1 (WXQuery). *The WXQuery subscription language comprises all subscriptions that consist only of the following expressions:*

1. `<`t`/>` (empty direct element constructor)
2. `<`t`>` $[\![\alpha_{1,2}]\!]^*$ `</`t`>` | `<`t`>` $[\![$`{`$\alpha_{3,4,5,6,7}$`}`$]\!]^*$ `</`t`>` (direct element constructor)
3. $[\![$`for` $\$x$ `in` $\$y[\![/\psi]\!]^?[\![$`|count` Δ $[\![$`step` $\mu]\!]^?$`|` | `|` $[\![/]\!]^?\overline{\psi}$ `diff` Δ $[\![$`step` $\mu]\!]^?$`|`$]\!]^?$ | `let` $\$a$ `:=` Φ`(`$\$y[\![/\psi]\!]^?$`)`$]\!]^+$ $[\![$`where` $\chi]\!]^?$ `return` α (FLWR expression)
4. `if` χ `then` α `else` β (conditional expression)
5. $\$y$`/`$\psi$ (output of subtrees reachable from node $\$y$ through path ψ)

6. $\$z$ (output of subtree rooted at node $\$z$)
7. `()` (empty 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]
           |/photon/det_time diff 20 step 10|
         let $a := avg($w/photon/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]
           |/photon/det_time diff 60 step 40|
         let $a := avg($w/photon/en)
         where $a >= 1.3
         return <avg_en> { $a } </avg_en> }
     </photons>
```

The definition of a data window is enclosed in "`|`" characters. Element-based windows—indicated by the keyword `count`—contain a fixed number of elements given by the numeric value of Δ. Optionally, a step 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 elements and, during each update, removes the 10 oldest entries from the window while adding the next 10 new elements arriving in the stream. If omitted, the step 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 indicates the time interval between two successive data windows. Again, the step defaults to Δ if omitted. Time-based windows can only be applied on data streams that are sorted according to the values of the 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 `|/photon/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

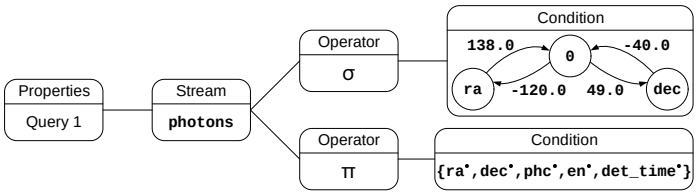


Fig. 3. Abstract Properties of Query 1

are of the form $\$v\ \theta\ c$, $\$v\ \theta\ \$w$, or $\$v\ \theta\ \$w + c$, where $\$v$ and $\$w$ represent paths of the form $\overline{\psi}$, 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 output elements, etc.) is done in a postprocessing 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

In our context, subscriptions and data streams can be represented by the same properties data structure. 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.

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 an original input data stream, a set of operators used to transform the 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 (Δ) and the step (μ). This approach supports flat WXQueries without nesting. An advanced approach supporting nested queries is part of future work.

3.2 Cost Model

We now introduce the cost model used by our algorithm. The cost function C currently 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. 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 I_q be the set of properties of all input data streams of q, $\overline{occ}(n_i)$ the average occurrence and $\overline{size}(n_i)$ the average size of element n_i in input stream i, and π_{p_i} the set of projection elements of p concerning input stream i. Then, $\overline{size}(p)$ is calculated using the following formula:

$$\overline{size}(p) := \sum_{i \in I_q} \left(\overline{size}(i) - \sum_{n_i \notin \pi_{p_i}} (\overline{occ}(n_i) \cdot \overline{size}(n_i)) \right)$$

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 $\overline{size}(n_i)$ denotes the average size of one element of type n_i (e. g., the `phc` element of a photon). Furthermore, for window and aggregation operators, $\overline{size}(p)$ has to consider average window and aggregate result sizes.

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:

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

Note that the expression $\sum_{i \in I_q} \overline{freq}(i)$ 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 operators, $\overline{freq}(p)$ depends on the step defined for the data window and the average frequency of the input 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:

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

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 I_o is denoted $\overline{load}(o, v, I_o)$. 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:

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

Cost function inputs like average frequencies of data stream items, average sizes and occurrences of elements, and selectivities of operators are obtained from statistics and selectivity estimations. The average load $\overline{load}(o, v, I_o)$ of an operator o on a peer v with input stream properties I_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 $\overline{freq}(i)$ of its only input stream i, the average load caused by σ on a peer v can be defined as $\overline{load}(\sigma, v, i) := bload(\sigma) \cdot pindex(v) \cdot \overline{freq}(i)$. 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:

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

In this function, EP 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_{EP} is the set of network connections and V_{EP} is the set of peers affected by plan EP. 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 EP is better than another plan EP' according to cost function C, expressed by $EP \prec_C EP'$, if and only if $C(EP) < C(EP')$.

3.3 Stream Sharing Algorithms

We now describe our stream sharing algorithms for registering and efficiently satisfying new continuous queries.

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 inputs to the algorithm are p_q and v_q, the properties of the new subscription q and the network node where it is registered, respectively. The output of the algorithm is the evaluation plan EP, 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 q refers to existing inputs—namely the plan using q's original input streams. The goal of our approach is to find 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.

The algorithm starts with the initialization of a FIFO queue L_V for network nodes (peers) and another queue L_P for properties. 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. Note that this plan does not reuse any existing preprocessed data streams in the network. The algorithm then basically performs a breadth-first search in the network graph for each input stream, inserting the node that corresponds to the super-peer at which the corresponding original input stream of q is registered into L_V. 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. 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 such peer, all properties of data streams travelling on network connections connected to the currently handled peer are subsequently inserted into L_P. These properties are then consecutively taken out of the queue and matched against the properties p_q of q. This is described 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 (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. Then, the value of cost function C for the plan reusing the found data stream is computed and compared against the current best solution. 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. 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 and all input streams of q have been considered, the algorithm terminates and returns the current best solution for plan EP as the final result.

Matching 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 are the properties of the data stream that is considered for reuse and the properties of the newly reg-

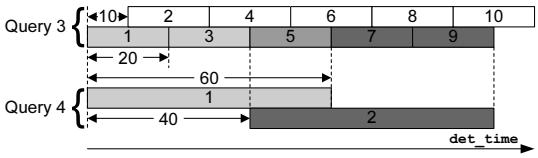


Fig. 4. Reusing Window-based Aggregates

istered subscription. The algorithm returns *true* if these properties match and *false* otherwise. For the properties to match, the data stream considered for reuse and the respective input stream of the new subscription must reference the same original input data stream in the network (e. g., the stream `photons` in the introductory example). Furthermore, for the streams to match, the operators and operator conditions applied to the streams must be compatible. In the case of selections, this means that each selection predicate in the data stream properties must be implied by a corresponding selection predicate in the properties of the new subscription. The next section presents more details on this. In the case of projections, the set of projection elements in the properties of the data stream considered for sharing must be a superset of the set of elements referenced in the properties of the new subscription. Finally, window-based aggregation operators and their data windows must be compatible.

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 [4]. In addition to integer valued variables and constants, we also allow decimal values with a finite number of decimal places.

The algorithm for matching predicates can match any predicates (e. g., selection predicates, join predicates, etc.) in that graph representation. 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.

Window-based Aggregation The sharing of result data streams of window-based aggregation queries is illustrated in Figure 4, using Queries 3 and 4 of Section 2 as examples. We require three conditions to hold for aggregate sharing to be possible. These are that the size and the step of the query window must be multiples of the size and the step of the window of the data stream to be reused, respectively. Further, the size of the stream window must be a multiple of its step. Note that all three conditions hold in our example.

Figure 5 shows some evaluation results for a set of 100 randomly generated queries, confirming the reduction of network traffic achieved by our stream sharing technique compared to traditional data and query shipping approaches. For more details on sharing the result data streams of window-based aggrega-

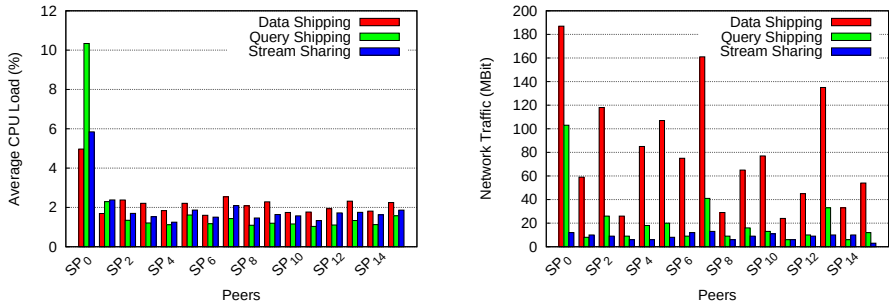


Fig. 5. 4 × 4 Grid Scenario: 16 Super-Peers, 2 Data Streams, 100 Queries

tion queries along with pseudocode representations of our algorithms and further performance evaluation results we refer to an accompanying technical report [5].

4 Related Work

Numerous DSMSs have been proposed in recent years [6–10]. The contributions presented in this paper can be used to augment existing DSMSs to support 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 [11]. 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 [12, 13]. 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 [14]. 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.

5 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.

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. 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
5. Kuntschke, R., Stegmaier, B., Kemper, A.: Data Stream Sharing. Technical Report TUM-I0504, Technische Universität München (2005)
6. 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
7. 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. In: Proc. of the Conf. on Innovative Data Systems Research, Asilomar, CA, USA (2003)
8. 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
9. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S.B.: Scalable Distributed Stream Processing. In: Proc. of the Conf. on Innovative Data Systems Research, Asilomar, CA, USA (2003)
10. Yao, Y., Gehrke, J.: The Cougar Approach to In-Network Query Processing in Sensor Networks. ACM SIGMOD Record **31**(3) (2002) 9–18
11. Sellis, T.K.: Multiple-Query Optimization. ACM Trans. on Database Systems **13**(1) (1988) 23–52
12. 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
13. Krishnamurthy, S., Franklin, M.J., Hellerstein, J.M., Jacobson, G.: The Case for Precision Sharing. In: Proc. of the Intl. Conf. on Very Large Data Bases, Toronto, Canada (2004) 972–986
14. Dong, X., Halevy, A.Y., Tatarinov, I.: Containment of Nested XML Queries. In: Proc. of the Intl. Conf. on Very Large Data Bases, Toronto, Canada (2004) 132–143