Customizable Parallel Execution of Scientific Stream Queries

Milena Ivanova and Tore Risch

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden

Technical report 2005-012
April 2005
ISSN 1404-3203

Customizable Parallel Execution of Scientific Stream Queries

Milena Ivanova   Tore Risch

Abstract

Scientific applications require processing high-volume on-line streams of numerical data from instruments and simulations. We present an extensible stream database system that allows scalable and flexible continuous queries on such streams. Application dependent streams and query functions are defined through an Object-Relational model. Distributed execution plans for continuous queries are described as high-level data flow distribution templates. Using a generic template we define two partitioning strategies for scalable parallel execution of expensive stream queries: window split and window distribute. Window split provides operators for customized parallel execution of query functions whose complexity depends on size of the data units on which they are applied. It reduces the size of stream data units using application dependent functions as parameters. By contrast, window distribute provides operators for customized distribution of entire data units without reducing their size. We evaluated these strategies for a typical high volume scientific stream application and show that window split is favorable when computational resources are limited, while window distribute is better when there are sufficient resources.

1 Introduction

In order to explore information from very high volume raw data generated by scientific instruments, such as satellites, on-ground antennas, and simulators, scientists need to perform a wide range of analyses over the data streams. Complex analyses are presently done off-line on data stored on disk using hard-coded predefined processing of the data. The off-line processing creates large backlogs of unanalyzed data and the high volume produced by scientific instruments can even be too large to store and process [13, 14]. Furthermore, off-line data processing prevents timely analysis after interesting natural events occurred.

We address these problems by utilizing an extensible scientific stream database system, GSDM[1], where scientists can specify in a flexible way analyses as on-line distributed continuous queries (CQs) over the streams. A distributed and

[1]Grid Stream Database Manager

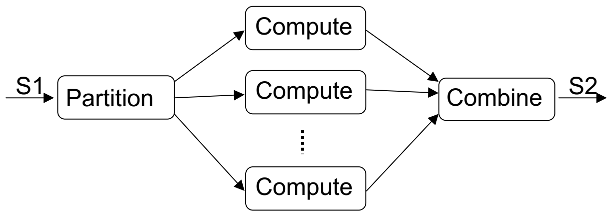


Figure 1: *A generic data flow distribution template for partitioning parallelism*

parallel architecture provides scalability for both data volumes and computations.

GSDM provides a generic framework for specifying distributed strategies to execute continuous queries on streams. The user specifies operators on stream data as declarative *stream query functions*, SQFs, defined over *logical windows* of stream data. The SQFs may contain user-defined functions implemented in, e.g., C and plugged into the system. The system is extensible with new types of stream data sources and SQFs over them.

The user may also specify *data flow distribution templates, DFDTs*, which are parameterized descriptions of how to express a CQ as distributed compositions of SQFs together with a logical site assignment for each SQF. For extensibility, a DFDT may be defined in terms of other data flow distribution templates. For scalable execution of CQs containing expensive SQFs we provide a generic DFDT for customizable data partitioning parallelism. It is illustrated in Figure 1 and contains three phases: *partition, compute,* and *combine*. In the partition phase the stream is split into sub-streams, in the compute phase an SQF is applied in parallel on each sub-stream, and in the combine phase the results of the computations are combined into one stream.

The generic DFDT has been used to define two different stream partitioning strategies: SQF dependent *window split (WS)* and SQF independent *window distribute (WD)*. Window split provides SQF dependent partition and combine strategies while window distribute is applicable on any SQF. Window split is favorable, e.g., for many numerical algorithms on vectors that scale through user-defined vector partitioning. Both strategies use a pair of non-blocking and order preserving SQFs to specify the partition and combine phases.

The partition phase in window split is defined by a DFDT, *operator dependent stream split(OS-Split)* to perform application dependent splitting of logical windows into smaller ones. An SQF, *operator dependent stream join(OS-Join)*, implements the combine phase.

Window split is particularly useful when scaling the logical window size for an SQF with complexity higher than O(n) over the window size. For example, our Space Physics application [14] requires, e.g., the FFT (Fast Fourier Transform) to be applied on large vector windows and we use OS-Split and OS-Join to implement an FFT-specific stream partitioning strategy. FFT is commonly used in signal processing applications and is computationally expensive. Therefore, it strongly affects the performance of entire class of application queries.

As a window distribute strategy, we provide a *Round Robin stream partitioning* (RR) strategy where entire logical windows of streams are distributed based on the order they arrive. In the combine phase, the result sub-streams are merged on their order identifier[2]. This is an extension of the conventional Round Robin partitioning [9] for data streams. Window distribute by Round Robin does not decrease the size of logical windows; therefore the compute phase of FFT could run slower than with window split.

Our experiments show that both partitioning strategies have advantages in specific situations. If the CQ is to be executed on a limited number of nodes, so that the compute phase overloads processors, the window split is preferable since it utilizes semantics of the SQFs to achieve a more scalable parallel execution. However, if the system has resources enough to avoid the overloading, window distribute with RR may have better performance depending on the cost of partitioning and combining SQFs.

Our contributions are:

- High-level data flow distribution templates (DFDTs) specify distributed execution patterns of CQs in terms of SQFs. This allows easy specification of CQs combined with various user-defined stream partitioning strategies. In particular we defined a generic DFDT for partitioned parallel execution of expensive SQFs.

- *Window split* strategies are defined by parameterizing the generic DFDT with a partitioning DFDT, *operator dependent stream split(OS-Split)*, and a combining SQF, *operator dependent stream join(OS-Join)*. They allow windows of different streams to be split and joined through user defined partitioning and combining functions while preserving the stream order.

- The same generic DFDT is also used for defining *window distribute* strategies by parameterizing it with a partitioning DFDT, *stream distribute(S-Distribute)* and a combining SQF, *stream merge(S-Merge)*. They provide SQF independent data distribution that preserves the stream order and are further parameterizable by, e.g., Round Robin partitioning.

- We compared window split and window distribute for an example scientific application to evaluate their scalability. Experimental results show that window split can improve scalability, in particular when SQFs require substantial computational resources.

- A GSDM system architecture has been designed and implemented. A coordinator module compiles DFDTs into execution plans where logical nodes are assigned to execution nodes, sets up the GSDM execution nodes, and supervises the CQ execution.

[2]E.g., in our application a time stamp is used.

The rest of the paper is organized as follows: In the next section we present related work. Section 3 presents GSDM. Different strategies for scalable execution of expensive scientific stream operators are described in Section 4 while Section 5 analyzes their performance. Section 6 summarizes.

2 Related work

Parallel CQs processing in [10, 11] is provided by the *flux* operator that encapsulates general partitioning strategies, such as hash partitioning. We also have customized general partitioning and in addition investigate operator-dependent window split strategies for expensive operators over streams of non-relational data.

The main advantage of the first version of *flux* [11] is adaptive partitioning on the fly for optimal load balancing of parallel CQ processing. One of the motivations is the fact that content-sensitive partitioning schemas as hashing can cause big data skew in the partitions and therefore need load balancing. We do not deal with load imbalance problems since the partitioning schemas we consider (WS and WD with RR), chosen to meet our scientific application requirements, are content insensitive, i.e. do not cause load imbalance in a homogeneous cluster environment. The last version of *flux* [10] encapsulates fault-tolerance logic and is not related to the problems addressed here.

The need for partition, compute, and combine phases for user-defined functions in object-relational databases was indicated by [8]. However, the idea to specify generic and modular data flow distribution patterns through DFDTs is to the best of our knowledge unique.

Data partitioning strategies for parallel databases [9] such as Round Robin can be used as parameters of window distribute strategy. What makes the stream partitioning strategies different is that the processing must preserve chronological ordering of the stream. We provide this property by special stream operators synchronizing the parallel result streams in the combine phase.

The idea to separate parallel functionality from data partitioning semantics by customized partitioning functions is similar to Volcano's [4] support functions parameterizing the *exchange* operator. In contrast, we have pairs of partition and combine operators where the combine operator preserves stream order. While window distribute parameterized by, e.g., Round Robin is similar to the *exchange* operator, window split is novel. Furthermore, we express stream partitioning and combining as high level declarative SQFs, which allows the user to customize the parallel execution using knowledge about the application semantics.

Most of the stream processing systems [1, 3, 6, 7] are based on the relational model, have fine granularity of stream data items, and small cost of stream operators per item. In contrast, the streams in the scientific applications we address have big total volume and data item size, and the operators are computationally expensive. Therefore, we address the problem for parallelizing expensive stream operators to achieve scalable execution.

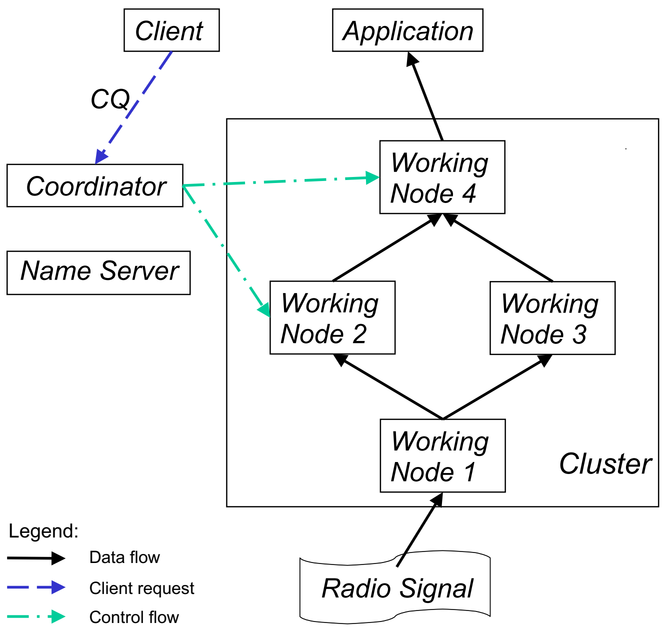


Figure 2: *GSDM System Architecture with an example data flow graph*

Some projects [2, 15] deal with processing of distributed streams with small items and cheap operators. This is different than the problem to efficiently partition expensive stream operators. Box splitting in distributed eddies[15] is a form of parallel processing of a stream operator where data partitioning is done as part of a tuple routing policy. This is similar to our window distribute, but we customize explicitly the data partitioning strategy and also provide order preservation. Window split does not have analogue in any stream database system.

Similarly to Tribeca[12] we utilize an extensible object-based model. Our stream operators for the window distribute strategy are related to Tribeca's demultiplexing (*demux*) and multiplexing (*mux*) operators, but they have more restricted partitioning based on data content for aggregation purposes rather than for parallelization.

The idea to extend the concept of a database query with numerical computations over scientific data was originally proposed by [16], but the work does not address parallel execution nor stream databases.

3 The GSDM System

Figure 2 illustrates the architecture of the GSDM system with an example of a generated data flow graph for execution of a CQ. Through a GSDM *client* the

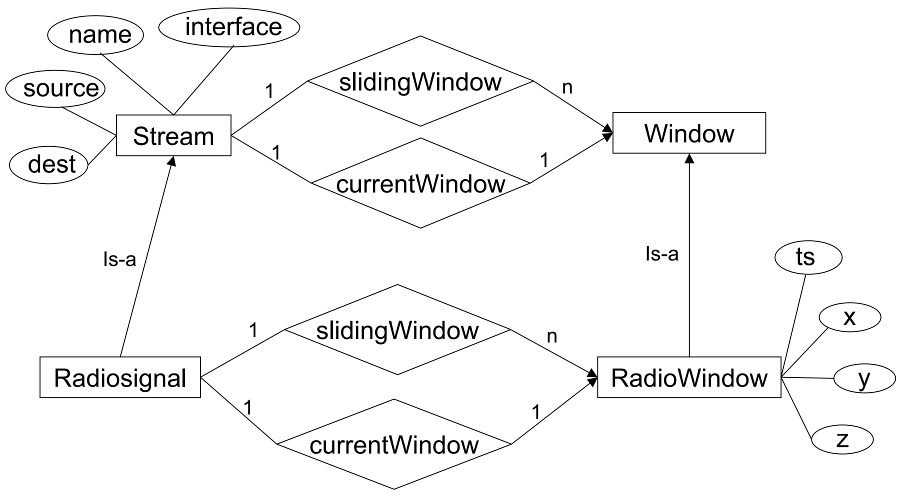


Figure 3: *Metadata of Radio Signal Stream Source*

user submits the CQ specification to the *coordinator*. The specification contains the characteristics of stream data sources such as data types and IP addresses, the destination of the result stream, and the SQFs to be executed in the query. The CQ can be specified to run for a limited amount of time or until explicitly stopped by the user.

The coordinator manages CQs and processing resources. Given a CQ and a DFDT, the coordinator acquires resources from a cluster computer and constructs an execution plan as a distributed data flow graph where GSDM *working nodes* execute SQFs.

3.1 Stream Query Functions

The stream data are modeled through an extensible Object-Relational data model where entities are represented as types organized in a hierarchy. The entity attributes and the relationships between entities are represented as functions on objects. In this model, the stream data sources are instances of a user-defined type *Stream*(Fig. 3) with functions *name* that identifies the stream, and *source* and *dest* that specify stream source and destination addresses, respectively.

In our model stream elements are objects called *logical windows*. A logical window can be an atomic object but is usually a collection, which can be ordered vector (sequence) or unordered bag. The elements of the collections can be any type of object.

The logical windows are represented as instances of subtypes of an abstract type *Window*. Streams with different types of logical windows are represented as subtypes of the type *Stream*.

A *stream query function (SQF)* is a declarative query that computes a logical window in a result stream given one or several input streams. The GSDM engine executes continuously an SQF to produce output windows inserted by the engine into the result stream. To provide referential transparency in SQFs, the GSDM engine moves a *cursor* over each stream. There is a library of *stream access functions* that return logical windows from streams relative to the current cursor position. For example, the generic function

```
currentWindow(Stream s) -> Window w
```

returns the current logical window *w* at the cursor of an input stream *s*.

There are also functions aggregating logical windows from a stream. For example, the function

```
slidingWindow(Stream s, Integer sz, Integer st)
              -> Vector of Window w
```

combines *sz* next logical windows in a stream *s* into a vector of logical windows. The parameter *st* is the sliding step.

The stream access functions are overloaded for each user stream subtype and generated automatically when a new user stream type is registered to the system. They do not have side effects since they operate on a list of pointers to logical windows maintained by the system.

The streams in our application [14] (Fig. 3) are radio signals produced by digital space receivers represented by type *Radiosignal*. The instrument produces three signal channels, one for each space dimension, and a time stamp. Thus, each logical window of type *RadioWindow* has the attributes *ts, x, y*, and *z*, where *ts* is a time stamp and *x, y*, and *z* are vectors of complex numbers representing sequences of signal samples.

The types and functions in the application specific part of Figure 3 are generated when the user defines an application stream type by calling a system procedure, *create_stream_type*[3], e.g.:

```
create_stream_type("RADIOSIGNAL", {"ts","x","y","z"},
   {"timeval","vector of complex",
   "vector of complex", "vector of complex"});
```

The SQF *fft3* below is defined on *Radiosignal* stream type and computes FFT on each of the three channels of the current logical window of the radio stream. It calls a foreign function *fft* that computes the FFT over a vector of complex numbers[4]:

```
create function fft3(Radiosignal s) -> RadioWindow
as select radioWindow({ts(v),fft(x(v)),fft(y(v)),fft(z(v))})
   from RadioWindow v
   where v = currentWindow(s);
```

[3]The notation {...} is used for constructing vectors (sequences) in GSDM.

[4]The function *radioWindow* is a system generated constructor of a new instance of type RadioWindow

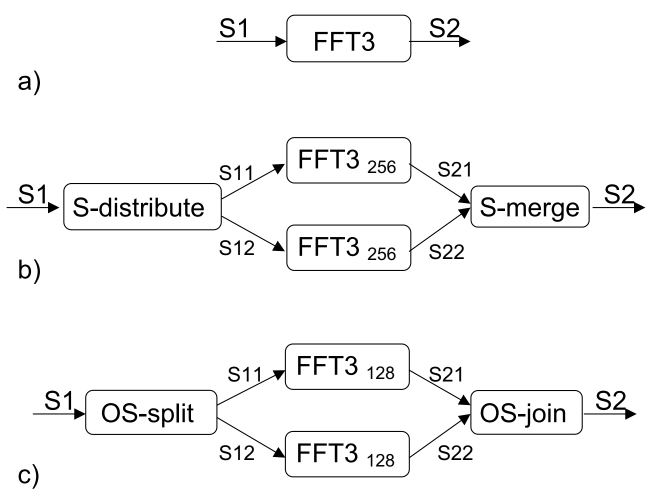


Figure 4: *(a)FFT Central Execution (b)Round Robin Strategy in 2 (c)Operator Dependent Strategy in 2*

To enable stream processing independent on the physical communication media of the streams, a stream type needs interface methods for different physical media (function *interface*). Every stream maintains its own buffer and cursor through the stream interface methods *open*, *next*, *insert*, and *close*. These methods have side effects on the state of the stream and are therefore called only by the GSDM engine when executing an SQF. The *next* method reads the next logical window from an input stream and moves the cursor, while *insert* emits a logical window to an output stream. The system provides support for streams communicated on TCP and UDP protocols, local streams stored in main memory, streams connected to the standard output, or to visualization programs.

3.2 Data Flow Distribution Templates

A data flow distribution template specifies a CQ as distributed composition of SQFs or other DFDTs. Each DFDT has a *constructor* that creates a data flow graph where vertices are SQFs assigned to logical execution sites. The arcs in the graph are producer-consumer relationships between SQFs. We provide a library of DFDTs including the generic *PCC (Partition-Compute-Combine)* that specifies a lattice-shaped data flow graph template as in Figure 1.

For example the *window distribute* data flow in Figure 4b is created by:

```
set wd= PCC(2,"S-Distribute","RRpart","fft3", "S-Merge",0.1);
```

The graph constructor PCC is parameterized on i) the degree of parallelism (*2*); ii) partitioning method (*S-Distribute*); iii) parameter of the partitioning method (*RRPart*); iv) SQF to be computed (*fft3*); v) the combining method (*S-Merge*); and vi) parameter of the combining method (*0.1*, a time-out).

The operator dependent *window split* data flow in Figure 4c is created by:

```
set ws = PCC(2,"OS-Split","fft3part","fft3",
            "OS-Join","fft3combine");
```

In this case the parameters *fft3part* and *fft3combine* are FFT-dependent window transformation functions defined in the next section. They are parameters of the partitioning method OS-Split and the combining method OS-Join.

Executions of SQFs on one node are specified by a DFDT constructor called *Central*. For example, the following call generates the central data flow graph shown in Fig. 4a:

```
set c = Central("fft3");
```

Furthermore, DFDTs can also be used in place of SQF arguments in calls to DFDT constructors in order to construct complex graph structures. For example, the following call creates the distributed graph in Fig. 8a:

```
set wd-tree = PCC(2,"S-Distribute","RRpart",
    "PCC",{2,"S-Distribute","RRpart","fft3",
         "S-Merge",0.1},
    "S-Merge",0.1);
```

3.3 Continuous Query Compilation and Execution

The data flow graph created by calls to DFDT constructors is compiled and executed. The compilation produces a physical distributed execution plan given the input and output streams of the CQ, along with computational resources for the execution, i.e. an IP address of a cluster computer.

For example:

```
set s1 = register_input_stream("Radiosignal","1.2.3.4","UDP");
set s2 = register_result_stream("1.2.3.5","Visualize");
compile(ws, {s1}, {s2}, "hagrid.it.uu.se");
```

In the example the data flow graph *ws* has one input stream of type *Radiosignal* accessible by a stream interface called *UDP*. The result stream connects to a visualizing application on the specific address using a stream interface called *Visualize*. The execution nodes are requested from the cluster named *hagrid.it.uu.se*.

The compiler first requests cluster resources and maps the logical execution sites specified through the DFDT to the execution nodes in the cluster.

The data flow graph is then traversed starting with SQF vertices connected to input streams. For the result of each traversed SQF the compiler creates internal streams connecting to the consuming SQFs.

The result of the compilation is a physical execution plan, also in form of data flow graph, that is installed on the working nodes distributed according to the execution site assignments.

Finally the execution plan is started by calling a run procedure to activate the data flow:

```
run(ws);
```

4 Query Execution Strategies

In this section we investigate different strategies for parallelizing an application dependent stream operator using as an example the SQF *fft3* defined in the previous section.

First, we look at what strategies are appropriate to parallelize application specific operators over scientific data streams and formulate the requirements for the strategies. In the next section, we evaluate their scalability measured in terms of total maximum throughput and size of the logical windows wrt. the SQFs.

In the work presented we consider data partitioning parallelism for an expensive SQF[5]. Future work will include a CQ optimizer that constructs and searches in a wider space of distributed execution patterns.

We can formulate the following requirements for stream data partitioning strategies to parallelize expensive SQFs:

1) Partitioning must preserve semantics of the SQF.

2) The partitioning strategy must be order preserving.

3) The partitioning strategy has to provide as good as possible load balancing.

The scalability of a parallel data flow graph depends both on the scalability of the SQF and of the partitioning strategy itself.

Our two overall stream data partitioning strategies, *window distribute* and *window split* fulfill the requirements stated above. Window distribute distributes entire logical windows to different partitions. Since the SQFs are executed on logical windows, window distribute does not affect the parallelized operator neither is it dependent on it. The routing of windows can be based on any well-known partitioning strategy, such as Round Robin, hash partitioning, or other user-defined partitioning. This is a parameter of the *S-distribute* DFDT.

In contrast, the window split strategy splits a single logical window into sub-windows that are distributed to corresponding partitions. In this way a stream operator can be executed in parallel on the sub-windows, which allows to achieve better scalability of expensive SQFs with respect to the sizes of the logical windows. In order to preserve the operator semantics window split needs knowledge about application data types and SQF semantics when creating and combining sub-windows. Therefore, the stream operators implementing

[5]Notice that complex queries can always be encapsulated in SQFs.

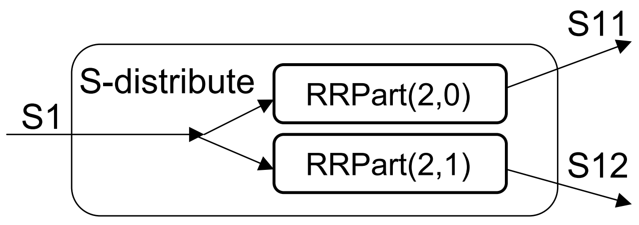


Figure 5: *S-Distribute DFDT with RRPart as parameter and 2 partitions*

the window split strategy need SQF-dependent parameters specifying window splitting and combining functions.

Both partitioning strategies utilize the generic PCC defined above.

4.1 Window Distribute Implementation in GSDM

For window distribute we provide a partitioning DFDT, *S-Distribute*, and a combining SQF, *S-Merge*, with the following signatures:

```
S-Distribute(Integer n, Function distrf) -> Dataflow
S-Merge(Vector of Stream s, Real timeout) -> Window
```

S-Distribute takes as parameter *n* the number of partitions and an SQF *distrf* that selects the next logical window for a sub-stream. *S-Distribute* generates a distributing hypernode in the data flow graph. Figure 5 illustrates the result of the following call to the constructor of *S-Distribute* with parameters 2 and *RRpart*, specifying Round Robin partitioning on two nodes:

```
S-Distribute(2,"RRpart");
```

The *RRpart* is an SQF that specifies a single Round Robin partition on any stream of logical windows:

```
create function RRpart(Stream s, Integer ptot, Integer pno)
                -> Window
as select w[pno]
   from Vector of Window w
   where w = slidingWindow(s,ptot,ptot);
```

Here, *ptot* is the total number of partitions and *pno* is the order number of the partition selected.

In order to fulfill the order preserving requirement above, the combine phase must order result sub-streams after the compute phase. This is the purpose of the *S-Merge* stream operator. It assumes that the sub-streams are ordered by,

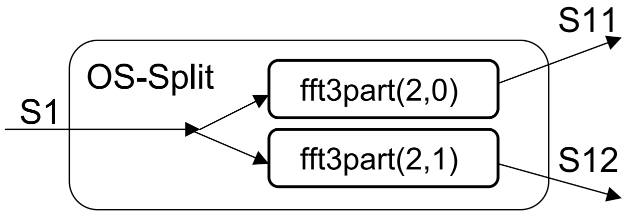


Figure 6: *OS-Split DFDT with fft3part as parameter and 2 partitions*

e.g., a time stamp, and thus it is a variant of merge join on the stream ordering attribute extended with an additional parameter - a time-outout period. The time-out is needed since the partitioned sub-streams are processed on different execution nodes, which introduces communication and/or processing delays at the merging node. Since the merge algorithm needs to be non-blocking for real time stream processing, it has a policy how to handle delayed or lost data. Our policy is to introduce the time-out. It is the time period that the *S-Merge* waits for a stream window to arrive if it is not present locally before assuming that the window was lost. Other policies, such as replacement or approximation of missing windows are also possible.

The following call to *S-Merge* merges the result sub-streams on time stamp with time-out parameter set to 0.1 sec.:

```
S-Merge({s21,s22},0.1));
```

The notations s21 and s22 are logical names of streams from the compute phase (Figure 4b).

Our choice to implement Round Robin as parameter of window distribute was based on the fact that it provides good load balancing. Other strategies, such as hash partitioning, are content-sensitive, i.e. the decision where to distribute a window is based on the content in the window. They usually introduce load imbalance due to the data skew. For some applications this disadvantage can be compensated by the benefits for queries such as join or grouping on the partitioning key. Such benefits cannot be expected in our signal processing application.

4.2 Window Split Implementation in GSDM

The window split strategy can be used for a particular stream operator if a pair of window transformation functions are defined that specify how to split a logical window into sub-windows and how to combine the result sub-windows while preserving the SQF semantics.

Data flow graphs for *Window split* partitioning use the DFDT *OS-Split* and the SQF *OS-Join* as parameters of the generic PCC. They have the following signatures:

```
OS-Split(Integer n, Function splitf) -> Dataflow;
OS-Join(Vector of Stream s, Function combinef) -> Window;
```

OS-Split takes as a parameter the number of partitions n, which is equal to the number of sub-windows to be created from one logical window. Another parameter is a window transformation function *splitf* that specifies how sub-windows are created from the original logical window. The following call to the constructor of OS-Split creates the hypernode in Figure 6:

```
OS-Split(2,"fft3part");
```

fft3part is an FFT specific window transformation function that splits a window into sub-windows by splitting its vector components:

```
create function fft3part(Radiowindow w,
   Integer ptot, Integer pno) -> RadioWindow
as select radioWindow({ts(w),
      fftpart(x(w),ptot,pno),
      fftpart(y(w),ptot,pno),
      fftpart(z(w),ptot,pno)})
```

Here *ptot* is the total number of partitions and *pno* is the order number of the partition selected.

fftpart partitions a vector according to the *FFT-Radix* K[5] algorithm where K is a power of 2. For example, when $K = 2$ the algorithm computes FFT for vector of size N by computing FFT on 2 sub-vectors of size $\frac{N}{2}$ formed from the original vector by grouping the odd and even index positions, respectively.

OS-Join combines logical sub-windows, one from each parallel SQF computation, into one logical result window. It is a form of equijoin on the ordering components of the windows that in addition takes as a parameter a window transformation function *combinef* that specifies how the logical result window is computed from the sub-windows. OS-Join also takes care of preserving the order of the result windows by processing sub-windows in chronological order.

The following call to *OS-Join* combines the result sub-streams s21 and s22 from the compute phase by calling the window transformation function *fft3combine*. It uses the FFT-Radix algorithm to compute the result vector components from the sub-vectors.

```
OS-Join({s21,s22},"fft3combine");
```

5 Experimental Results

In this section we present the experiments we conducted in order to investigate how the two stream partitioning strategies scale and when it is favorable to

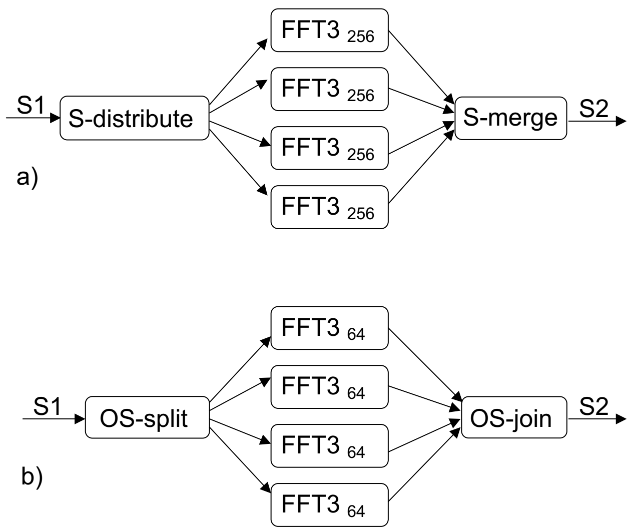


Figure 7: *Parallel strategies with flat partitioning in 4 (a)Window Distribute with Round Robin (b)Window Split with fft3part and fft3combine*

use operator dependent window split that utilizes knowledge about the SQF semantics.

The experimental set-up included three main strategies for the example SQF *fft3*. The central execution on a single node(Figure 4a) is a reference strategy. The second strategy is window distribute (WD) using Round Robin(Figure 4b). The third strategy is window split (WS) using FFT-dependent split and join operators (Figure 4c). In all the cases synchronization of the partitions after the parallel execution is performed and taken into account in the measurements.

The parallel strategies WD and WS were tested for degree of parallelism 2, 4 and 8. Figure 7 shows the data flow graphs for degree of parallelism 4.

For degree of parallelism 4 we considered in addition a distributed implementation of partition and combine phases as hypernodes forming a tree structure as shown in Figure 8.

A potential advantage of such tree-structured partitioning is that it allows for scaling the partition and combine phases with higher degree of parallelism. The tree structure in the example has 2 levels where each partitioning node creates 2 partitions. Analogously, the tree-structured combine phases have 2 levels that combine the results from 2 partitions.

The experiments were done on a cluster computer with processing nodes having Intel(R) Pentium(R) 4 CPU 2.80GHz and 2GB RAM. The nodes were connected by a gigabit Ethernet. The data was produced by a digital space receiver. TCP was used for communication between GSDM working nodes.

We measure the performance of the strategies by the time it takes to process

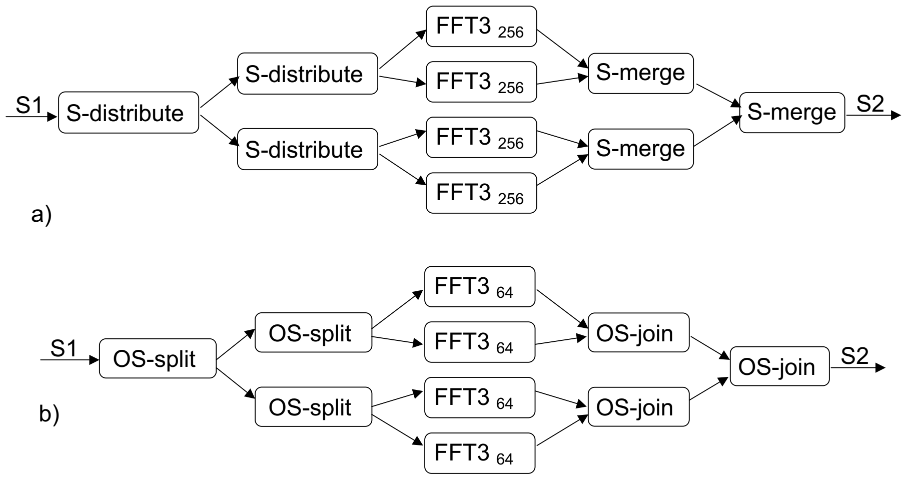


Figure 8: *Parallel strategies with tree partitioning in 4 (a)Window Distribute with Round Robin (b)Window Split with FFT3part and FFT3combine*

a stream segment of the same total size of 74MB. The segment contains 2MB signal samples for each of the 3 channels and was chosen in such a way that even the fastest strategies run long enough so that the slow start-up of TCP communication is stabilized. To investigate peak throughput the tests were run with increasing input rates until the point where the internal stream buffers started to grow, indicating overload. All the diagrams show execution times for such loss-less maximum throughput.

An important metrics for any parallel system is the scale up. It measures in our case the effect of the logical window size on the performance. Many scientific stream functions need to scale with the increase of the logical window size, e.g. to improve the precision of the results. In order to investigate the scalability with respect to the window size, six different logical window sizes from 256 elements (9KB) to 8192 elements (289KB), were used in all of the experiments.

Another important parallel performance metric is speed up. It is the ratio of the time elapsed in the central execution towards the time elapsed in the parallel execution for the same problem size, which in our case means the same logical windows size. In order to analyze the speed up we also ran the central reference strategy for all window sizes.

The execution of distributed scientific stream queries combines expensive computations with high volume communication. In order to investigate the importance and the effect of each of them on the total data flow performance, we ran two sets of experiments - one with highly optimized *fft3* function implementation and one with a slow implementation, where we deliberately introduced some delays in the FFT algorithm. Figure 9 shows the execution times of FFT

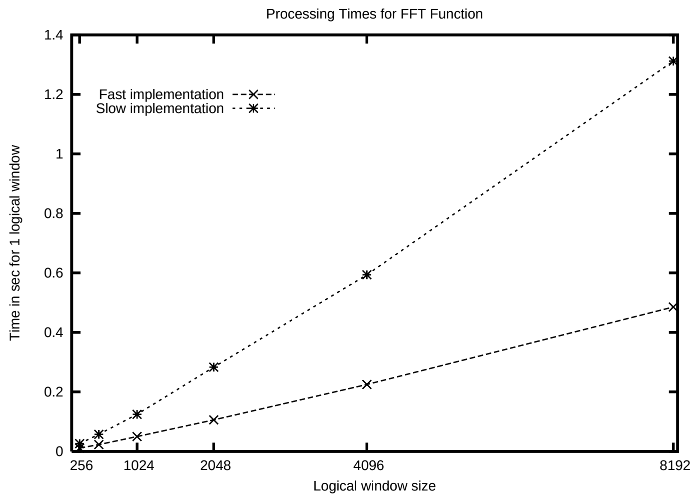


Figure 9: *Times for FFT implementations*

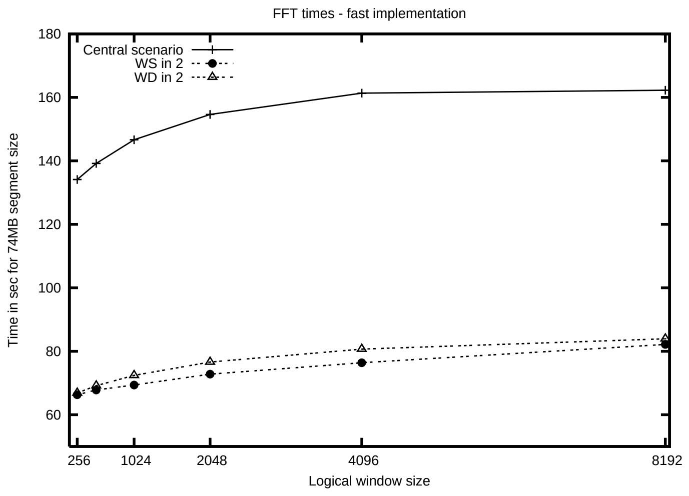
implementations for a single logical window of different sizes.

Figure 10 illustrates that the total elapsed time increases with the increase of the logical window size for the central and both parallel strategies with degree of parallelism 2 and in both fast and slow experiment sets. In the central case, this increase corresponds to the FFT operation complexity, $O(n \log n)$. We observe similar behavior in the parallel case with degree 2 since the total time spent at FFT processing nodes is higher than the time spent on splitting and merging nodes and thus, FFT performance determines the total data flow graph throughput. The WS strategy is faster than the WD strategy, since the parallel FFT processing nodes work on logical windows with vectors having size smaller by a factor of two than the vector size in WD strategy. Given the operator complexity this results in less total computational time.

Figure 11 illustrates that the fast and slow sets of experiments differ substantially for degree of parallelism 4. Here we compare four strategies: WD and WS were both implemented with *flat* partitioning, i.e. in a single node, and *tree* partitioning, i.e. by a structure of two partitioning levels.

We analyze first the performance of the experiments with fast FFT implementation (Fig. 11a). Figure 12a shows the maximum load of nodes in the different phases of the distributed data flow. The total time spent on FFT nodes is smaller than the time spent in partition and combine phases for all strategies and logical window sizes. Thus, the total data flow scalability is limited by the performance of the partitioning and combining phases that mainly communicate data. Even though the compute phase of WS is more efficient than the compute phase of WD, the system cannot benefit from this since it is the partition and combine phases that are main bottlenecks and limit the flow.

The WS strategy has more expensive operator dependent splitting and merg-



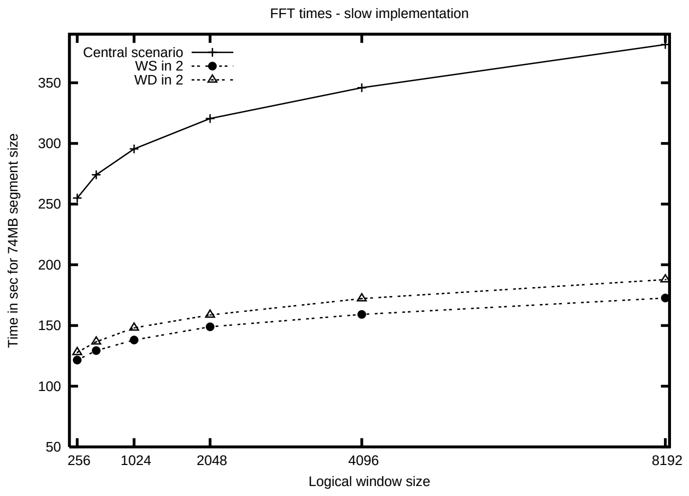
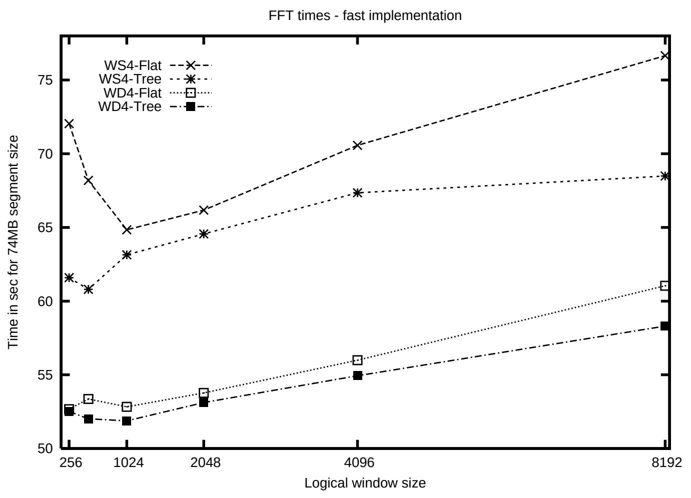


Figure 10: *FFT times for central and parallel in 2 execution (a)Fast implementation (b)Slow implementation*

ing SQFs than the WD with RR strategy and has consequently smaller total throughput. For example, the *OS-Split* using *fft3part* copies vector elements in order to create partitioned logical windows and the *OS-Join* computes the result windows using *fft3combine*, that executes the last step in FFT-Radix algorithm. The computation involves one multiplication and one sum of complex numbers for each element of the vector components of the result window. For WS4-Flat strategy with degree of parallelism 4 both *fft3part* and *fft3combine* are more expensive than the corresponding functions in the outermost nodes of



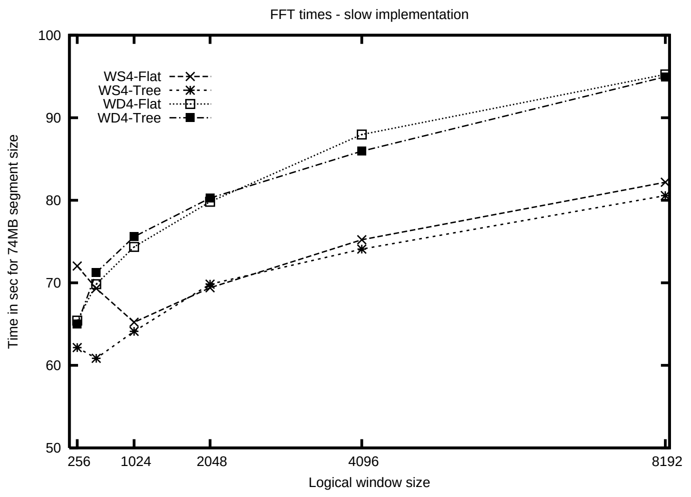


Figure 11: *FFT times for parallel in 4 execution (a)Fast implementation (b)Slow implementation*

WS4-Tree strategy where they have degree of parallelism 2. This explains the worst performance of WS4-Flat among all the parallel strategies with degree 4.

In addition the performance of both WS strategies decreases when the window size is under some threshold, 512 for WS4-Tree and 1024 for WS4-Flat, due to the memory management overhead for bigger number of small windows for the same total stream segment size.

For both WD and WS strategies the distributed tree-structured partition and combine phases show better performance than the flat one. The advantage is

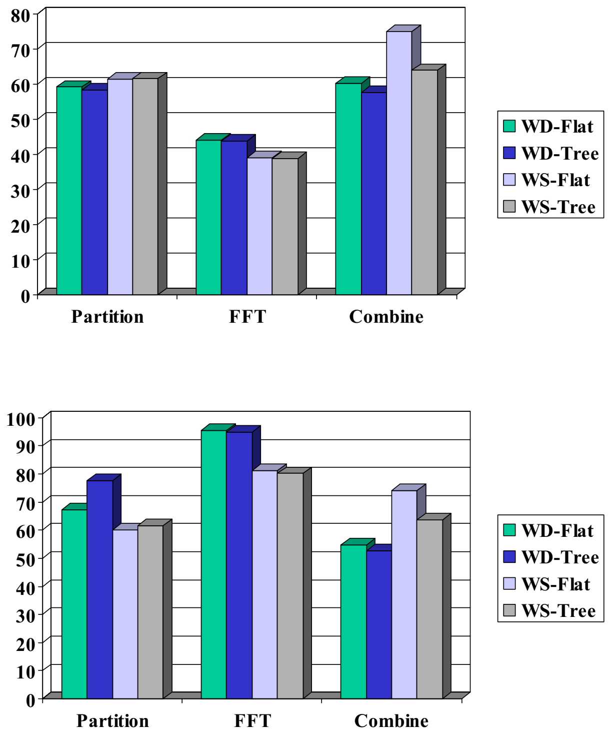


Figure 12: *Real time spent in partition, FFT, and combine phases of the parallel-4 strategies for logical window of size 8192. a)Fast implementation b)Slow implementation*

more substantial for big logical window sizes, which means that tree-partitioning structure has better scalability with respect to the logical window size. The main disadvantage is that it utilizes more executing nodes than the corresponding flat partitioning strategies to provide the same degree of parallelism for the main FFT operator. Figure 13 shows real time spent in communication, operator processing, and system tasks in the partition and combine phases using the WD-Tree and WD-Flat strategies for windows of size 8192. Here the operator time is the total execution time of SQFs *RRSplit* and *S-Merge*. The communication time includes the total overhead of transferring data between the GSDM and the communication subsystem. The figure illustrates the advantage, though a small one, of the tree structured partition and combine because of the better communication times in the outermost nodes of the partition and combine phases due to less overhead for smaller number of TCP connections.

In the experiments with slow FFT implementation for degree of parallelism 4 the compute phase is slower than partition and combine phases (Fig. 12b).

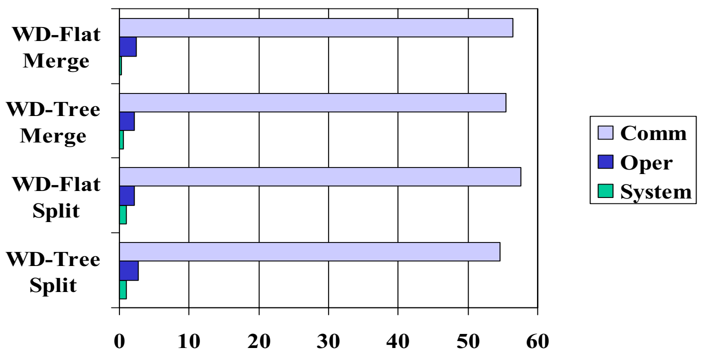


Figure 13: *Times for WD-Tree and WD-Flat with RR, window size 8192. Profile of splitting and merging nodes on communication/processing/system time in sec.*

Therefore, the WS4-Tree strategy has a better performance than WD with RR partitioning for all window sizes while WS4-Flat has about the same performance except a decrease for sizes less than 1024. For these small logical window sizes the WS4-Flat split and join nodes are more loaded than the FFT processing nodes.

As a conclusion, the strategy that gives the best flow depends on the ratio between the computational and communication costs. When, for a particular degree of parallelism and logical window size the nodes in the compute phase are loaded less than the nodes in the partition and combine phases, the latter ones become the bottleneck that limits the flow. These nodes spend most of the time in communicating data and a little percentage in the SQF that partitions or combines the data.

In our setting for fast FFT implementation and degree of parallelism 4, the partition and combine phases of WD are more efficient than the corresponding phases of WS due to simpler partitioning and combining algorithms.

However, if the nodes in the compute phase are loaded more than the nodes in the partition and combine phases, the compute phase limits the throughput. This occurs in our setting for slow FFT implementation and degree of parallelism 4, as well as for both implementations with degree of parallelism 2. In these settings the window split strategy showed to be more efficient than window distribute since it utilizes knowledge about FFT semantics to make the computation more efficient.

The ratio between the computational and communication cost is used as a basis to determine the meaningful degree of parallelism for a given SQF. For example, we also measured both parallel strategies with degree of parallelism 8. The system did not benefit from the larger number of parallel nodes and showed even worse times, since the compute nodes were much faster than the partition and combine nodes.

When choosing the best strategy the system must take into account the total

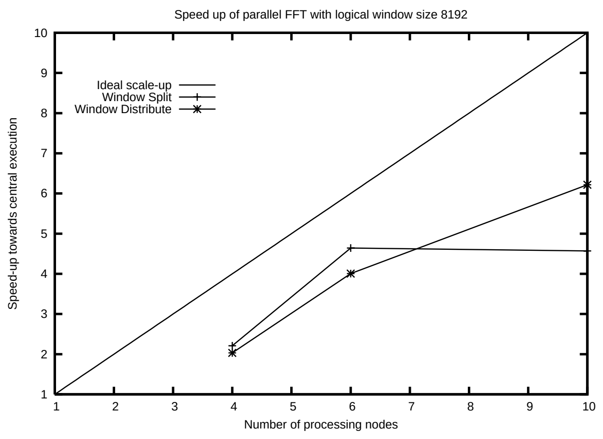


Figure 14: *Speed up of parallel FFT strategies for window size 8192.*

number of execution nodes available. Given an expensive function (slow FFT version), Figure 14 shows that the window split strategy has better speed-up than window distribute when resources are limited to a small total number of processing nodes. For example, when resources are limited to 6 computational nodes, 2 of which are dedicated to split and join, WS achieves a speed up of 4.6 for all window sizes while WD has a speed up of 4. For bigger number of nodes, e.g. 10 in the diagram, window distribute using RR shows better result.

6 Conclusions and Future Work

We presented an extensible stream database system where continuous queries (CQs) on data streams are executed as distributed data flow graphs containing stream query functions (SQFs). An SQF is a query over logical stream windows. The data flow graphs are defined using a user-extensible library of data flow distribution templates (DFDTs).

For example many expensive computations use a lattice shaped distribution pattern for scale-up with partition, compute, and combine phases. Using a generic DFDT for such lattice shaped distributions, we implemented two overall stream partitioning strategies, *window split* and *window distribute*.

Both strategies are customizable with stream partitioning and combining functions as parameters. Window split allows to utilize knowledge about SQF semantics to achieve better performance on the parallel computing nodes for expensive SQFs. By contrast, window distribution partitions data independent of SQF.

We evaluated the strategies in a cluster environment with real scientific application data. The application requires scalability in both data throughput and window size. We measured how the total loss-less throughput scales as the window size increases.

The experiments showed that window split is better than window distribute when the compute phase is more loaded then the partition and combine phases. This happens when executing with limited resources an SQF that is increasingly more expensive for larger windows, such as FFT.

By reducing the size of windows for SQFs with higher than linear complexity the total processing time is reduced, thus increasing the total throughput.

When the partitioning nodes are more loaded than the computational ones, the scalability of the entire data flow graph is limited by the scalability of the partitioning strategy itself. Therefore it is favorable to use a fast partitioning strategy, i.e. window distribution with Round Robin in our experimental settings.

In our current system we utilize in a training mode the parameterized high level DFDT specifications to vary the degree of parallelism and the partitioning strategies. A built-in performance monitoring sub-system measures the performance of different data flow graphs in order to find the optimal one.

In our continuing work we will investigate how to utilize adaptively query monitoring and knowledge about the trade-offs of the partitioning strategies during the query execution.

We are also investigating how GSDM can utilize computational Grids for stream query executions.

References

[1] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[2] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[3] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD Conf.*, 2003.

[4] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[5] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[6] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60, 2002.

[7] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[8] Kenneth W. Ng and Richard R. Muntz. Parallelizing user-defined functions in distributed object-relational dbms. In *IDEAS*, pages 442–445, 1999.

[9] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[10] Mehul A. Shah, Joseph M. Hellerstein, and Eric A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD Conference*, pages 827–838, 2004.

[11] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[12] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX*, pages 13–24, 1998.

[13] Alexander S. Szalay, Jim Gray, Ani Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The SDSS SkyServer: public access to the sloan digital sky server data. In *SIGMOD Conference*, pages 570–581, 2002.

[14] LOIS the LOFAR Outrigger In Scandinavia. http://www.lois-space.net/

[15] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[16] Richard H. Wolniewicz and Goetz Graefe. Algebraic optimization of computations over scientific databases. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 13–24. Morgan Kaufmann, 1993.

Recent technical reports from the Department of Information Technology

2004-052	Bharath Bhikkaji, Torsten Söderström, and Kaushik Mahata: *Recursive Algorithms for Estimating the Parameters in a One Dimensional Heat Diffusion System: Derivation and Implementation*
2004-053	Bharath Bhikkaji, Kaushik Mahata, and Torsten Söderström: *Recursive Algorithms for Estimating the Parameters in a One Dimensional Heat Diffusion System: Analysis*
2004-054	Lars Ferm, Per Lötstedt, and Paul Sjöberg: *Adaptive, Conservative Solution of the Fokker-Planck Equation in Molecular Biology*
2004-055	Per Lötstedt, Jonas Persson, Lina von Sydow, and Johan Tysk: *Space-Time Adaptive Finite Difference Method for European Multi-Asset Options*
2004-056	Erik Borälv: *Design and Evaluation of the CHILI System*
2004-057	Erik Borälv: *Evaluation and Reflections on the Design of the WeAidU System*
2004-058	Anna Eckerdal: *On the Understanding of Object and Class*
2005-001	Henrik Brandén and Per Sundqvist: *Preconditioners Based on Fundamental Solutions*
2005-002	Torbjörn Wigren: *MATLAB Software for Recursive Identification and Scaling Using a Structured Nonlinear Black-box Model — Revision 1*
2005-003	Claes Olsson: *Structure Flexibility Impacts on Robust Active Vibration Isolation Using Mixed Sensitivity Optimisation*
2005-004	Michael Baldamus, Joachim Parrow, and Björn Victor: *A Fully Abstract Encoding of the π-Calculus with Data Terms*
2005-005	Torsten Söderström: *Accuracy Analysis of the Frisch Scheme for Identifying Errors-in-Variables Systems*
2005-006	Agnes Runqvist, Magnus Mossberg, and Torsten Söderström: *On Optimal Sensor Locations for Nonparametric Identification of Viscoelastic Materials*
2005-007	Linda Brus: *Nonlinear Identification of an Anaerobic Digestion Process*
2005-008	Linda Brus: *Nonlinear Identification of a Solar Heating System*
2005-009	Claes Olsson: *Disturbance Observer-Based Automotive Engine Vibration Isolation Dealing with Non-Linear Dynamics and Transient Excitation*
2005-010	Pär Samuelsson, Björn Halvarsson, and Bengt Carlsson: *Cost-Efficient Operation of a Denitrifying Activated Sludge Process - An Initial Study*
2005-011	Per Carlsson and Arne Andersson: *A Flexible Model for Tree-Structured Multi-Commodity Markets*
2005-012	Milena Ivanova and Tore Risch: *Customizable Parallel Execution of Scientific Stream Queries*

UPPSALA
UNIVERSITET

April 2005
ISSN 1404-3203
http://www.it.uu.se/