Customizable Parallel Execution of Scientific Stream Queries

Milena Ivanova Tore Risch

Department of Information Technology
Uppsala University, Sweden
{milena.ivanova, tore.risch}@it.uu.se

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 parallel execution of query functions by reducing 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 evaluate these strategies for a typical high volume scientific stream application and show that window split is favorable when expensive queries are executed on limited resources, while window distribution is better otherwise.

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 stream database system, GSDM[1], where scientists can specify in a flexible way analyses as on-line distributed continuous queries (CQs) over the streams. Since the target scientific applications have high volume data and expensive computations, GSDM has been designed with a distributed and parallel architecture to provide scalability for both data volumes and computations.

The user specifies operators on stream data as declarative *stream query functions* (SQFs), defined over stream data units called *logical windows*. The SQFs may contain user-defined functions implemented in, e.g., C and plugged into the system. New types of stream data sources and SQFs over them can be specified.

GSDM provides the user with a generic framework for specifying distributed execution strategies by *data flow distribution templates* (or shortly templates). They are parameterized descriptions of CQs as distributed compositions of SQFs together with a logical site assignment for each SQF. For extensibility, a data flow distribution template may be defined in terms of other templates. For scalable execution of CQs containing expensive SQFs we provide a generic template for customizable data partitioning parallelism. The template, illustrated in Figure 1, 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.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

[1]Grid Stream Database Manager

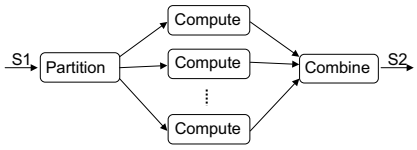

Figure 1: *A generic data flow distribution template for partitioning parallelism of expensive stream queries*

The generic distribution template 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 another template, *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 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. Hence, it strongly affects the performance of an entire class of application queries.

As a window distribute strategy, we provide *Round Robin stream partitioning* (RR) where entire logical windows of streams are distributed based on the order they arrive. In the combine phase, the result substreams 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 and therefore the compute phase of FFT is expected to run slower than with window split.

We evaluated the scalability of the two strategies in terms of maximum processing throughput for different logical window sizes. The experiments show that both partitioning strategies have advantages in specific situations. If the CQ is to be executed with limited resources, so that the load of the compute nodes exceeds the load of the partition and combine nodes, the window split is preferable since it utilizes query semantics to achieve a more scalable parallel execution. However, if the system has resources allowing a high degree of parallelism where partition and combine nodes become more loaded than the compute nodes, window distribute with RR may have better performance depending on the cost of partitioning and combining SQFs.

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

We have the following contributions:

- High-level data flow distribution templates specify distributed execution plans for CQs in terms of SQFs. This allows easy specification of CQs and various user-defined stream partitioning strategies. In particular we define a generic distribution template for partitioned parallel execution of expensive SQFs.

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

- The same generic distribution template is also used for defining *window distribute* strategies by parameterizing it with a partitioning template, *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 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 has a substantial computational cost and are executed on limited resources, while window distribute is better when resources are not limited.

- A distributed and parallel system architecture of the GSDM prototype has been designed, implemented and used in the experiments. A coordinator server compiles CQ specifications into distributed execution plans, sets up the GSDM execution nodes, and supervises the CQ execution.

The rest of the paper is organized as follows: Section 2 presents the related work on parallel stream processing. Section 3 presents GSDM architecture, data model, and query language. Different strategies for scalable execution of expensive stream operators are described in Section 4 and their performance is analyzed in Section 5. Section 6 summarizes.

2 Related work

Parallel CQs 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 (window split and window distribute 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 templates 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 the 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 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.

Some projects [2, 15] deal with processing of distributed streams but 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 a part of the 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.

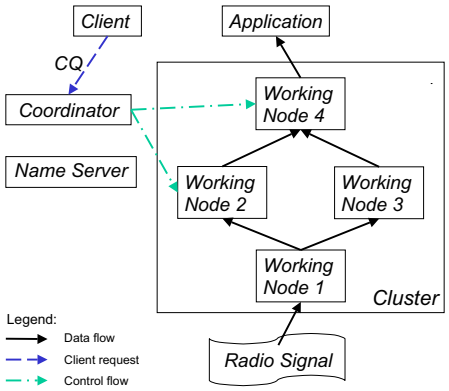

Figure 2: *An example of user interaction with distributed GSDM System*

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 an example of user interaction with the distributed GSDM system. The user submits a *continuous query (CQ)* specification to the *coordinator* through a GSDM *client*. The CQ specification contains the characteristics of stream data sources such as data types and IP addresses, the destination of the result stream, and what stream operators to be executed in the query. The CQ can be specified to run for a limited amount of time or until explicitly stopped.

The coordinator handles requests for CQs from the GSDM clients and manages CQs and GSDM *working nodes* (WNs). Given the CQ specification, the coordinator acquires resources from a cluster computer and constructs an execution plan as a distributed data flow graph where working nodes execute SQFs. The name server is a lightweight server providing the communication between all GSDM servers.

Figure 3 shows the software architecture of the co-

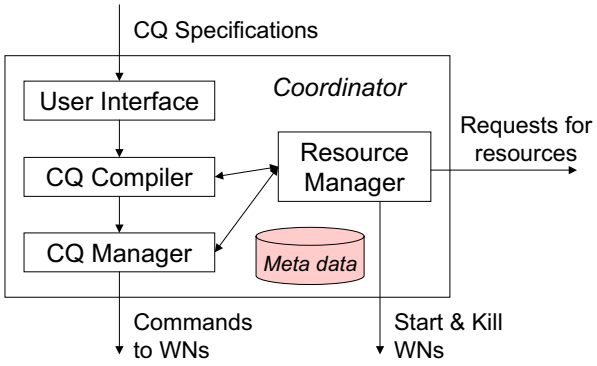

Figure 3: *Coordinator Architecture*

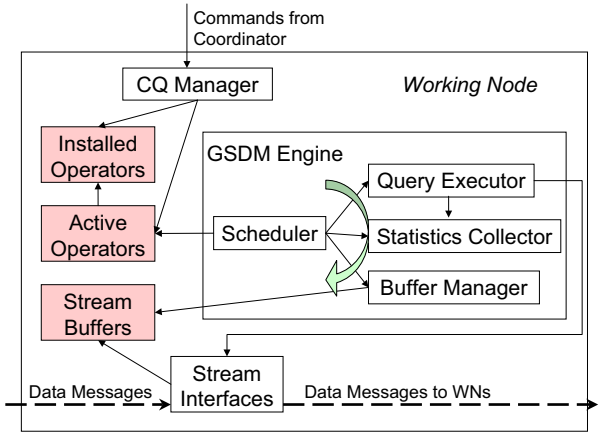

Figure 4: *GSDM Working Node Architecture*

ordinator. The *user interface* module provides GSDM client user primitives to specify, start and stop CQs. Given CQ specifications the *CQ compiler* produces distributed physical execution plans.

The *resource manager* module communicates with the resource managers of cluster computers to acquire processing resources. It also *starts* and *stops dynamically* working nodes when preparing or finishing the CQ execution. The architecture allows starting additional working nodes when necessary during the query execution, e.g., to increase the degree of parallelism.

The *CQ manager* installs, activates, and starts distributed execution plans by sending commands to the working nodes. The coordinator stores in its local database *meta-data* about the data flow graphs and the working nodes. The meta-data are accessed and updated by all the modules.

Each working node (Fig. 4) executes the part of the execution plan assigned to it and sends intermediate streams to the next working nodes in the plan. All SQFs installed at a node are organized in a hash-table, the *installed operators*. The input streams have buffers accessed by *stream interfaces* and a *buffer manager*.

GSDM working nodes have two server modes. When working in *regular server mode* they listen for messages coming on TCP sockets that typically contain calls to the WN *CQ manager* primitives for installation and activation of execution plans.

In *CQ server mode* the *GSDM engine* executes continuously SQFs over the incoming streams. The *scheduler* assigns processing resources to different tasks in the working node. The scheduler scans a list of active SQFs, *active operators*, schedules the operators according to a chosen *scheduling policy* and calls the *query executor*. The operators access stream data by calls to *stream interfaces* methods. A GSDM engine in CQ server mode also listens for messages as in the regular server mode, but in contrast the main body of incoming messages contain stream data. In addition, the scheduler calls some system tasks, such as buffer management and statistics collection.

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. 5) 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 an 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 instances of different 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. Every input stream maintains its own buffer and *cursor*, where the GSDM engine moves the cursor with the advance of SQF execution. 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
```

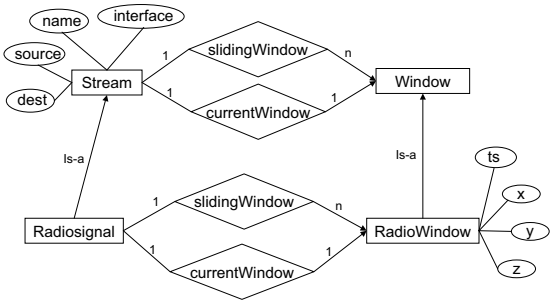

Figure 5: *Metadata of Radio Signal Stream Source*

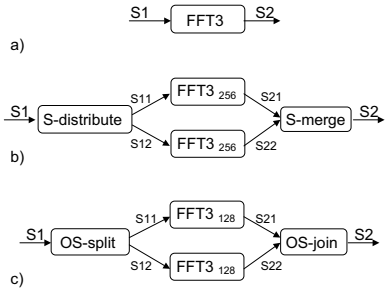

Figure 6: *(a)FFT Central Execution (b)Round Robin Strategy in 2 (c)Operator Dependent Strategy in 2*

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 type of stream source 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. Thus, stream access functions can be called possibly many times in SQF definitions providing referential transparency.

The streams in our application [14] are radio signals produced by digital space receivers represented by type *Radiosignal* (Fig. 5). 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 5 are generated when the user defines an application stream type by calling a system procedure, *create_stream_type*[3]:

```
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

To enable stream processing independent on a stream's physical communication media, the stream type needs interface methods for different physical media (function *interface*). The stream interface includes *open*, *next*, *insert*, and *close* methods. These methods have side effects on the state of the stream and are not called in SQF definitions. 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 a distributed composition of SQFs or other templates. Each template has a *constructor* that creates a data flow graph where vertices are SQFs assigned to logical execution sites. The arcs in the graph are stream producer-consumer relationships between SQFs. We provide a library of templates including the generic *PCC (Partition-Compute-Combine)* that specifies a lattice-shaped data flow graph pattern as in Fig. 1.

For example, the *window distribute* data flow in Figure 6b is created by:

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

The PCC constructor 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 6c 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 central node are specified by the *Central* template. For example, the following call generates the central data flow graph shown in Fig. 6a:

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

Notice that templates can also be used in place of SQF arguments in the constructor calls in order to generate complex graph structures. For example, the following call creates the distributed graph in Fig. 10a:

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

3.3 CQ Specification and Execution

The CQ specification contains i) the characteristics of input and result streams; ii) a data flow graph created by a call to a distribution template constructor; and iii) 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");
run(ws);
```

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 CQ will be executed on the nodes of cluster named *hagrid.it.uu.se*.

Given the CQ specification the *CQ compiler* produces a physical distributed execution plan. The run procedure starts the execution performing the following steps: i) the resource manager *starts* the GSDM working nodes on the cluster in *regular server mode*; ii) the compiled plan is *installed* on the working nodes distributed according to the execution site assignments; iii) the plan is *activated* by adding stream operators to the active operators list and performing initialization operations, such as creating stream buffers and opening TCP connections; iv) working nodes are switched into *CQ server mode* to start the execution.

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 streamed data and formulate the requirements for the strategies. In the next section, we evaluate their scalability.

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 plans.

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

1) It must preserve semantics of the SQF.

2) It must be order preserving.

3) It has to provide good load balancing.

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 template *S-distribute*. The chosen partitioning strategy affects the load balance of the parallel computing nodes.

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

Both partitioning strategies utilize the generic PCC template and preserve ordering by specialized stream operators in the combine phase.

4.1 Window Distribute Implementation in GSDM

For window distribute we provide a partitioning template, *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
```

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

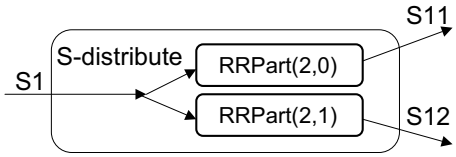

Figure 7: *S-Distribute template with RRpart as parameter and 2 partitions*

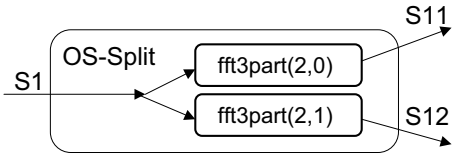

Figure 8: *OS-Split template with fft3part as parameter and 2 partitions*

S-Distribute takes as parameters the number of partitions n and an SQF *distrf* that selects the next logical window for a sub-stream. The *S-Distribute* constructor generates a distributing hyper-node in the data flow graph. Figure 7 illustrates the result of a call to the *S-Distribute* constructor 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, 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 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-out 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 current 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 6b).

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.

Window split partitioning is implemented by a partitioning template *OS-Split* and a combining SQF *OS-Join* with 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 a logical window is split into sub-windows. The following call to the OS-Split constructor creates the hyper-node in Figure 8:

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

The *splitf* parameter *fft3part* is an FFT-specific window transformation function that creates a sub-window from a logical window 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.

The function *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 from the compute phase by calling the FFT-specific window transformation function *fft3combine*.

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

The *fft3combine* function uses the FFT-Radix algorithm to compute the result vector components from the sub-vectors.

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 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 6a) is a reference strategy. The second strategy is window distribute (WD) using Round Robin (Figure 6b). The third strategy is window split (WS) using FFT-dependent split and join operators (Figure 6c). 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. The data flow graphs for degree of parallelism 4 are illustrated in Figure 9.

For degree of parallelism 4 we consider in addition two strategies, WD4-Tree and WS4-Tree (Fig. 10), where the partition and combine phases have a distributed tree-structured implementation. The tree structure in the example has two levels where each partitioning node creates two partitions and, analogously, each combine node recombines the results from two partitions. A potential advantage of such tree-structured partitioning is that it allows for scaling the partition and combine phases with higher degree of

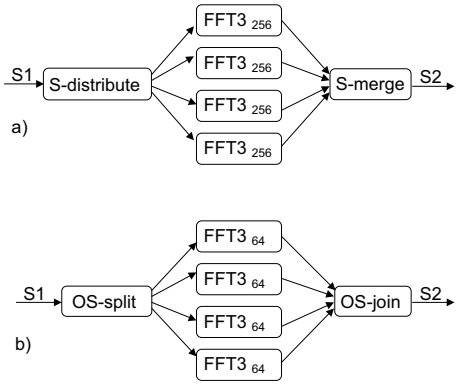

Figure 9: *Parallel strategies with flat partitioning in 4 (a)Window Distribute with Round Robin (b)Window Split with fft3part and fft3combine*

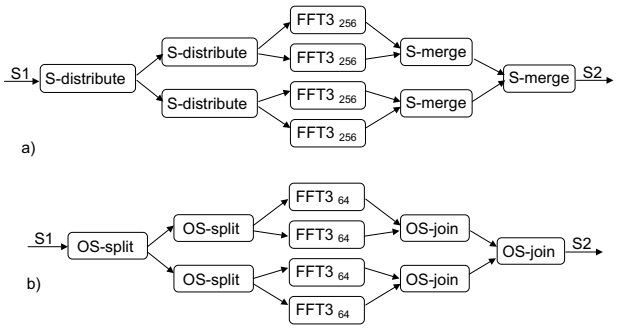

Figure 10: *Parallel strategies with tree partitioning in 4 (a)Window Distribute with Round Robin (b)Window Split with fft3part and fft3combine*

parallelism in cases when communication and/or computational costs in these phases limit the flow.

The experiments were done on a cluster computer with processing nodes having Intel(R) Pentium(R) 4 CPU 2.80G-Hz and 2GB RAM. The nodes were connected by a gigabit Ethernet. The communication between GSDM working nodes used TCP/IP protocol. The data was produced by a digital space receiver. For efficient inter-GSDM communication complex vectors were encoded in binary format when sent to and received from TCP/IP sockets.

An important metric for any parallel system is the scale up. In the case of parallel data flow graphs it depends both on the scalability of the SQF in the compute phase and on the scalability of the partitioning strategy itself in the partition and combine phases.

We measured the scalability in terms of two criteria: total maximum throughput and size of the logical windows with respect to the SQFs. We measured the throughput by the time to process a stream segment of 2MB signal samples for each of the 3 channels. With binary encoding the amount of communicated data was approximately 50MB. The size of the seg-

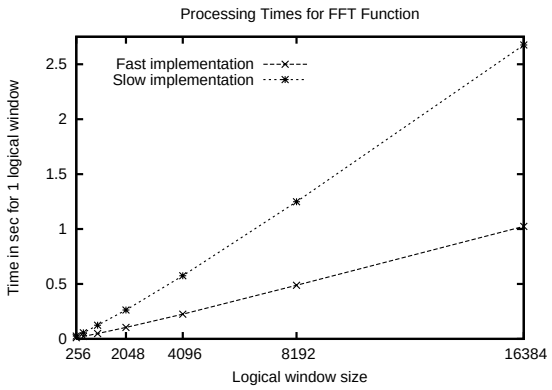

Figure 11: *Times for FFT implementations*

ment 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 stream rates until the point where the idle time of the most loaded nodes decreased under a threshold of 3%. All the diagrams show execution times for such maximum throughput.

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, seven different logical window sizes from 256 to 16384 signal samples 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 window 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 impact of each of them on the total data flow performance, we ran two sets of experiments - one with a highly optimized *fft3* function implementation and one with a slow implementation, where we deliberately introduced some delays in the FFT algorithm. Figure 11 shows the execution times of FFT implementations for logical windows of different sizes.

Figure 12 illustrates the increase of the total elapsed time with increase of the logical window size for the central strategy and both parallel strategies with degree of parallelism 2, and in both fast and slow experiment sets. For all the strategies the FFT processing nodes are most loaded and therefore the throughput is determined by the FFT operation complexity, $O(n \log n)$. 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

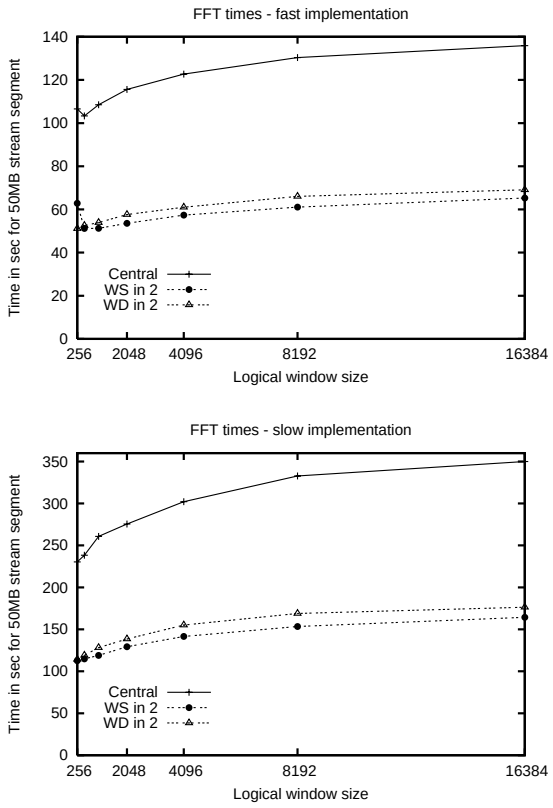

Figure 12: *FFT times for central and parallel in 2 execution (a)Fast implementation (b)Slow implementation*

a factor of two than the vector size in WD strategy. Given the operator complexity this results in less total time in the dominating compute phase.

In the fast experimental set for the smallest window size 256 we observed an exception of this behavior because of overhead per logical window due to increased memory management and communication.

Figure 13 illustrate fast and slow experimental sets for degree of parallelism 4. Here we compare four strategies: WD and WS with *flat* partitioning, i.e. in a single node, and WD4-Tree and WS4-Tree with *tree* partitioning, i.e. by a structure of two partitioning levels.

FFT processing nodes in WD4 strategies are more loaded than the partition and combine nodes for all logical window sizes. Hence, the WD throughput curves are similar to the central and WD2 strategies. This is illustrated in Figure 14 which shows the total real time spent in communication, computation, and system tasks in the different phases of the distributed data flow.

WD4-Tree strategy with a distributed implementation of the partition and combine phases did not improve the result substantially since the FFT nodes limit the flow and we skip therefore this strategy for

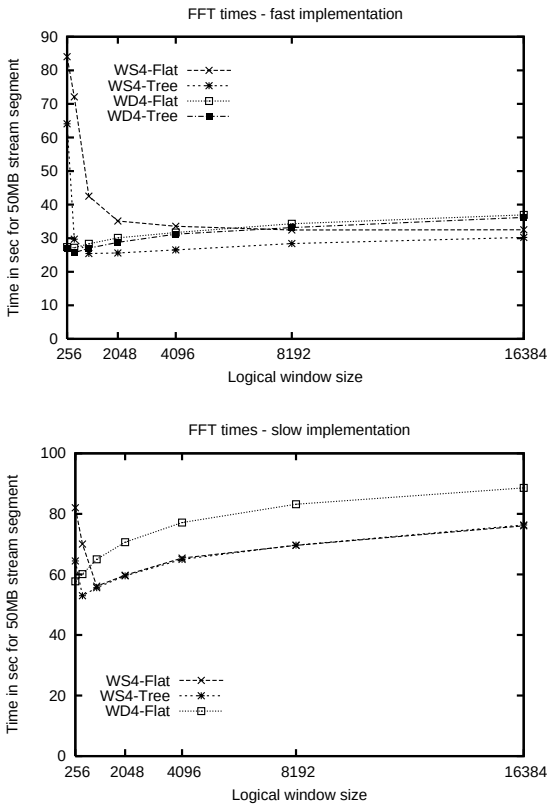

Figure 13: *FFT times for parallel in 4 execution (a)Fast implementation (b)Slow implementation*

the slow experimental set.

The partitioning and combining nodes of WS4 strategies are much more loaded than the correspondent nodes in WD. First, the WS strategy has more expensive operator dependent splitting and merging SQFs. 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 of the FFT-Radix algorithm. The computations involve 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 partition and combine nodes of WS4-Tree strategy where they process 2 partitions.

The second source of higher load of WS partition and combine nodes is that they communicate more logical windows with smaller size compared to WD strategies. Therefore, the overhead per logical window is bigger there. This overhead is smaller for WS4-Tree strategy than for WS4-Flat strategy since the outermost partition and combine nodes communicate an amount of logical windows smaller by a factor of 2. As a result the combining node in WS4-Flat strategy

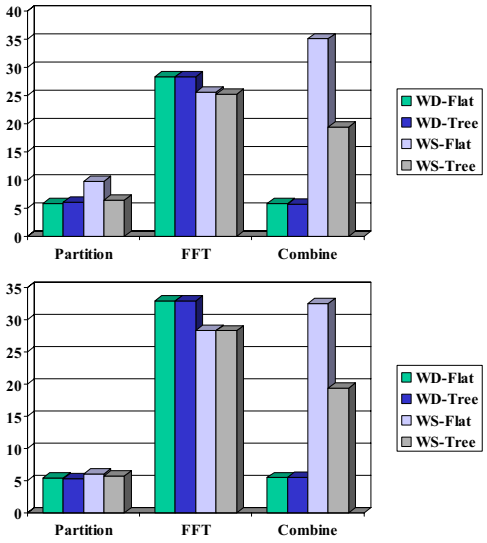

Figure 14: *Real time spent in partition, FFT, and combine phases of the parallel-4 strategies. a)Logical window of size 2048 b)Logical window of size 8192*

is the bottleneck for windows of size smaller than 512 in the slow experimental set and for all window sizes in the fast experimental set. The load of this node determines the maximum throughput of the data flow. Even though the compute phase of WS is more efficient than the compute phase of WD, the system cannot benefit from this because of the combine phase limitation.

In the slow experimental set FFT processing nodes in WS-Flat strategy are more loaded than the combining node for logical windows of size bigger than 512. Hence, the throughput curve for these sizes follows the FFT complexity behavior.

The WS4-Tree throughput curves are similar to WS4-Flat curves. In the slow experimental set and logical windows of size higher than 512 the FFT nodes have the highest load. Therefore WS4-Tree and WS4-Flat strategies have the same throughput for these sizes. For window sizes smaller than 1024 in the slow experimental set as well as for all window sizes in the fast experimental set, the combining node limits the flow. Hence, WS4-Tree strategy shows a higher throughput since its combining node is less loaded.

Table 1 illustrates the percentage of the total time spent in communication in the partition, compute, and combine phases for the logical window size 8192.

As a conclusion, the strategy that gives the maximum throughput depends on the load balancing between partition, combine and compute nodes.

When, for a particular degree of parallelism and logical window size 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 occurred for all experiment with all WD strategies and for the WS strategy with degree of parallelism 2 for both experimental sets, and in the slow exper-

	Part	Comp	Comb
WS Proc	0.42	57.66	13.8
WS Comm	15.94	2.82	5.17
WS Comm %	95%	4.6%	26.7%
WD Proc	0.04	62.95	0.09
WD Comm	7.56	2.72	5.01
WD Comm %	93.9%	4.1%	91.2%

Table 1: *Communication and computational costs in different PCC phases*

imental set with degree of parallelism 4 and logical windows of size bigger than 512. 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 compute phase more efficient.

However, if the nodes in the partition and combine phases are less loaded than the nodes in the compute phase, the former ones become the bottleneck that limits the flow. This situation occurred for WS strategies with degree of parallelism 4 for all window sizes in the fast experimental set and for window sizes smaller than 1024 in the slow experimental set. Which of the strategies has higher throughput in this situation depends on the proportion between the load of WS combining nodes and WD computing nodes. Figure 14a illustrates that WD4 strategies have higher throughput than WS4-Flat strategy for logical windows of size 2048 where the WS combine node is more loaded than WD compute nodes. However, the WS4-Tree strategy has more efficient combine node and is best for this window size. This also confirms that the tree-structured partitioning increases the throughput by scaling the combine phase.

Figure 14b illustrates that for windows of size 8192 and bigger the WS combine nodes becomes more efficient than the WD compute nodes and therefore both WS strategies have higher throughput than WD strategies.

Figure 15 shows that the window split strategy has better speed-up than window distribute for an expensive function (slow FFT version) and limited resources. 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 4.72 for window size 8192 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.

Optimal Data Flows

In presence of user-defined expensive queries and varying execution environments, it is hard to estimate processing and communication costs in order to construct a precise cost model for cost based query optimization. Instead we envision to run the system in a training

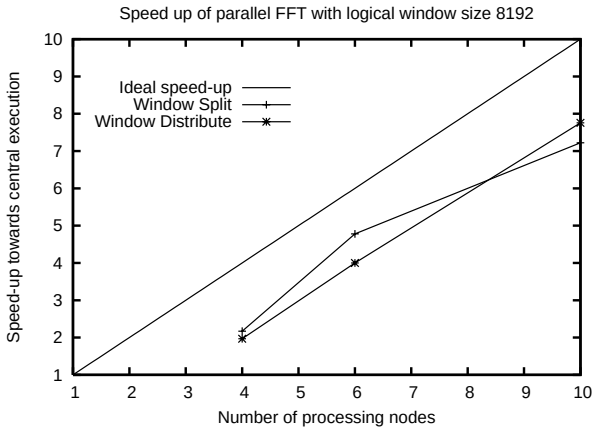

Figure 15: *Speed up of parallel FFT strategies for window size 8192.*

mode to find the best plan among a space of parallel plans. Using the experimental results we suggest the following heuristics to generate the plan space:

- Given a number of nodes, the optimal degree of parallelism for a given PCC parallel strategy can be found by: i) start with 2 partitions and measure the load of partition, compute and combine nodes. ii) increase the degree of parallelism while there are available resources and the compute nodes have highest load;
- Choose the best plan among the optimal plans for all parallel strategies. The choice can be based on maximum throughput and/or resources used.

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.

Many expensive computations use a lattice shaped distribution pattern for scale-up with partition, compute, and combine phases. Using a generic template, PCC (Partition-Compute-Combine) 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 ap-

plication 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 than the partition and combine phases. This happens when executing with limited resources an SQF which 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 in the compute phase 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 template specifications to vary the degree of parallelism and the partitioning strategies. A built-in performance monitoring subsystem 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.

Acknowledgments

This work has been supported by VINNOVA under contract #2001-06074. We would like to thank the team of prof. Bo Thidé at Swedish Institute of Space Physics and Uppsala University for the useful discussions and application problems and data provided.

References

[1] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the 28th VLDB Conf.*, pages 215–226, 2002.

[2] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the CIDR Conf.*, 2003.

[3] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the ACM SIGMOD Conf.*, 2003.

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

[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[6] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the ACM SIGMOD Conf.*, pages 49–60, 2002.

[7] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, Mayur Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the 1st CIDR Conf.*, 2003.

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

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

[10] M. A. Shah, J. M. Hellerstein, and E. A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the ACM SIGMOD Conf.*, pages 827–838, 2004.

[11] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the ICDE Conf.*, pages 25–36, 2003.

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

[13] A. S. Szalay, J. Gray, A. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and Jan van den Berg. The SDSS SkyServer: public access to the sloan digital sky server data. In *Proc. of the SIGMOD Conf.*, pages 570–581, 2002.

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

[15] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of the 29th VLDB Conf.*, pages 333–344, 2003.

[16] R. H. Wolniewicz and G. Graefe. Algebraic optimization of computations over scientific databases. In *Proc. of the 19th VLDB Conf.*, pages 13–24, 1993.