ACTA UNIVERSITATIS UPSALIENSIS
Uppsala Dissertations from the Faculty of Science and Technology
66

Milena Ivanova

Scalable Scientific Stream Query Processing

UPPSALA
UNIVERSITET

Dissertation at Uppsala University to be publicly examined in MIC campus, room 1211, Polacksbacken, on Monday, November 7, 2005 at 13:15, for the Degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Ivanova, M. 2005. Scalable Scientific Stream Query Processing. Acta Universitatis Upsaliensis. *Uppsala Dissertations from the Faculty of Science and Technology* 66. 137 pp. Uppsala. ISBN 91-554-6351-7

Scientific applications require processing of high-volume on-line streams of numerical data from instruments and simulations. In order to extract information and detect interesting patterns in these streams scientists need to perform on-line analyses including advanced and often expensive numerical computations. We present an extensible data stream management system, GSDM (Grid Stream Data Manager) that supports scalable and flexible continuous queries (CQs) on such streams. Application dependent streams and query functions are defined through an object-relational model.

Distributed execution plans for continuous queries are specified as high-level data flow distribution templates. A built-in template library provides several common distribution patterns from which complex distribution patterns are constructed. Using a generic template we define two customizable partitioning strategies for scalable parallel execution of expensive stream queries: window split and window distribute. Window split provides parallel execution of expensive query functions by reducing the size of stream data units using application dependent functions as parameters. By contrast, window distribute provides 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. Profile-based optimization automatically generates optimized plans for a class of expensive query functions. We further investigate requirements for GSDM in Grid environments.

GSDM is a fully functional system for parallel processing of continuous stream queries. GSDM includes components such as a continuous query engine based on a data-driven data flow paradigm, a compiler of CQ specifications into distributed execution plans, stream interfaces and communication primitives. Our experiments with real scientific streams on a shared-nothing architecture show the importance of both efficient processing and communication for efficient and scalable distributed stream processing.

Keywords: data stream management systems, parallel stream processing, scientific stream query processing, user-defined stream partitioning

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

ISSN 1104-2516
ISBN 91-554-6351-7

Printed in Sweden by Universitetstryckeriet, Uppsala 2005
Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala
www.uu.se, acta@ub.uu.se

To my parents and my son

Contents

1. Introduction

This Thesis presents the design, implementation and evaluation of Grid Stream Database Manager (GSDM), a prototype of an extensible stream database system for scientific applications. The main motivation of the project is to provide scalable execution of computationally expensive analyses over data streams specified in a high-level query language. This chapter presents the problem description and introduces background knowledge about the main enabling technologies for the GSDM prototype: database management systems (DBMSs), distributed and parallel DBMSs, and the evolving area of data stream management systems. At the end of the chapter, we summarize the main contributions of the Thesis and describe the Thesis organization.

1.1 Motivation

Scientific instruments, such as satellites, on-ground antennas, and simulators, generate very high volume of raw data often in form of streams [55, 82]. Scientists need to perform a wide range of analyses over these raw data streams in order to explore information and detect interesting events. Complex analyses are presently carried out off-line on data stored on disk using hard-coded predefined processing of the data. The off-line processing has a number of disadvantages that reduce the potential usage of the raw data. It creates large backlogs of unanalyzed data that prevents timely analysis after interesting natural events occurred. The high data volume produced by scientific instruments can also be too large to be stored and processed.

One of the driving forces behind the development of the GSDM prototype were the requirements of scientific applications from LOFAR/LOIS projects [54, 55]. The goal of the LOFAR project [54] in the Netherlands is to construct a radio telescope to receive signals from space and process them entirely in software. The LOIS (LOFAR Outrigger in Scandinavia, http://www.lois-space.net/) extends LOFAR with dedicated space radio/radar facilities and IT infrastructure with up to a few thousand units. As a part of LOIS a scientific instrument has been constructed that is a specialized three-poled antenna receiving radio signals. The signals are transformed from analogous into digital format, filtered initially by hardware, and sent in real time to the registered clients (receivers). At the receiver side there is need for a data stream process-

ing system that allows users, scientists in space physics, to detect interesting events in these high-volume signals by on-line analyses that include advanced and often expensive numerical computations.

The presence of high volume data and several users who want to perform similar analyses on the data suggest the use of database technology. Database management systems have proven their efficiency in managing large amounts of data, providing fast extraction of data of interest through declarative query languages, allowing for concurrent data access to multiple users, etc. However, several specific characteristics of scientific stream data and applications make them not fitting well in the current DBMSs.

This Thesis presents our efforts to bring the advantages of database technology to the class of scientific stream applications by the design and implementation of a data stream management system where users can express and efficiently execute expensive scientific computations as high-level declarative database queries towards the stream data.

The following three sections present the key technologies used in this Thesis. We end the chapter with a summary of our contributions.

1.2 Database Management Systems

Database management systems(DBMSs) (e.g. [34]) are software systems that allow for creating and managing large amounts of data. A *database* is a collection of data managed by a DBMS. The DBMSs *i)* allow users to create new databases and specify the logical structure of the data called *schema*; *ii)* allow users to query and modify the data using an appropriate language, called a *query language* or *data manipulation language*; *iii)* support secure storage of large amounts of data over long period of time; *iv)* provide concurrent access of multiple users to data.

DBMSs utilize various *data models*, which are primitives used for describing the structure of the information in the database, the *schema*. The evolution of DBMSs follows the development of new data models.

The first commercial DBMSs appeared in the late 1960s evolving from the file systems that were used as the main tool for data management until then. These database management systems utilized hierarchical and network data models that provided users with a view over data close to the physical data representation and storage. These early data models and systems did not support high-level query languages. In order to retrieve the required data, users had to navigate through a graph or tree of data elements connected by pointers. Thus, database programming required considerable effort and changes in the physical representation of data required rewriting database applications.

The *relational data model* proposed by Codd [26] at the beginning of 1970s

influenced significantly the development of database technology. According to this model, data is presented to the users in form of two-dimensional tables called *relations*. The relations have one or more named *columns* and data entries called *rows*, or tuples. The crossing points of columns and rows contain *data values* that can be of different atomic types, e.g. numbers or strings of characters. The simplicity of this conceptual view of data, close to the traditional non-electronic data representations, was one of the main reasons for the popularity it gained especially for business applications. At the same time, data is internally organized in complex data structures that allow for efficient access and manipulation.

In contrast to the earlier data models, the relational model allows for expressing queries in a very *high-level query language* which substantially increases the efficiency of database programming. The queries can be specified using two main formalisms: the procedural *relational algebra* and the declarative *relational calculus*. Based on these formalisms a number of query languages have been proposed, among which *Structured Query Language (SQL)* became the widely used standard. Instead of navigating through low-level data structures as in the early DBMSs, the users declaratively specify in SQL *what* data to be retrieved. The SQL query processing module takes care to translate the declarative query into an efficient execution plan specifying *how* data is retrieved. The separation of the query languages from the low-level implementation details provides another important feature: *data independence*. Two levels of data independence are distinguished: the ability to change the physical data organization without affecting the application programs is called *physical data independence*, while *logical data independence* insulates programs from changes to the logical database design.

By the 1990s the relational databases were commonly used in business applications. However, a number of applications from new domains, such as science, computer-aided engineering, and multimedia put requirements to the database technology that exposed the limitations of the relational model. Among these requirements is the need to represent more complex objects and new types of data such as audio and video, and to define specific operations over them. These applications became a driving force for the development of a new generation of DBMSs based on the *object-oriented (OO) data model*. All concepts in the OO paradigm are presented by *objects* classified in *classes*. A class consists of a type and possibly functions or procedures, called *methods*, which can be executed on objects of that class. The type system is very powerful: starting from *atomic types*, such as integers and strings, the user can build new types by using *type constructors* for record structures, collection types (sets, bags, arrays, etc.), and reference types. Record structures and collection operators can be applied repeatedly to construct even more complex types. Objects are assumed to have an *object identity (OID)* that identifies an

object independently of its value. Classes are organized in a class hierarchy, i.e. it is possible to declare one class *A* to be a *subclass* of another class *B*. In that case class *A inherits* all the properties of class *B*. The subclass may also have additional properties, including methods, that may be either in addition or in place of methods of the superclass.

Typically, OODBMSs were implemented by extending some object-oriented programming language, e.g. C++, with database features such as persistent storage, concurrency control, and recovery. The object-oriented data model is more powerful than the relational one when modeling real-world complex objects and may provide higher performance. However, early OODBMSs did not provide declarative query languages. Queries were specified by a navigation through a graph of objects where the arcs are defined by OIDs stored as attribute values of other objects.

During the last decade the development of both RDBMSs and OODBMSs followed a common goal, namely to combine in one system the declarative power of the relational DBMSs with the modeling power of the object-oriented paradigm. In the world of relational DBMSs most of the major vendors extended gradually their systems with object-oriented capabilities establishing in this way the new generation of *object-relational* DBMSs. The object-relational model includes the following main extensions of the relational model [34, 80]:

- Extensible base type system. New user-defined base data types (UDTs) can be introduced together with user-defined functions, operators, and aggregates operating on values of these types;
- Support for complex types by type constructors for rows (records of values), collections (sets, bags, lists, and arrays), and reference types;
- Special operations, methods, can be defined for, and applied to, values of a user-defined types;
- Types can be organized in a hierarchy with support for inheritance of properties from super types to subtypes;
- Unique object identifiers that distinguish an object independently of the object's data values.

Most of the object-oriented extensions above were included in the object-relational standard SQL:1999 [58] and its next edition SQL:2003 [31].

Simultaneously object-oriented DBMSs have been developing to incorporate declarative query languages as well in order to gain the advantages of the relational systems. The ODMG (Object Data Management Group) created a standard including *Object Definition Language (ODL)* and *Object Query Language (OQL)* [30]. OQL combines the high-level declarative programming of SQL with the object-oriented programming paradigm. It is intended to be used as an extension of some object-oriented host language, such as C++ or Java.

In this Thesis we utilize an object-relational model for modeling streams

with complex content. User-defined types represent both stream data sources, i.e. the scientific instruments, and numerical stream data produced by them. User-defined functions implement application-specific operations. Inheritance among UDTs allows for code re-use, and encapsulation provides for data independence of the application queries from the physical stream representations.

1.3 Distributed and Parallel DBMS

The architecture of a DBMS can be *centralized* or *distributed*. In centralized systems all the data is stored in a single repository and is managed by a single DBMS. In distributed database systems [64] data is stored in multiple repositories and is managed by a set of cooperating homogeneous DBMSs. The distributed DBMSs provide improved performance and reliability at the price of higher complexity. The distribution is manual and very often appears naturally as a consequence of distributed business activities, for example, a bank has one or several branches in different cities and countries and it is convenient to store and process branch-related data locally instead of in a single central database.

Parallel DBMSs [29] are a special kind of distributed database systems with transparent data distribution usually in one location to achieve better performance through parallel execution of various operations. The development of the parallel databases is in response to demands of applications that query extremely large databases or perform extremely large number of transactions per second, which the centralized DBMSs cannot handle.

The efficiency of parallel systems is evaluated by their *speedup* and *scaleup*. The speedup measures the ability of a parallel system to run a given task in less time by increasing the degree of parallelism. The scaleup measures the ability to process larger tasks in the same elapsed time by providing more resources. A parallel system has *linear* speedup when it executes a given task N times faster when having N times more resources. If the speedup is less than N the system is said to demonstrate *sublinear* speedup. A parallel system can also show *super linear* speedup when the increased number of resources leads to finer granularity of the subtasks so that, e.g., data fit into the cache and save time from intermediate I/O operations.

Two kinds of scaleup can be measured in a parallel DBMS [29]. The *batch scaleup* is the ability to execute large tasks when the database size increases. The *transaction scaleup* measures the ability to scale with the increase of both the database size and the rate of the transactions.

The utilization of parallelism in database systems is connected mostly with the relational data model and SQL. The set-oriented relational model and the declarative high-level query language allow for SQL compilers to automati-

cally exploit parallelism. The database applications do not need to be rewritten in order to benefit from the parallel execution provided implicitly by the underlying parallel DBMS. This parallel transparency makes them different from many other applications for parallel systems.

In this Thesis we utilize distributed and parallel DBMS technology to provide for scalable execution of queries with computationally expensive user-defined functions on data streams.

1.3.1 Parallel Database Architectures

Parallel architectures used for parallel database systems can be divided in three main classes: shared-memory, shared-disk, and shared-nothing.

In a *shared-memory architecture* processors have access to common memory, typically via a bus or an interconnection network. The advantage of this architecture is the extremely fast communication between processors via shared memory. However the scalability is limited since the bus or the interconnection network becomes a bottleneck. Large machines of this class are of *NUMA (nonuniform memory access)* type. The memory is physically distributed among the processors, but a shared address space and cache coherency are supported by the hardware, so that the remote memory access is very efficient. NUMA architectures require rewriting the operating system and the database engines.

In a *shared-disk architecture* processors have private memories, but access common set of disks via an interconnection network. The scalability is better than in the shared-memory architecture, but is limited by the common interconnection to the disks. The communication between processors is much slower than in shared-memory architectures since it goes through the communication network.

In a *shared-nothing architecture* each node of the machine consists of a processor, memory, and one or more disks. The processors communicate via a high-speed interconnection network. This architecture provides better scalability since it minimizes resource sharing and interference between processors. Memory and disk accesses are performed locally in a processor and only the queries and answers with reduced data sizes are moved through the network. Shared-nothing machines are furthermore relatively inexpensive to build. The main drawback is the high cost of communication between processors. Data are sent in messages that have considerable overhead associated with them.

The so-called *hierarchical architecture* combines some of the above architectures in several levels. The highest level consists typically of shared-nothing nodes connected via an interconnection network. Each of the nodes in its turn is a shared-memory or shared-disk machine. Thus, the hierarchical

architectures combine the performance of shared-memory with the scalability of shared-nothing architectures.

Even though the shared-memory architecture provides better performance due to more efficient interprocessor communication, the shared-nothing architecture is most commonly used for high-performance database systems, not at least because of its better cost-efficiency [29].

In the present work we use a shared-nothing architecture for stream data management where GSDM servers communicate over TCP/IP. This facilitates parallel processing on shared-nothing cluster computers, but also enables utilization of distributed resources, including resources on the Internet.

1.3.2 Types of Parallelism for DBMS

DBMSs can exploit different types of parallelism. *Inter-query parallelism* means execution of multiple queries generated by concurrent transactions in parallel. It is used to increase the transactional throughput, i.e. the number of transactions per second, but the response times of the individual transactions are not shorter than they would be if the transactions were run in isolation.

Intra-query parallelism decreases the query response time. In can be *inter-operator* parallelism, when operators in the query execution plan are executed in parallel on disjoint sets of processors, and *intra-operator* parallelism, when one and the same operator is executed by many processors, each one working on a subset of the data. The inter-operator parallelism can be *independent* or *pipelined*. In both cases the degree of parallelism is limited by the number of operators in the query plan that are independent or allow pipelining, which is typically not very large.

Intra-operator parallelism requires parallel implementation of the operators in the query plans. An operator is decomposed into a set of independent sub-operators, called *operator instances*. Data are assigned to different operator instances using some *data partitioning* strategy. Typical data partitioning strategies used in parallel implementations of the relational operators are Round Robin, hash and range partitioning [64]. The intra-operator data partitioned parallelism is the most important source of parallelism for the relational DBMSs.

Several factors in parallel query execution decrease the benefits of the parallelism. Among them are the processes' *startup costs*, the *interference*, when the processes compete for shared hardware or software resources, and *load imbalance*. In an ideal situation a task will be divided into exactly equal-sized subtasks. In reality, the sizes of subtasks are often skewed and the time of the parallel execution is limited by the time of the slowest subtask.

The extensibility of object-relational DBMSs with new UDTs and user-defined functions (UDFs) allows to utilize new techniques for data partition-

ing and parallel query processing. In addition to the parallel techniques for the relational DBMSs, *inter-function* and *intra-function parallelism* are possible in ORDBMS [63]. The inter-function parallelism allows independent or pipelined UDFs in a query to be executed in parallel. The intra-function parallelism allows a UDF over a single value to be broken into multiple instances that operate on parts of the value simultaneously. For example, a function over a single image can be written to work on a set of pixel rows. Therefore, intra-function parallelism requires to partition single valued data with respect to the UDF. Furthermore, the data partitioning techniques for intra-operator parallelism can be extended by using the result of a function or collection type values as a basis for hash or range partitioning. Such partitioning functions can utilize knowledge about the distribution or the structure of the data.

In this Thesis we provide a generic and declarative way to specify intra-function parallelism through stream data partitioning for computationally expensive functions on data streams defined through UDTs.

1.4 Data Stream Management Systems (DSMSs)

During the last couple of years the attention of the database research community has been attracted by a new kind of applications that require on-line analysis of dynamic data streams [10, 17, 22, 27, 53, 56, 81].

Examples include network monitoring applications analyzing Internet traffic, financial analysis applications that monitor streams of stock data reported form various stock exchanges, sensor networks used to monitor traffic or environmental conditions, or analyses of Web usage logs and telephone call records. The target problem of this thesis is on-line analyses of streams generated from scientific instruments and simulators, which is an another example of a data streaming application.

The applications get their data continuously from external sources, such as sensors, software programs or scientific instruments, rather than from humans issuing transactions. Typically the stream sources *push* the data to the applications. Usually data must be processed on-the-fly as it arrives, which puts high constraints on the processing time and memory usage, especially for streams with high-volume or bursty rates. Very often the applications are trigger-oriented where a human operator must be alerted when some conditions in the data are fulfilled.

A *data stream* is an ordered and continuous sequence of items produced in real-time [10, 36]. The stream can be ordered implicitly by the items' arrival times or explicitly by timestamps generated at the source. The streams are conceptually *infinite* in size and hence they cannot be completely stored, but once processed a stream item is discarded or archived. Since streams are produced

continuously in real-time, the total computational time per data item must be less than the average inter-arrival times between items in order for the processing to be able to keep pace with the incoming data streams. The real-time requirements necessitate main-memory stream processing where data can be spooled to disk only in the background. The system has no control over the order in which the data items arrive, either within a stream or across multiple streams, and must *react* to the arrivals [19]. Re-ordering of data items for processing purposes is limited by the storage limitations and the real-time processing requirements.

Queries over streams run continuously over a period of time and incrementally return new results as new data arrive. Therefore, they are named *continuous queries(CQs)*, or also *long-running* or *standing* queries [10, 36].

The specific characteristics of streams and continuous queries put the following important requirements on a *data stream management system (DSMS)*:

- The data model and query semantics must allow operations over sub-streams of a limited size, called *windows*;
- The data stream management system must provide a support for *approximate answers* of queries. The inability to store complete streams necessitates to represent them by approximate summary structures. Furthermore, data can be intentionally omitted by sampling or dropping data items to reduce the processing load for high volume or bursty streams, which also leads to approximate answers.
- Query plans for stream processing may not use blocking operators that require the entire input to be present before any results are produced.
- On-line stream algorithms are restricted to one pass over the data due to performance and storage constraints.
- Long-running queries may encounter changes in system conditions and stream properties during their execution lifetime. Therefore, an efficient stream management system should be able to automatically discover the changes and adapt itself to them.
- The presence of long-running queries and on-the-fly processing necessitates shared execution of multiple queries to ensure scalability. The shared execution mechanism must allow to easily add new queries and to remove old ones over time.
- Many applications have more strict real-time requirements where unusual values or patterns in the stream must be quickly detected and reported. Query processing in those cases aims to minimize the average response time, or latency, measured by the time a data item has arrived to the system until the moment when the result stream item is reported to the user.

Several DSMSs have been designed during the last years, mainly as academic research projects, and DSMSs are still rare in the commercial world.

Examples include Aurora [2], CAPE [70], Gigascope [27], NiagaraCQ [22], STREAM [59], Nile [41], Tribeca [81], and TelegraphCQ [19]. Gigascope and Aurora are examples of DSMS prototypes that are in production use. We will present the related DSMS projects in more details in Chapter 7.

Most of the existing prototypes are based on extensions of the relational model where stream data items are transient tuples stored in virtual relations. In object-based models [81] data sources and items are modeled as hierarchical data types with associated methods. In all the cases windows on streams are supported that can be classified in the following way:

- Depending on how the endpoints of a window move along the stream two sliding endpoints define a *sliding window*, while one fixed endpoint and one moving define a *landmark window*.
- When the window is defined in terms of a time interval it is *time-based* while *count-based* windows are defined in terms of number of items.
- Windows can be also distinguished based on the *update* frequency: eager re-evaluation updates the window upon arrival of each new data item, while lazy re-evaluation creates *jumping windows* updated at once for a number of arrived items.

The query languages of the systems based on the relational model have SQL-like syntax and support windows processed in stream order [6]. There are also procedural languages: e.g. in Aurora [2] the users construct query networks by connecting operator boxes via a graphical user interface.

Non-blocking stream processing is provided by three general techniques: windowing, incremental evaluation, and exploiting constraints. Any operator can be made non-blocking by limiting its scope to a finite window that fits in memory. Operators must be incrementally computable to avoid re-scanning the entire window or stream. Another mechanism to provide for non-blocking execution is to exploit schema or data constraints [13]. Schema-level constraints are for example pre-specified ordering or clustering in streams, while data constrains are special stream items referred to as *punctuations* [86] that specify dynamic conditions holding for all future items.

Ordering of stream data is defined through *timestamps* [79]. There are two general ways in which timestamps are assigned to stream items:

1. Elements are timestamped on entry to the DSMS using its *system time*;
2. Elements are timestamped at the sources before sending them to the DSMS using a notion of *application time*.

As an alternative to timestamps order numbers can sometimes be used. Timestamps associated with streams have an important role in stream query processing. For example, they can be used to determine which operator in the query plan to be scheduled next or to decide what data can be expired from the internal operator states. Furthermore, the system timestamps can be used at the end of the processing to compute the response time (latency) that an

item has spent in the system in order to check how well the application's QoS requirements are met [2].

Temporal databases also operate with system-supported timestamps. There are three notions of time defined in temporal database [47, 78]: a *valid time* of a fact is the time when the fact is true; a *transaction time* of a database fact is the time when the fact is stored in the database; and *user-defined time* is a domain of time values in which an attribute is defined and which is uninterpreted by the DBMS.

There is no notion of arrival time of data in the temporal databases. The arrival (or system) time in a DSMS is somewhat similar to the transaction time in sense that after that time the data item may be retrieved. The application time in a DSMS is similar to the valid time notion in a temporal DBMS, e.g. a sensor reading that is timestamped at the sensor can be interpreted as the valid time when this reading is true.

Temporal databases store temporal information associated with other data focusing on maintaining the full history of each data value over time. DSMS store temporarily the recent past of the stream and are more concerned to provide on-the-fly processing of new data items.

Sequence databases [72, 73] provide support for data over ordering domains such as time or linear positions. Thus, operators exploiting logical ordering of the data are analogous to stream operators, e.g. moving average over time-based windows. One important difference is that sequence databases assume having control over the order in which single and multiple sequences are processed, e.g. random access to individual elements based on their positions is provided. Since stream systems keep only the recent past of the streams rather than the entire sequences, the query processing is limited to be carried out as data arrive to the system.

In this Thesis we designed and implemented a main-memory continuous query engine for stream processing in real-time. The engine executes in a push-based manner operators over streams, which are window-based, order-preserving, and non-blocking. The GSDM is the first functioning DSMS prototype providing for scalable parallel processing of computationally expensive queries over stream data.

1.5 Summary of Contributions and Thesis Outline

We present the design, implementation, and evaluation of an object-relational data stream management system [44, 69, 48] for scientific applications with the following distinguishing properties:

- On system architecture we designed and implemented a distributed architecture consisting of a *coordinator* server and a number of *working*

nodes that run in a shared-nothing computer architecture. High-volume disk-stored databases traditionally limit the query processing to be performed close to the data and usually on dedicated resources. Main-memory stream processing releases this limitation and opens new opportunities and challenges for dynamic resource allocation. The GSDM system architecture allows for dynamic configurations on undedicated resources given that tools for dynamic resource allocation are provided.

- The system is built upon an object-relational model that allows specifying user-defined types for numerical data from scientific instruments and implement operations over them as user-defined functions.
- The object-relational model is used to represent types of stream sources organized in a hierarchy. The basic system functionality concerning streams is implemented in terms of a generic stream type from which stream sources of particular types inherit properties. The access to stream data on different communication and storage media is encapsulated in stream interfaces with a uniform set of methods. The system treats uniformly external streams, inter-GSDM streams, and local streams inside a GSDM node.
- We provide a framework for high-level specifications of data flow graphs for scalable distributed execution of CQs. In particular, we provide *partitioned parallel execution* of computationally expensive CQs. The parallel execution is customizable by specification of *user-defined stream partitioning* strategies.
- Two general strategies for partitioned parallelism were investigated, *window split* and *window distribute*. The window split strategy is an innovative approach that is a form of user-defined intra-function parallelism through object partitioning. Through the customization users provide the system with knowledge about the semantics of a user-defined function to be parallelized for the purposes of more efficient execution. Both partitioning strategies are specified in a uniform way by declarative *data flow distribution templates*.
- The core of a working node is a *CQ execution engine* that processes CQs over streams. Query processing is based on a data-driven data flow paradigm implemented in a distributed environment. The operators constituting the CQ execution plan run in a push-based manner.
- Different stream partitioning strategies are evaluated in a parallel shared-nothing execution environment using example queries from space physics applications over real data from scientific instruments [55].
- On query optimization, we develop a profile-based off-line optimization framework and apply it to automatically generate optimized parallel plans for expensive stream operations based on a data flow distribution template for partitioned parallelism.

The rest of the Thesis is organized as follows. Chapter 2 presents the software architecture of GSDM. Modeling of stream data, specification of continuous queries, and specification of distributed and parallel CQ execution through data flow distribution templates is given in Chapter 3. The two main stream partitioning strategies for scalable execution of expensive CQs are presented and experimentally evaluated in Chapter 4. Chapter 5 presents technical details related with definition and management of continuous queries at the GSDM coordinator, while Chapter 6 describes details about continuous query execution at working nodes. Chapter 7 analyses the requirements and possibilities for utilizing a data stream management system in computational GRID environments in a more general way than in a single shared-nothing cluster computer. Chapter 8 presents an overview of related research areas and prototype systems and puts the GSDM prototype in this context. Chapter 9 summarizes the Thesis and discusses future work.

2. GSDM System Architecture

This chapter presents the architecture of the Grid Stream Database Manager prototype - an extensible distributed data stream management system. We start with an example scenario illustrating how the distributed system components interact in order to execute users requests. The software architecture of the GSDM coordinator and working node servers is presented.

2.1 Scenario

Figure 2.1 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 operations to be executed in the query. The stream data source in the example is a scientific instrument that contains a specially designed 3-poled antenna for radio signals connected to a server with capabilities to broadcast the signal to a number of clients [55]. The CQ contains application-specific stream operations to compute properties of the radio signal that are interesting for the scientists. The result stream of the query is sent to an application that visualizes the computed properties of the signals.

Given the CQ specification, the coordinator constructs a distributed execution plan where GSDM *working nodes (WN)* execute operators on streams. The coordinator requests resources from available cluster computers and starts dynamically GSDM working nodes on the cluster nodes. Next, it installs the distributed execution plan on the nodes, starts the execution, and supervises it.

Each working node executes a part of the execution plan that is assigned to it and sends intermediate result streams to the next working nodes in the plan. In the example, WN1 partitions the radio signal stream into two sub-streams sent to WN2 and WN3, respectively. WN2 and WN3 perform an application stream operator on the sub-streams in parallel, and WN4 combines the result sub-streams and sends the final result stream to the specified destination address, where the visualization application is listening for a stream with specific data format.

The name server in the figure is a lightweight server that keeps track of the GSDM peers locations. In the scenario all working nodes run on a cluster

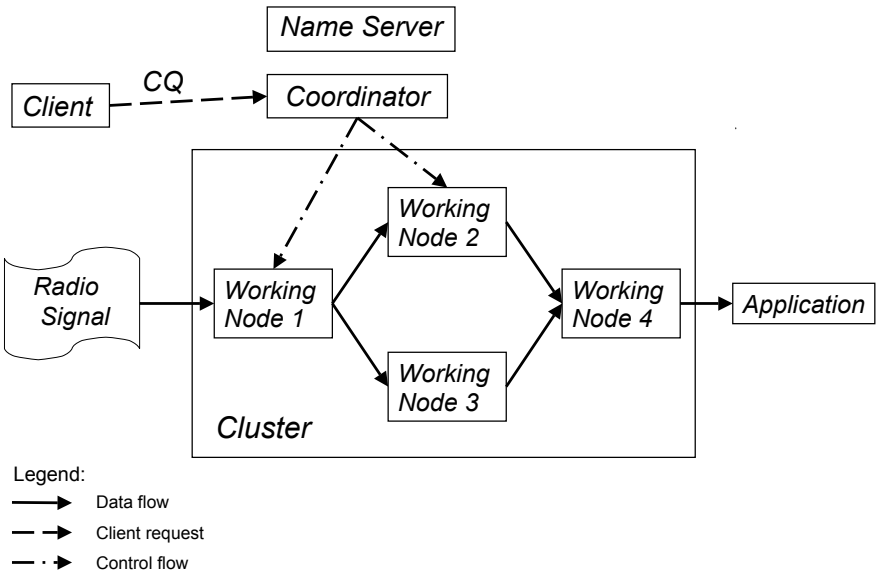


Figure 2.1: GSDM System Architecture with an example data flow graph

computer, while the client, the coordinator, the name server, and the application run outside the cluster. Alternatively, the coordinator and the name server can be also set up to run on the cluster.

2.2 Query Specification and Execution

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.

The GSDM system utilizes 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 an abstract system type *Stream* and stream elements are objects called *logical windows* that are instances of a user-defined type *Window*. 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. Different types of logical windows are represented as subtypes of the *Window*

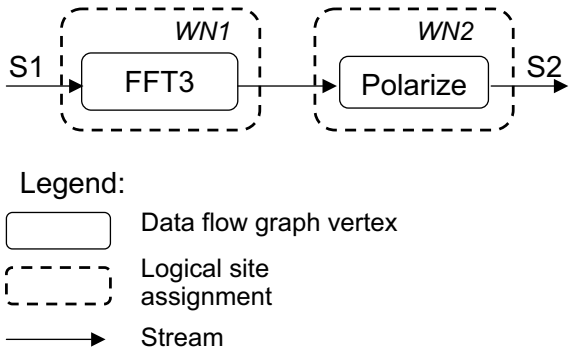


Figure 2.2: An example data flow graph

super-type and the stream sources with particular types of logical windows are represented as subtypes of the type *Stream*.

A *stream query function (SQF)* is a declarative parameterized query that computes a logical window in a result stream given one or several input streams and other parameters. SQFs are defined as functions in the query language of the system, AmosQL [5, 68].

An SQF is a *stream producer* with respect to its result stream and a *stream consumer* with respect to its input streams. We say that two SQFs have a *producer-consumer relationship* if the result stream of one of them is an input stream for the other.

A *continuous query (CQ)* is a query that is installed once and executed on logical windows of the incoming stream data to produce a stream of outgoing logical windows. A CQ is expressed in GSDM as a composition of SQFs connected by stream producer-consumer relationships. The composition has structure of a directed acyclic graph that we shall call a *data flow graph*. Figure 2.2 illustrates an example graph of two vertices annotated with two SQFs, named *fft3* and *polarize* respectively, and connected by a producer-consumer relationship.

Since GSDM is designed for distributed stream processing, it 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[1] for each SQF in the strategy. The typical template parameters are the SQFs composing the CQ and their arguments. For extensibility, a data flow distribution template may be used as a parameter of an-

[1] A *logical execution site* is a GSDM working node that will execute as a process on a computer, a *physical execution site*.

other template, which allows to construct complex distributed compositions of SQFs.

Each template has a *constructor* that creates a distributed data flow graph. Each vertex in the data flow graph is annotated with an SQF and the parameters for its execution. Each arc of the graph is a producer-consumer relationship between two SQFs. The SQFs are assigned to, possibly different, logical execution sites as specified by the template. We provide a library of templates specifying central execution, parallel execution, and pipelined execution of SQFs, as well as partitioning of a stream through a user-provided partitioning SQF. More details about the library will be presented in Chapter 3.

In order to specify a CQ the user chooses a template and calls its constructor providing the SQFs and their arguments as parameters of the call. For instance, the following call to a *pipe* template constructor creates the graph in Figure 2.2:

```
set p = pipe("fft3",{},"polarize",{});
```

The constructor will assign the two SQFs to two different logical execution sites, WN1 and WN2, for pipelined parallel execution. In this case the functions do not have non-stream parameters, which is denoted by $\{\}$[2].

The templates specify compositions of SQFs that are not connected to particular stream sources. Therefore, the user has to specify the characteristics of the stream data sources and the result stream. For each input stream the user provides its type, the source address of the program or instrument sending the stream, and stream interface to be used. Further, the user specifies the destination address to which the result stream should be sent and the stream interface to be used. For example:

```
set s1 = register_input_stream("Radio","1.2.3.4",
            "RadioUDP");
set s2 = register_result_stream("1.2.3.5",
            "Visualize");
```

In the example the user registers one input stream of type *Radio* accessible by a stream interface called *RadioUDP* from server with address *"1.2.3.4"*. The user also specifies a result stream that should be sent to a visualizing application on a given address using a stream interface called *Visualize*.

A complete CQ specification in GSDM contains both a data flow graph, specifying an abstract composition of SQFs, and input and output streams to which the graph shall be bound. For example:

```
set q = cq(p, {s1}, {s2});
```

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

creates a continuous query executing the SQFs specified in the data flow graph *p* over the input stream *s1* to produce the result stream *s2*.

Semantically, the result of an SQF is one output stream, but the system allows it to be replicated to many consumers. If multiple output streams are given in the CQ specification, the result of the CQ will be replicated to several applications.

Given the CQ specification, the CQ is then *compiled* in order to create an *execution plan* containing compiled SQFs and stream objects connecting each pair of SQFs for which a producer-consumer relationship has been defined. The compilation is done by a procedure *compile*, e.g.:

```
compile(q);
```

In order for a query to be executed computational resources need to be allocated. Using knowledge about the available computing resources, the coordinator allocates resources and provides information about them in a system function *resources* to be used during the execution.

Next, the CQ execution is started by a procedure *run*, e.g.:

```
run(q);
```

Since continuous queries run continually, the system needs knowledge about when to stop a CQ. By default the CQ runs until stopped explicitly by the user. Alternatively the user can specify some *stop condition*. We provide for two kinds of stop conditions: a count-based when the CQ runs until the specified number of logical windows from the input streams are processed, or time-based condition when the CQ runs during a specified time interval. The stop condition is provided when the CQ is started. For example, the following call specifies that the query should run for two hours:

```
run(q, "TIME", 120);
```

Finally, the execution of a CQ can be stopped by a *deactivation*, which might be initiated locally at the working nodes or from the coordinator. For example, if a CQ is specified to run without stop condition, it can only be stopped when the user issues an explicit command:

```
stop(q);
```

The system allows to resume the CQ execution later on by calling the *run* procedure again, perhaps with a different stop condition.

2.3 GSDM Coordinator

Figure 2.3 shows the software architecture of the coordinator. It is a special server that handles requests for CQs from the GSDM clients and manages

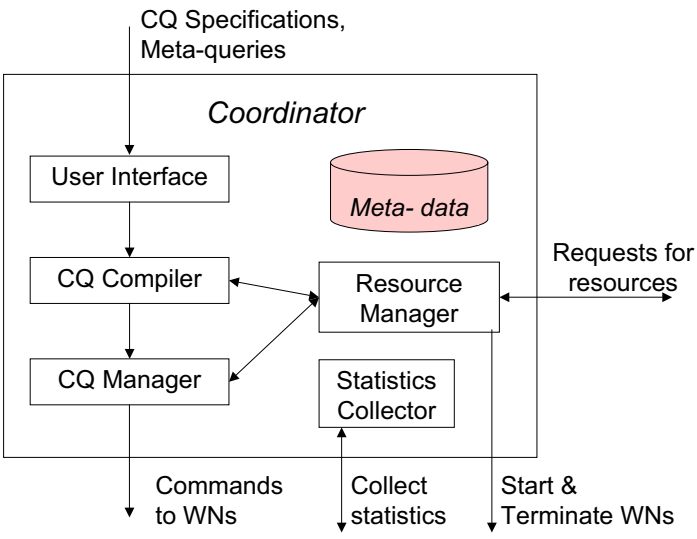


Figure 2.3: Coordinator Architecture

CQs and GSDM working nodes. The *user interface* module provides primitives to users at a GSDM client to specify, start, and stop CQs. The users can also submit meta-queries to the coordinator about, e.g., CQ performance or execution location. Given the CQ specification, the *CQ compiler* produces distributed execution plans.

The *resource manager* module is responsible for communication with the resource managers of cluster computers in order to acquire execution resources. It also manages *dynamically* the GSDM working nodes. The coordinator *starts* new working nodes when preparing the CQ execution and *terminates* them when the query is stopped. The architecture allows for starting additional working nodes when necessary during the query execution, e.g., to increase the degree of parallelism.

The *CQ manager* controls the distributed execution plans by sending commands to the GSDM working nodes. The interface between the coordinator and the working nodes includes a set of communication primitives, illustrated in Figure 2.4. Resource manager commands are illustrated in Figure 2.4 as thick dashed arrows. There are also communication primitives used by the *statistics collector* module to gather periodically statistical information from working nodes in order to analyze the CQ performance.

The coordinator stores in its local database *meta-data* about continuous queries, streams, execution plans, and working nodes. The meta-data are accessed and updated by all the modules in the coordinator.

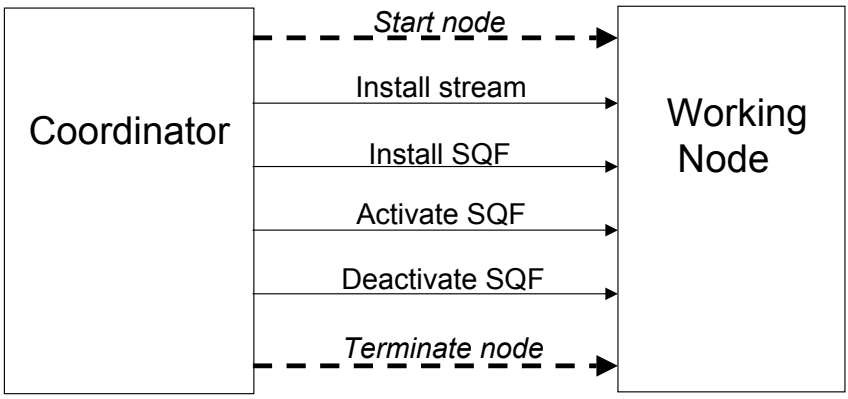


Figure 2.4: Coordinator - Working node communication primitives

2.4 GSDM Working Nodes

Figure 2.5 shows the architecture of a GSDM working node. The *CQ manager* handles the coordinator requests for initializing of execution plans.

All compiled SQFs installed on a working node are stored in a hash-table, *installed operators*. In order to start the execution of a CQ the CQ manager at the working node activates the SQFs involved in the execution plan by adding them to a list of *active operators*.

The *GSDM engine* executes continuously SQFs over the incoming stream data. It consists of four modules: a *scheduler*, *query executor*, *statistics collector*, and *buffer manager*. The scheduler assigns processing resources to different tasks. It scans the active operators and schedules them according to a chosen *scheduling policy*. It checks for incoming messages containing stream data or commands arriving on TCP or UDP sockets.

The query executor is called by the scheduler to execute an SQF one or several times depending on the scheduling policy. The executor first prepares the data from the SQF's input streams, calls the SQF, and then inserts the result windows from the execution into the SQF's result stream. The executor accesses stream data by calling methods from *stream interfaces*, which are code modules encapsulating different implementations of streams.

The statistics collector measures various parameters of CQ performance, such as processing times of SQFs, stream rates, and times spent in inter-GSDM data communication. The statistics modules are called either from the scheduler or the query executor to update internal statistical data structures. Statistics are periodically reported to the coordinator's statistics collector.

One of the GSDM design considerations was to provide for *physical data*

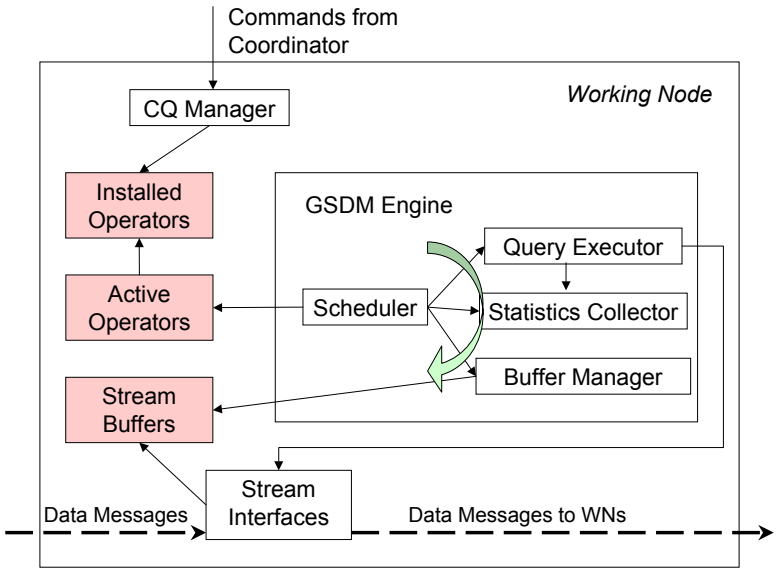


Figure 2.5: GSDM Working Node Architecture

independence to the applications (Section 1.2), which here means to enable specification and execution of CQs independent on the physical communication media of the streams. Hence, the access to stream data for each kind of stream is encapsulated in a *stream interface*. It includes the methods *open*, *next*, *insert*, and *close*. These methods have side effects on the state of the stream and are not called in SQF definitions, but by the query executor. The *next* method reads the next logical window from an input stream while *insert* emits a logical window to an output stream. The *open* method prepares the data structure or the communication used by the stream, and the *close* method cleans up when the stream will not be used any more.

We shall use the term *input stream* for a stream that is an input for some SQF. The system maintains a *buffer* for each input stream together with a *cursor* for each SQF that uses it as an input. When the *next* method reads the next logical window it also moves the cursor forward as a side effect. The system allows for sharing an input stream buffer among many SQFs by supporting an individual cursor for each of them. The *buffer manager* cleans automatically data in stream buffers no longer needed by any SQF.

Streams on different kinds of physical media are implemented by buffers, cursors, and interface methods for each kind. GSDM 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

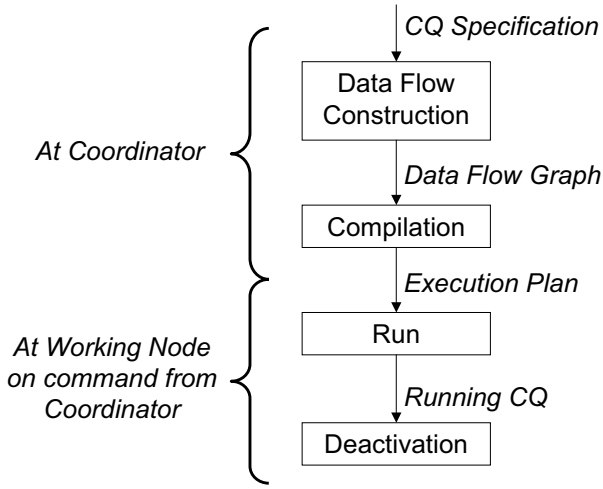


Figure 2.6: Life cycle of a CQ

programs. For the purposes of repeatable experiments, we also implemented a special *player* stream that gets its data from a file containing a recorded stream segment. GSDM can be used for continuous query processing of, e.g., multimedia data streams by providing an implementation of buffers, cursors, and interface methods for them.

Local streams in main-memory are used when SQFs connected by a producer-consumer relationship are assigned at the same execution site. *Inter-GSDM streams* provide the communication between GSDM working nodes. In order to provide loss-less and order-preserving communication they are currently implemented using TCP/IP. *External streams* provide the communication between GSDM and data sources or applications. Local and external streams are implemented as an object of type *Stream*. For each inter-GSDM stream the system creates two *dual* stream objects: an output inter-GSDM stream on the working node where the stream-producer SQF is executed, and an input inter-GSDM stream on the downstream node where one or more stream-consumer SQFs are executed.

2.5 CQ Life Cycle

After a CQ is specified by the user it goes through several phases in its life cycle as shown in Figure 2.6. This section describes the phases using an example.

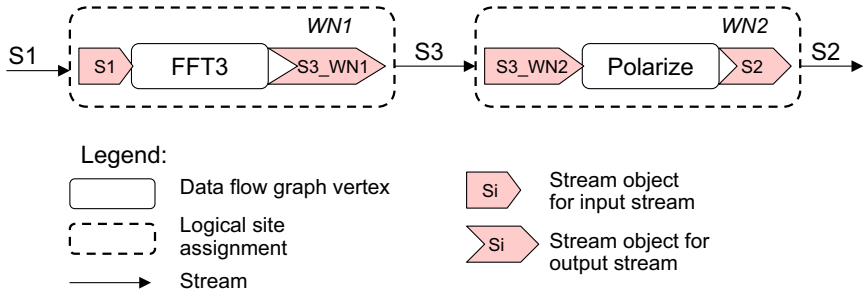


Figure 2.7: A compiled data flow graph

2.5.1 Compilation

The main purpose of the compilation is to create a description of an execution plan given a data flow graph, and input and output streams. It includes the following steps:

- Create stream objects implementing the producer-consumer relationships between SQFs. The stream objects are also assigned to logical sites determined by the site assignment of the SQFs they connect.
- Bind the SQFs to the stream objects implementing the input and result streams.

For the above example query *q* in Figure 2.2 the compilation will perform the following steps to produce the compiled graph in Figure 2.7:

- Bind the input of the first SQF, *fft3*, to the stream object representing the input stream *s1* of *q*.
- Create a pair of objects of type *stream* to implement the producer-consumer relationship between *fft3* and *polarize* SQFs. The first object, *S3_WN1*, is an output inter-GSDM stream assigned to WN1 and bound to the output of the producer SQF *fft3*. The second object, *S3_WN2*, is an input inter-GSDM stream assigned to WN2 and bound to the input of the consumer SQF *polarize*.
- Finally, the output of *polarize* will be bound to the stream object *s2* representing the output stream of the CQ.

2.5.2 Execution

The *run* procedure executes the execution plan for a CQ by performing the following steps:

1. The resource manager *maps* the logical execution sites in the plan to the

allocated resources and *starts* the GSDM working nodes. The resources are nodes of a cluster computer or some other networked computer.
2. The CQ Manager at the coordinator *installs* the execution plan on the working nodes. The plan is distributed according to the execution site assignments. If a stop condition is specified, it is also installed as part of this stage.
3. Finally, the CQ Manager *activates* the plan by adding SQFs to the active operators list and performing initialization operations, such as creating stream buffers and opening TCP connections.

Installation

The purpose of the installation is to create runnable execution plans at the working nodes, without actually starting their executions. Using the description of an execution plan, the coordinator dynamically creates and submits to the working nodes a set of commands containing installation primitives. The primitives create stream objects and data structures at the working nodes.

For the example query the following installation commands are generated at the coordinator and sent for execution to the working nodes:

```
WN1: install_stream("Radio","s1","1.2.3.4",
                    "WN1","RadioUDP");
     install_SQF("Q1","fft3",{"s1"},{});
     install_stream("Radio","s3_WN1","Q1",
                    "WN2","TCP");
WN2: install_stream("Radio","s3_WN2","WN1",
                    "WN2","TCP");
     install_SQF("Q2","polarize",{"s3_WN2"},{});
     install_stream("Polarized","s2","Q2",
                    "1.2.3.5","Visualize");
```

The installation on different nodes is independent of each other. Locally at each node it follows the order of input streams, SQF, and result stream for each SQF, since the implementation of the installation primitives requires the installation of the input streams before the installation of the SQF that process them.

Activation

The purpose of a CQ *activation* is to start its execution. The activation of a CQ is conducted by activation of all SQFs in its execution plan. The activation of an SQF includes the following steps:

- The SQF is prepared by opening its input and result streams and creating the data structures it uses.
- The SQF is added to the list of *active operators*, which are tasks scheduled by the GSDM scheduler.

Since each SQF pushes its result stream to its downstream consumers, the consumers of a stream need to be activated before its producer, so that the consumers are listening to the incoming data messages when the producers are activated. Thus, correct operation is provided by activating the data flow graph in a reverse stream flow order, starting from the SQF(s) producing the result stream(s) of the query and moving upstream to the SQFs operating on the source streams.

Again, the coordinator creates and submits to the working nodes a set of commands containing activation primitives. For the example query the activation is performed in the following order:

1. `WN2:activate("Q2");`
2. `WN1:activate("Q1");`

When all the SQFs in the execution plan are activated, the CQ execution starts. The execution at each working node is scheduled by the GSDM *scheduler*. It executes a loop in which it scans the *active operator* list and schedules tasks executing SQFs from the list. When an SQF is scheduled it executes on the windows at its current cursor positions in its input streams and produces logical windows in the result stream. The computed result windows are inserted into the result stream and the cursors of the input stream buffers are moved forward by the system. By scheduling SQFs execution in a loop the GSDM engine achieves *continuous execution of SQFs* over the new incoming data in the input streams.

For the example query the following SQF calls are scheduled and executed:

WN1: `fft3(s1);`

WN2: `polarize(s3_WN2);`

where *s1* and *s3_WN2* denote the stream object with names "s1" and "s3_WN2", respectively.

2.5.3 Deactivation

The deactivation of an SQF, which is an inverse of the activation, includes deleting the SQF from the *active operators* list and performing clean-up operations, such as closing the input and result streams [3] and releasing memory.

The deactivation might be initiated either locally at the working node or from the coordinator. If a CQ is specified to run without stop condition, the coordinator initiates the deactivation on command from the user. If the CQ has an associated stop condition, the schedulers at the working nodes check it and issue a deactivation command when the condition evaluates to true.

[3] If an input stream is used by other SQFs, it is not actually closed, but instead only the buffer cursor for the deactivated SQF is deleted.

3. An Object-Relational Stream Data Model and Query Language

This chapter presents stream data modeling and specification of continuous queries on streams in GSDM. Modeling of stream data is based on an object-relational data model where both stream sources and data items are represented by objects. Continuous queries are specified as distributed compositions of *stream query functions (SQFs)*, which are constructed through *data flow distribution templates*. The concepts of SQFs and templates were introduced in chapter 2. This chapter describes how SQFs are specified and data flow graphs constructed through a library of template constructors.

3.1 Amos II Data Model and Query Language

The GSDM prototype leverages upon the data model, query language and query execution engine of Amos II [67, 68]. The kernel of Amos II is an object-relational extensible database system designed for high performance in main memory. Next, we will introduce the main concepts of the Amos II data model and query language which are utilized in GSDM for the purposes of stream modeling and querying.

The Amos II data model is an object-oriented extension of the Daplex [76] functional data model. It is based on three main concepts: *objects*, *types*, and *functions*. Objects model all entities in the database. Objects can be self-described *literals* which do not have explicit object identifiers (OIDs), or *surrogates* that are associated with OIDs. Literal objects can be collections of other objects. The system supported collections are *bags* (unordered sets allowing duplicates) and *vectors* (order-preserving collections).

Each object is an instance of one or several types. Types are organized in a super type/subtype hierarchy supporting multiple inheritance. The set of all instances of a type forms its *extent*. When an object is an instance of a type it is also an instance of all the super types of that type. The extent of a subtype is a subset of the extent of its super types. A *type set* of an object is the set of all types that the object is an instance of. One of the types, called *most specific type*, is the type specified when the object is created.

Functions model object attributes, methods, and relationships between ob-

jects. A function consists of a *signature* and an *implementation*. Function signatures define the types of the arguments and the result. The implementation specifies how to compute the result of a function given a tuple of argument values. Depending on the implementation functions are classified into four groups. *Stored functions* represent attributes of objects. The extent of stored functions, i.e. the set of tuples mapping function's arguments and results, is stored in the database. They correspond to attributes in object-oriented databases and tables in relational databases. *Derived functions* are defined in terms of queries over other functions. They correspond to views in relational databases and methods without side effects in object-oriented databases. *Foreign functions* are implemented in some other programming language and correspond to methods in object-oriented databases. They provide interfaces for wrapping external systems and storage structures. *Database procedures* are functions defined using a procedural sublanguage. They correspond to methods with side effects and constructors in object-oriented databases. Functions can be overloaded, i.e. a function can have different implementations depending on the types of its arguments.

The query language of Amos II, AmosQL, is developed from the functional query languages OSQL and Daplex. AmosQL has nested sub-queries, aggregation operators and is relationally complete. General queries are formulated through the *select* statement as in SQL:

```
select <result>
from <type extents>
where <condition>
```

3.2 Stream Data Model

We extended the object-relational data model of Amos II with two system types: *Stream* represents stream sources, and *Window* represents stream data items called *logical windows* (Fig. 3.1)[1]. The *name* function defined on type *Stream* identifies the stream source, and *source* and *dest* functions specify stream source and destination addresses, respectively. The *interface* function specifies the stream interface implementation introduced in Section 2.4.

Figure 3.1 illustrates that logical windows are represented as instances of subtypes of type *Window*. Stream sources with different types of logical windows are represented as subtypes of the type *Stream*. The streams in the example application [55] are radio signals produced by digital space receivers represented by type *Radio*. The instrument produces three signal channels,

[1]We use the term *logical window* for a single data item to distinguish from the term window commonly used in other DSMSs for sequence of data items.

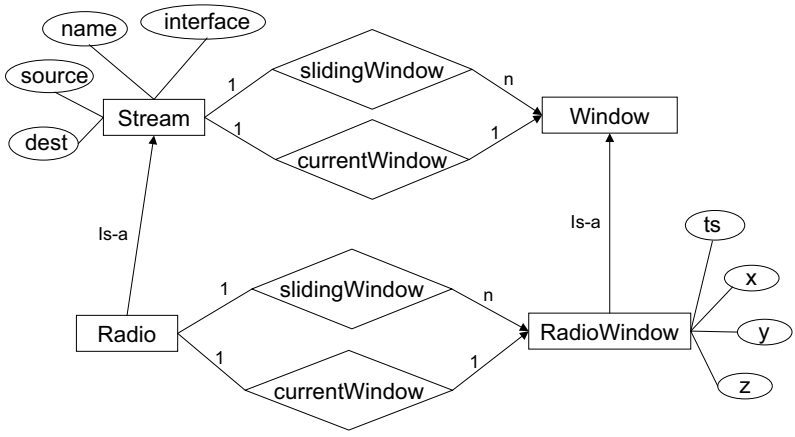


Figure 3.1: Meta-data of Radio Signal Stream Source

one for each space dimension, and a time stamp. Thus each logical window of type *RadioWindow* has the functions *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.

3.2.1 Window Functions

As we described in Chapter 1, continuous queries need to be executed in an incremental non-blocking manner. One of the common means to provide this is *windowing*, i.e. the stream operations execute on sub-streams called windows. We provide this feature through a library of *window functions* that given a stream return one or more logical windows from the stream relative to the current cursor position. We have the following built-in window functions:

- The generic function
  `currentWindow(Stream s)-> Window w`
  returns the current logical window *w* at the cursor of an input stream *s*.
- Count-based windows on streams [36] are implemented by the function
  ```
  slidingWindow(Stream s, Integer sz,
     Integer st)-> Vector of Window w
  ```
  It returns a vector of *sz* next logical windows in a stream *s*. The parameter *st* is the sliding step.
- A stream can either be timestamped or not. Timestamped streams have a distinguished attribute storing the time. The value of the time attribute can be either explicitly set in the logical window (*application time* in terminology of [79]) or obtained by calling a system function to get, e.g., the arrival time of the window to the current GSDM server (*system time* according to

[79]). Time-based windows on timestamped streams are implemented by the function

```
timeWindow(Stream s, Real span)->
   Vector of Window w
```

It combines all logical windows with timestamps in the interval $[ts-span, ts]$ defined by the time span *span* and the timestamp *ts* of the logical window at the current cursor position in the stream *s*.

The window functions are overloaded for each user stream subtype and generated automatically by the system when a new stream type is created.

As we described in Section 2.4 the access to data in the stream buffers is performed through stream interface methods, which have side effects on the state of the stream. However, in order to allow for non-procedural specification of SQFs, we need to ensure *referential transparency* of functions. The concept of referential transparency means that whenever the function is called with the same argument it gives the same result [40]. In our case this means that all functions used in SQF definitions, including the window functions, should give the same result during the execution of the current SQF on the current state of the input streams. Therefore, the stream interface methods cannot be directly called in a declarative SQF, but instead the side-effect-free window functions are used that access stream data relative to the cursor positions for the currently executed SQF.

The system maintains for each pair of SQF and input stream a local buffer of references to current logical windows in the stream. The window functions operate on these local buffers and thus do not have side effects. Hence, whenever a window function is used in a declarative SQF it returns the same data during the current SQF execution. Since the stream state changes in between consecutive executions of SQFs, what is provided is termed *local referential transparency* within each SQF execution.

The query executor module manages the contents of the local buffers of input streams by calls to the stream interface methods that actually move the stream cursors. It also inserts result logical windows from the SQF execution into the output stream.

3.2.2 Stream Types

The system is extensible with new stream data sources with new types of logical windows and operations over them. All types and functions modeling a new type of stream source are created by a system procedure, *create_stream_type*, with the following signature:

```
create_stream_type(Charstring tpname,
   Vector of Charstring attrnames,
   Vector of Charstring typenames) -> Type tp;
```

where *tpname* is the name of the new type of stream, *attrnames* is a vector of attribute names, and *typenames* is a vector of attribute type names. For example, the types and functions in the application specific part of Figure 3.1 are generated by the following call:

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

Given this information, the system procedure creates two types: one for the stream source *Radio* and one for the logical windows of this stream type *RadioWindow* as follows:

```
create type Radio under Stream;
create type RadioWindow under Window;
```

The system procedure also creates three groups of functions: *constructors* for logical windows of type *RadioWindow*, *attribute functions* extracting attributes *ts, x, y*, and *z* from the logical windows, and overloads the built-in window functions *currentWindow*, *slidingWindow*, etc. over the new types *Radio* and *RadioWindow*. For example, the generated constructor for type *RadioWindow* has the signature:

```
radioWindow(timeval ts,
   vector of complex x,
   vector of complex y,
   vector of complex z) -> RadioWindow
```

The attribute functions for type *RadioWindow* have the signatures:

```
ts(RadioWindow v) -> Timeval ts
x(RadioWindow v) -> Vector of Complex x
y(RadioWindow v) -> Vector of Complex y
z(RadioWindow v) -> Vector of Complex z
```

The only requirement in order to introduce a new stream type is that data types of the attributes must be previously defined in the system. Due to the extensibility of the object-relational data model new base types can also be introduced for representation of application data. For instance, we extended the base type system with the type *Complex* in order to represent signal samples that are complex numbers.

3.2.3 Registering Stream Interfaces

Stream interfaces encapsulate access to stream data on different communication and storage media providing a unified set of methods to the rest of the system. A new stream type can use some of the generic stream interfaces already available in the system. For example, the stream interfaces for inter-GSDM streams, local streams, and standard output streams are generic and can be used for all new stream types without changes. New stream interfaces can also be registered for a stream type by the means of a system procedure, *register_stream_interface*, with the following signature:

```
register_stream_interface(Charstring tpname,
   Charstring intname,
   Charstring openfn,
   Charstring nextfn,
   Charstring insertfn,
   Charstring closefn)
   -> Boolean;
```

The first parameter *tpname* specifies the stream type for which the interface can be used and the second, *intname*, is the name of the interface. The last four parameters are names of the functions implementing the stream interface methods *open*, *next*, *insert*, and *close*. For example, the following call registers an interface named *RadioUDP* for the *Radio* stream type using UDP communication protocol:

```
register_stream_interface("Radio","RadioUDP",
   "openUDP", "nextUDP",
   "insertUDP","closeUDP");
```

When a new stream source is registered to the system, the value of its *interface* attribute is used to determine what interface implementation to be used for it. In this way different stream interfaces are supported for the same stream type and can be used by different stream objects of this type.

3.3 Query Language

As we defined in Chapter 2, continuous queries in GSDM are specified as *data flow graphs* with vertices that are *SQFs*, and arcs that are *producer-consumer relationships* between SQFs. In this section we present the specification of SQFs.

3.3.1 Defining Stream Query Functions

An SQF is defined on one or more input streams and can also have other, non-stream parameters. There are two kinds of SQFs: *transforming SQFs* that transform a single input stream, and *combining SQFs* combining multiple input streams.

The transforming SQFs have as a first parameter the input stream on which they operate. The specification contains calls to *window functions* to extract current logical windows from the input stream and *constructors* to create new logical windows. All *attribute functions* returning components of logical windows can be used in the definition. Furthermore, the SQF specification may contain any built-in or *user-defined* operations on attributes or entire logical windows. User-defined operations are implemented as foreign functions in, e.g., C or Java, and plugged into the system. The select clause must return logical windows of the result stream type.

The SQF *fft3* below is defined on *Radio* stream type and computes the Fast Fourier Transform (FFT) on each of the three channels of the current logical window of the radio stream:

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

The `from` clause defines *v* to be of type *RadioWindow*. The `where` clause binds *v* to the current logical window of the stream `s` returned by the *currentWindow* function. The *fft* function is a foreign function that computes the FFT over a vector of complex numbers[2]. The function has the following signature:

```
fft(Vector of Complex x) -> Vector of Complex y
```

The function is called for each of the signal channels and a result logical window is created by the system generated constructor *radioWindow* called in the `select` clause.

3.3.2 SQF Discussion

The SQF specifications are asymmetric in sense that they are defined on streams but produce a window, not a stream. If an SQF were to return a stream object, the insert method of the stream interface must be called in the select clause,

[2]We chose FFT as an illustration example since it is commonly used in signal processing applications and is computationally expensive.

which would lead to a side effect in SQFs and violate the referential transparency. Therefore, we leave the insertion of the result logical window(s) into the output stream to the *query executor*. The select clause can construct a single logical window, a bag of logical windows, or sequences of logical windows.

The SQFs have to be defined over streams rather than logical windows in order to allow for aggregation of multiple subsequent data items in the input streams, as it is done by *slidingWindow* or *timeWindow* functions.

3.3.3 Transforming SQFs

The transforming SQFs provide functionality for streams similar to the functionality of the unary relational algebra operators *project* and *select*. In addition, aggregation and arbitrary transformations can be performed. To demonstrate this, next we present some more examples of SQFs used in the radio signal application.

The SQF *xchannel* extracts only the time stamp and the x channel of a radio signal stream analogous to the *project* operator in the relational algebra. It calls a *channelWindow* constructor to construct a new logical window out of the two selected components from the logical window of the input stream:

```
create function xchannel(Radio s) -> ChannelWindow
as select channelWindow(ts(v),x(v))
   from RadioWindow v
   where v=currentWindow(s);
```

A simple sampling on the stream extracting randomly one logical window out of a sequence of *n* is specified by the following *sample* function, defined on the *stream* type:

```
create function sample(Stream s, Integer n)-> Window
as select w[rand(0,n-1)]
   from Vector of Window w
   where w = slidingWindow(s,n,n);
```

This function can provide functionality of *load shedding* by random dropping of stream items similar to the *drop* operator in [2]. More complex sampling algorithms implemented as functions can be plugged into the system and used by substituting the expression in the select clause.

Partitioning of streams for the purposes of parallel execution of SQFs is implemented as a set of *partitioning SQFs*, each extracting the logical windows for each partition. For example Round Robin partitioning is implemented through an SQF *RRpart* that specifies a single Round Robin partition on any stream *s* 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);
```

The function is parameterized on the order number *pno* of the partition (starting with 0) and the total number of partitions *ptot*.

Selecting a sub-stream of elements fulfilling a predicate is illustrated in the next SQF. The result stream contains only logical windows of *Radio* source that have maximum magnitude of the x channel bigger than some threshold *thr* that is a parameter of the function:

```
create function xmagn(Radio s, Real thr)
   -> RadioWindow
as select v
   from RadioWindow v
   where v = currentWindow(s) and
      max_magn(fft(x(v)))>thr;
```

The *max_magn* is a function computing the maximum magnitude of a signal applied on the x-channel of the radio stream.

The following function *polar* computes the parameters of the polarization ellipse of the *Radio* signal that characterizes the direction of the electromagnetic field. The example illustrates how the user can encapsulate several application operations on the stream data in one SQF:

```
create function polar(Radio s) -> PolarWindow
as select polarWindow(polarization(ts(v),
      fft(x(v)),fft(y(v)),fft(z(v))))
   from RadioWindow v
   where v = currentWindow(s);
```

Here *polarization* is a function computing six vectors of real numbers that characterize the polarization ellipses for a range of frequencies. The *polar* SQF returns logical windows of type *PolarWindow* that contain a timestamp and six vectors of real numbers.

Aggregations of stream data can also be expressed as SQFs that call appropriate aggregate operations over the logical windows. Let *aggregate_window* be a used-defined function that aggregates *n* RadioWindows with signature:

```
aggregate_window(Vector of RadioWindow w)
   -> RadioWindow
```

In order to aggregate the stream using jumping, i.e. non-overlapping windows of size *n*, we specify the following SQF:

```
create function agg_stream(Radio s,Integer n)
   -> RadioWindow
as select aggregate_window(slidingWindow(s,n,n));
```

The same aggregation can be performed over sliding overlapping windows of the stream by adding a parameter for the overlapping step *st*:

```
create function agg_stream_sliding(Radio s,
   Integer n, Integer st) -> RadioWindow
as select aggregate_window(slidingWindow(s,n,st));
```

3.3.4 Combining SQFs

Combining SQFs are defined over more than one input stream and perform some kind of union or join to combine the stream data. To allow definitions over any number of input streams, the first parameter of combining SQFs is of type *Vector of Stream*. For the purposes of partitioned parallel execution of SQFs we define two combining SQFs, *S-Merge* and *OS-Join*. Other functions using different combining patterns can be defined to meet the requirements of the application.

S-Merge merges a number of input streams ordered on an *ordering attribute* into a single ordered stream. The ordering attribute in our implementation is the distinguished time stamp attribute, which all timestamped streams have. *S-Merge* is analogous to the relational *union* operator and in addition it preserves the stream order.

Since *S-Merge* compares logical windows from multiple streams, it may block if a stream stales and its buffer is empty. In order to provide non-blocking behavior *S-Merge* has a *time-out* parameter. The time-out is the time the system waits for a logical window from a stream if the stream buffer is empty, before assuming that the window is lost. The function signature is as follows:

```
S-Merge(Vector of Stream s, Real timeout) -> Window
```

The *OS-Join* SQF is an analogue to the relational *join* operator on the stream ordering attribute, which in addition allows a function-parameter to be specified that combines the logical windows with the same value of the ordering attribute. The function-parameter must be *window transforming function*, or in other words it must be defined on logical windows and return logical windows. The function signature is as follows:

```
OS-Join(Vector of Stream s, Function combinewin)
   ->Window;
```

where *combinewin* has the signature:

```
combinewin(Vector of Window v) -> Window;
```

3.4 Data Flow Distribution Templates

As described in Chapter 2, distributed execution strategies for CQs are specified through a generic framework of *data flow distribution templates*.

Each template has a *constructor* creating a data flow graph where the vertices called *nodes* are annotated with SQFs and the parameters for their execution. The nodes in the graph are assigned to *sites* which denote the logical execution places for the nodes. By assigning the nodes to different sites, a template specifies a *distribution pattern*. An arc between two nodes represents a *producer-consumer* relationship between the SQFs annotating the nodes.

Data flow graphs are represented as instances of a type *Dataflow*. Hence, the template constructors are functions returning an object of type *Dataflow*. Each data flow has properties *arity* and *width* describing the number of input and output streams of the graph, respectively.

Data flow graphs are defined using the concept of producer-consumer relationships rather than streams. During the compilation of the graph the CQ compiler analyses the producer-consumer relationships between SQFs and their site assignments. Based on this the compiler creates internal GSDM streams and binds them to the corresponding SQFs. In this way the stream parameters are deduced and added automatically by the system during the compilation. Therefore, when the template constructors are called only non-stream parameters of SQFs are specified.

To allow more complex compositions of SQFs a data flow distribution template may be used in place of an SQF parameter of another template. In this case, the template constructor creates a sub-graph in the result data flow graph. The system has an built-in library of several basic templates for *central*, *partition*, *parallel*, and *pipe* data flow graphs. Using the library the users can construct more complex data flow graphs in AmosQL, as we will show for the definition of the generic template for partitioned parallelism, *PCC*.

In the following we present an overview of the built-in templates and will describe the implementation details in Chapter 5.

3.4.1 Central Execution

Executions of SQFs on one central node are specified by a template named *central*. Given an SQF, its non-stream parameters, and its number of input streams, the constructor creates a graph with a single node annotated with the specified SQF and assigned to an execution site. The constructor has the signature:

```
central(Charstring fun, Vector args,
   Integer inp) -> Dataflow d;
```

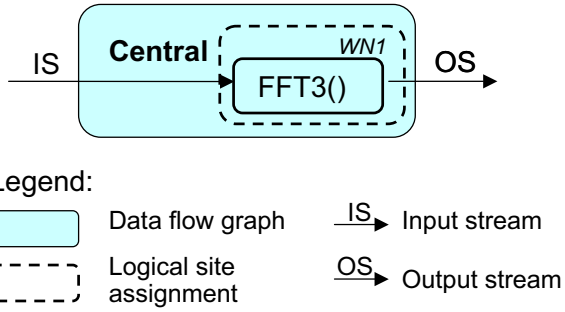


Figure 3.2: Data flow graph for cental execution

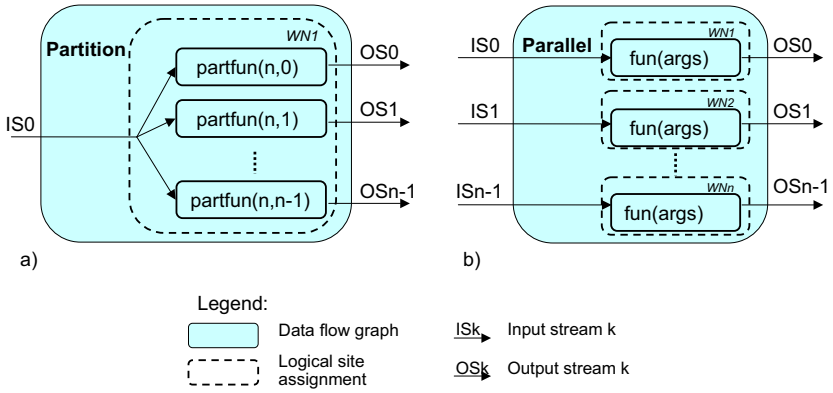


Figure 3.3: Partition data flow distribution template. All operators are assigned to a single logical site

The SQF can process more than one input stream. The *inp* parameter specifies the number of input streams. The constructor is overloaded for the common case of SQFs with one input stream and without non-stream parameters. For example, the following call generates the central data flow graph shown in Fig. 4.6 for execution of the *fft3* SQF:

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

3.4.2 Partitioning

The *partition* template creates a graph that splits a stream into *n* partitions using a user-provided partitioning function. The template has the signature:

```
partition(Integer n, Charstring partfun)
   -> Dataflow d;
```

The template creates a graph, illustrated in Figure 3.3a, with arity one and width n, where n is the total number of partitions. The graph contains n nodes and each node is annotated with the same partitioning function *partfun* but with different parameters $(n,i), i \in [0, n-1]$, where i is the order number of the partition. The nodes selecting partitions are independent of each other, share the common input stream, and are assigned to the same logical execution site.

The partitioning function that is the *partfun* parameter of the *partition* template has the following signature:

```
partfun(Stream s, Integer n, Integer pno)
   -> Window
```

where *pno* is the order number of the partition (starting with 0) and n is the total number of partitions. These parameters are automatically set by the *partition* template when the nodes of the partitioning graph are created.

If the partitioning function needs to be defined with additional parameters besides the above total number of partitions and order number, they are specified as a third parameter of type Vector, i.e. the *partfun* has the signature:

```
partfun(Stream s, Integer n, Integer pno,
   Vector params) -> Window
```

For this purpose, the *partition* template is overloaded to accept such additional parameters of the partitioning function. In this case the template has the following signature:

```
partition(Integer n, Charstring partfun,
   Vector params) -> Dataflow d;
```

3.4.3 Parallel Execution

The *parallel* template creates a graph specifying a number of parallel computations. It has the signature:

```
parallel(Integer n, Charstring fun,
   Vector params)-> Dataflow d;
```

The constructor creates a graph, illustrated in Figure 3.3b, with n input and result streams and n nodes annotated with the same function *fun* and parameters *params*. The nodes are connected to different input and result streams of the graph. The function *fun* will be executed in parallel on different sub-streams by assigning nodes to n different execution sites. There are no dependencies between the parallel nodes.

If the function needs to be executed with different parameters on different parallel branches, the order number of the branch is provided as a special parameter in *params* with value #. The *parallel* template substitutes this special

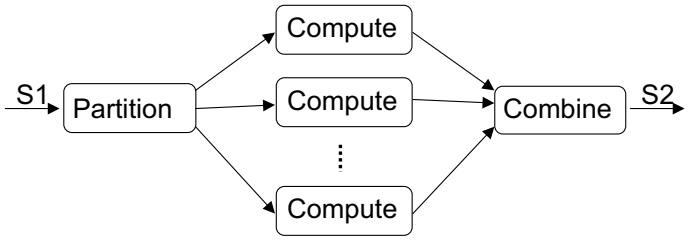


Figure 3.4: PCC: A generic data flow distribution template for partitioned parallelism

parameter with the order number of the partition when the parameters of the parallel branch are set.

3.4.4 Pipelined Execution

The *pipe* template specifies distributed pipelined execution of SQFs as we showed in Figure 2.2. The constructor takes as parameters two data flow graphs and connects them by setting producer-consumer relationships between the nodes in the graphs. More specifically, the nodes associated with the result streams of the first graph are connected to the nodes associated with the input streams in the second graph. Hence, the width attribute of the first data flow graph must be equal to the arity attribute of the second one. The signature of the constructor is:

```
pipe(Dataflow comp1, Dataflow comp2) -> Dataflow d;
```

For convenience the *pipe* constructor is overloaded to take as parameters SQFs. In that case, the central template is called first to create a central data flow graph with a node annotated with the SQF:

```
pipe(Charstring fun, Vector params, Integer inp,
     Dataflow comp2) -> Dataflow d
as select pipe(central(fun,params,inp),comp2);
```

3.4.5 Partition-Compute-Combine (PCC)

In order to provide for scalable execution of CQs containing expensive SQFs we define a generic template for customizable data partitioning parallelism. The template, called *PCC (Partition-Compute-Combine)*, specifies a lattice-shaped data flow graph pattern as shown in Figure 3.4. In the *partition* phase the input 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 signature of the PCC constructor is as follows:

```
PCC(Integer n, Charstring partfun,
    Charstring fun, Vector params,
    Charstring combfun, Vector combparams)
    -> Dataflow d;
```

The parameters specify the following properties of the three phases: i) the degree of parallelism (*n*); ii) a partitioning SQF (*partfun*); iii) an SQF to be computed in parallel (*fun*); iv) the SQF parameters (*params*); v) the combining method (*combfun*); and vi) the parameters of the combining method (*combparams*).

Using the defined above templates, we define the PCC template for partitioned parallelism as a pipe of three stages: partition, compute, and combine, as follows:

```
create function PCC(Integer n, Charstring partfun,
   Charstring fun, Vector params,
   Charstring combfun, Vector combparams)
   -> Dataflow d
as select pipe(pipe(partition(n,partfun),
      parallel(n,fun,params)),
      central(combfun,combparams,n));
```

The following PCC call constructs a graph for partitioned parallelism for the *fft3* function using Round Robin partitioning and combining the result after the parallel processing by the *S-Merge* SQF (Fig. 3.5):

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

In the next chapter we will use the PCC template to implement the main stream partitioning strategies in the thesis.

3.4.6 Compositions of Data Flow Graphs

In order to provide construction of more complex data flow graphs, template constructors can be used in place of SQF parameters in calls to other template constructors. For example, the *parallel* template (and hence PCC too) accepts as its *fun* parameter another template, as in the following call:

```
set p = PCC(2,"fft3part",
            "PCC",{2,"fft3part","fft3",{},
                   "OS-Join",{"fft3combine"}},
            "OS-Join",{"fft3combine"});
```

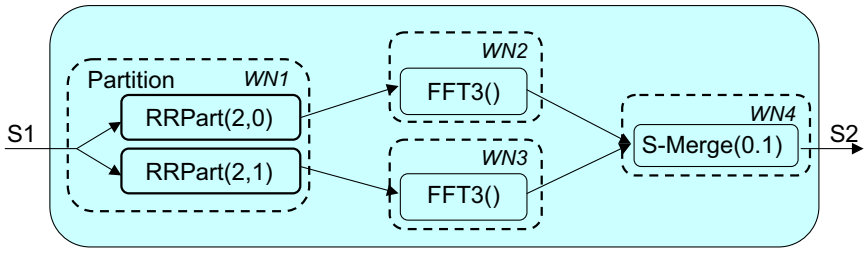


Figure 3.5: Round Robin partitioning in 2

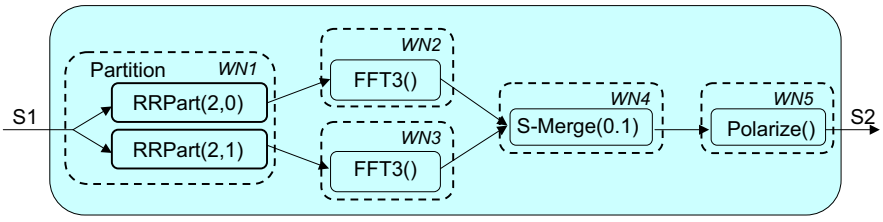


Figure 3.6: A combination of partitioned with pipelined parallelism

By specifying the PCC template in place of the *fun* parameter, the *parallel* template creates a graph of $n = 2$ parallel sub-graphs, that compose the compute phase of PCC. The above call is used to create a tree-structured distribution pattern to be shown in the next chapter (Fig 4.8). Notice, that this is different than a direct call to a template constructor that would create a single graph instead of *n* subgraphs.

Distributed execution patterns combining partitioned and pipelined parallelism can be specified using a combination of *pipe* and *PCC* templates. The next example creates the distributed execution pattern shown in Figure 3.6, where the *fft3* SQF is computed in two parallel branches, followed by the *polarize* SQF assigned to another execution site.

```
set pp = pipe(PCC(2,"RRpart",
                  "fft3",{},"S-Merge",{0.1}),
              "polarize",{});
```

4. Scalable Execution Strategies for Expensive CQ

Many classes of scientific continuous queries contain computationally expensive stream operators. Consequently, the real-time processing requirement for such CQs puts high demands for system scalability. In this chapter we address the scalability problem by investigating different strategies for partitioned parallelism for an expensive SQF. We begin with a formulation of the requirements for stream partitioning strategies and define two overall strategies: SQF dependent *window split (WS)* and SQF independent *window distribute (WD)*. The implementation of the strategies in GSDM is described and an experimental evaluation of their scalability is presented. The scalability is measured in terms of two factors: scaling the maximum throughput and scaling the size of the logical windows with respect to the SQFs. Finally, a formal analysis of system throughput is presented and compared with the experimental results.

4.1 Window Split and Window Distribute

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

1. Partitioning must preserve the semantics of the SQF.
2. Partitioning must be order preserving and non-blocking.
3. Partitioning has to provide as good as possible load balancing.

While requirements 1) and 3) are generally valid for any data partitioning strategy, the second requirement is specific for stream partitioning and follows from the general requirements for stream processing as described in Chapter 1.

We define two overall stream data partitioning strategies, *window split* and *window distribute* that fulfill the requirements stated above.

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 of smaller size, which allows for achieving scalable execution of expensive SQFs. Since the logical window is an object that is a single data unit from the SQF point of view, window split strategy provides *intra-function* parallelism [63] for computationally expen-

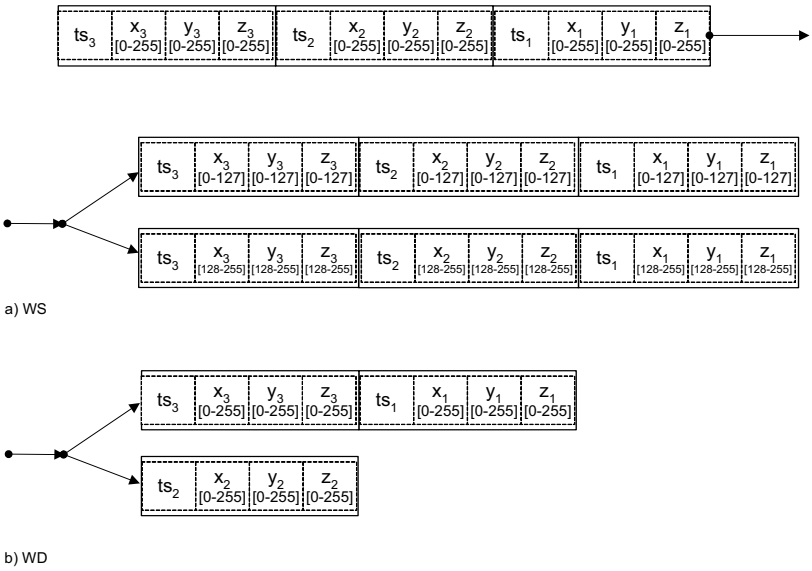


Figure 4.1: Partitioning of Radio *stream logical windows (a)Window Split; (b)Window Distribute*

sive functions (SQFs). Fig. 4.1a illustrates window split on a stream of type *Radio* into two partitions.

Window split is particularly useful when scaling the processing for big logical window sizes. For example, in the Space Physics application [55] the FFT operation needs to be applied on large vector windows. The bigger the size of the vectors the more precise is the result in the frequency domain and wider range of frequencies are covered further away from the central frequency that is a parameter of the instrument.

Since SQFs are executed on entire logical windows, the window split strategy needs to take care to preserve the SQF semantics when creating and combining sub-windows. There are no general rules applicable to all SQFs, but instead the system needs specific knowledge about application data types and SQF semantics. This knowledge is provided in GSDM by parameterizing the window split strategy with *SQF-dependent* window splitting and combining functions. This also means that the window split strategy is applicable only for SQFs for which window splitting and combining functions are provided.

How well the load balancing requirement is fulfilled by the window split strategy for a particular SQF depends on the concrete SQF-dependent window splitting function that is a parameter of the strategy.

The requirement for non-blocking and order preserving stream processing concerns the implementation of the strategy. In GSDM it is fulfilled by an

SQF, *operator dependent stream join (OS-Join)* that implements the combine phase of window split.

Window distribute distributes several logical windows among different partitions. The routing of windows can be based on any well-known partitioning strategy, such as Round Robin and hash partitioning [64], or can utilize some user-defined partitioning function. The concrete routing strategy is a parameter of window distribute. As an example of the window distribute strategy we provide a *Round Robin stream partitioning* (RR) strategy where logical windows in their entirety are distributed among partitions based on their ordering attribute. Fig. 4.1b illustrates window distribute in two partitions by RR applied on a *Radio* stream.

Since window distribute dispatches entire logical windows, it does not affect the semantics of the parallelized SQF. Therefore, in contrast to window split, the partitioning and combining phases of window distribute strategy are *SQF-independent* and the strategy is applicable to any SQF.

In the combine phase of window distribute, the result sub-streams are merged on their order identifier[1]. In our implementation this is provided by a non-blocking and order preserving SQF *stream merge (S-Merge)*. Thus, the Round Robin partitioning for streams is an extension of the conventional Round Robin partitioning that provides ordering in the result stream after the parallel execution.

The load balancing of the window distribute depends on the chosen partitioning function that is a parameter of the strategy. Our choice to use Round Robin for routing windows 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 for the class of the example signal processing applications.

4.2 Parallel Strategies Implementation in GSDM

In an object-relational database system the user can extend the system by defining new data types and operations on them. In order for the data and the query processing over them to be really incorporated into the system, it is necessary to be able to extend different modules of the system, e.g. to add access methods appropriate for the new data or user-defined aggregation operations [80]. In the same course of reasoning, the system should be extensible with parallel execution strategies applicable for the new operations.

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

We provide such extensibility in GSDM through *customizable* data flow distribution templates defined in Chapter 3.

In particular, the generic template for partitioned parallelism *PCC*, is parameterized with *partitioning* and *combining* functions, which in their turn accept parameters specifying a particular partitioning strategy. This allows the system to be extended in a generic way for new applications not only with new data types and operators, but also with data partitioning strategies appropriate for the user-defined operators.

Both window split and window distribute strategies are implemented using the *PCC* template with corresponding parameters: the partitioning and combining methods. The partitioning phase implemented by the *partition* template takes as a parameter a partitioning function with the following signature:

```
partfun(Stream s, Integer ptot, Integer pno)
   -> Window
```

The function returns the next logical window of a stream *s* for the partition with order number *pno* given the total number of partitions *ptot*.

The combining function is defined on a vector of streams and eventually other parameters and returns one or more logical windows of the combined result stream:

```
combfun(Vector of Stream s, ...) -> Window
```

We will illustrate the parallel strategies implementation using as an example the SQF *fft3* defined in the previous chapter.

4.2.1 Window Split Implementation

In order to use the window split strategy for a particular SQF the corresponding SQF-dependent partitioning and combining functions need to be provided. The partitioning function specifies how a sub-window is extracted from the original logical window while preserving the SQF semantics. In the combining phase the sub-windows belonging to the same original logical window are selected and combined according to the SQF semantics.

For the example SQF *fft3* we have implemented a partitioning SQF *fft3part*. It transforms the current window of the *Radio* stream into a sub-window by extracting those parts of the vector components that correspond to the partition number *pno*:

```
create function fft3part(Radio s,
   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))
   from RadioWindow w
   where w=currentWindow(s);
```

fftpart is a foreign function that partitions a vector according to the *FFT-Radix K* [50] algorithm where *K* is a power of 2. The *fftpart* function signature is as follows ($K = ptot$, i.e. the total number of partitions):

```
fftpart(Vector of Complex v, Integer ptot,
   Integer pno) -> Vector of Complex;
```

When $K = 2$ the *FFT-Radix 2* 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. Therefore, *fftpart(v,2,0)* returns a vector of those v elements that are in the even index positions, while *fftpart(v,2,1)* returns a vector of the elements in the odd index positions.

The combine phase of window split is implemented by an SQF, *operator dependent stream join (OS-Join)*. It selects sub-windows, one from each parallel SQF computation that match on the ordering attribute, and combines them into one logical result window by applying a *window transformation* function. The window transformation function specifies how logical result windows are computed from sub-windows while preserving the SQF semantics. Hence, it is an SQF-dependent parameter of *OS-Join*. The functions have the following signatures:

```
OS-Join(Vector of Stream s, Function combinewin)
        ->Window;
combinewin(Vector of Window v) -> Window;
```

OS-Join is a form of equi-join on the ordering attribute of sub-streams, which in addition combines the matching sub-windows and preserves the order of the result windows by processing sub-windows according to the ordering attribute.

For the example SQF *fft3* we defined an FFT-specific window transformation function, *fft3combine*, with the signature:

```
fft3combine(Vector of RadioWindow v) -> RadioWindow;
```

It implements the last step of the FFT-Radix K algorithm in order to compute vector components of the result logical window from sub-vectors in the FFT-computed sub-windows.

The *window split* strategy for the SQF *fft3* is implemented by using *fft3part*, *OS-Join*, and *fft3combine* functions as arguments of the *PCC* constructor. The

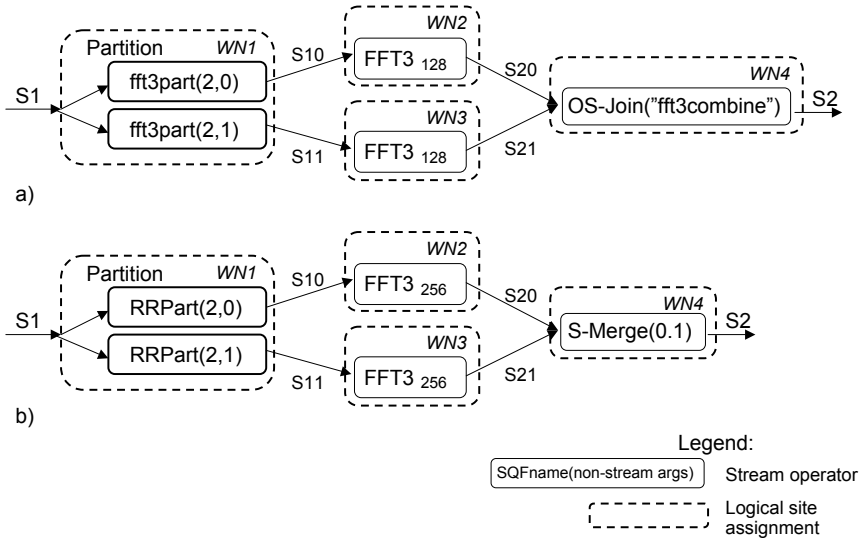


Figure 4.2: Parallel strategies with partitioning in 2. (a)Window Distribute with Round Robin; (b)Window Split with fft3part and fft3combine

following call creates the parallel data flow graph in Figure 4.2a with degree of parallelism two:

```
PCC(2,"fft3part","fft3",{},
    "OS-Join",{"fft3combine"});
```

The subscripts in Fig. 4.2 denote vector sizes in the logical windows processed by *fft3* for input logical windows of size 256.

The SQFs in the graph created from the PCC constructor are connected by producer-consumer relationships. When the graph is compiled into an execution plan, these relationships are implemented by internal data streams between the SQFs, denoted in the figure by $s10$, $s11$, etc. When the execution plan is installed on the GSDM working nodes, the query executors will execute the following actual calls to SQFs:

WN1: `fft3part(s1,2,0);`
`fft3part(s1,2,1);`
WN2: `fft3(s10);`
WN3: `fft3(s11);`
WN4: `OS-Join({s20,s21},`
`functionnamed("fft3combine"));`

4.2.2 Window Distribute Implementation

In order to implement window distribute by Round Robin we defined a partitioning function *RRpart*. It 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);
```

The combining SQF, *S-Merge*, has the signature:

```
S-Merge(Vector of Stream s, Real timeout) -> Window
```

combines the result sub-streams after the compute phase preserving the order defined by the order identifiers, timestamps in our implementation. *S-Merge* assumes that the sub-streams are ordered on the 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 continuous 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 logical window to arrive in a stream with empty buffer before assuming that data in the stream has been lost or delayed, in which case the processing continues with the remaining streams. Other policies, such as replacement or approximation of missing windows are also possible.

The following call to the *PCC* constructor creates the parallel data flow graph in Figure 4.2b using window distribute partitioning by Round Robin in two partitions:

```
PCC(2,"RRpart","fft3",{},"S-Merge",0.1);
```

When the graph is compiled and installed on the working nodes, the actual calls to the SQFs are as follows:

WN1: `RRPart(s1,2,0);`
`RRPart(s1,2,1);`
WN2: `fft3(s10);`
WN3: `fft3(s11);`
WN4: `S-Merge({s20,s21},0.1));`

4.3 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.

4.3.1 Performance Metrics

The main purpose of the strategies for partitioned parallelism is to provide scalable execution of expensive functions on high-volume streams. The scalability is desirable with respect to two factors: high total data volume and big size of logical windows. The second factor originates in a requirement for scientific computations to scale with the increase of the logical window size, e.g., in order to improve the precision of the results. Therefore, we measured the scalability of the parallel strategies in terms of two criteria: total maximum throughput and size of the logical windows with respect to the SQFs.

The scalability measured with respect to the logical window sizes corresponds to the *batch scalability* since it investigates how the system scales with the increase of the size of the processing tasks. Since each arrival of a stream data item triggers a chain of transactions, the scalability measured when the input stream rate increases can be thought of as *transactional scalability* [29].

We measured the throughput by the time to process a stream segment of fixed size containing 2MB signal samples for each of the three channels. The amount of communicated data was approximately 50MB using binary encoding for the vectors of complex numbers. The size of the stream segment 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 became under a threshold of 3%. All the diagrams presenting execution times show the times for such maximum throughput.

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.

We observe that metrics for parallel stream processing are somewhat different than for other parallel algorithms. Traditionally parallel algorithms are designed and evaluated assuming some initial distribution of data among the processors and distribution of the results after the processing is completed [50]. Therefore, the input data distribution and combining of the results are often not considered when the performance of parallel algorithms is measured. In a parallel stream system the data comes continuously from stream sources and has to be distributed, processed, and combined in near real time. There-

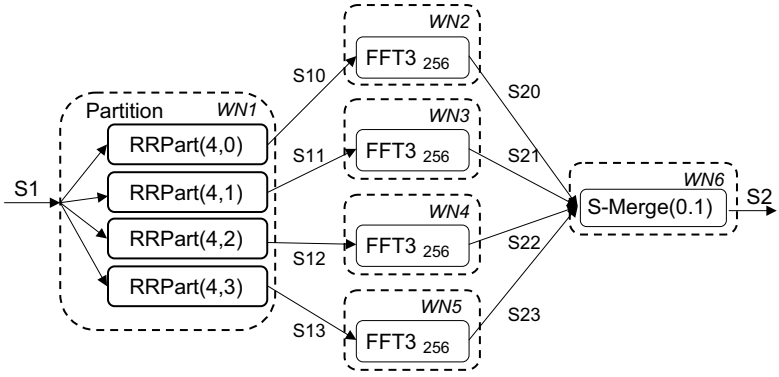


Figure 4.3: Window Distribute with Round Robin: flat partitioning in four.

fore, partitioning and combining phases in the partitioned parallel stream processing must be taken into account when measuring the performance.

4.3.2 FFT Experiments

The experimental set-up for the example SQF *fft3* included three main strategies. The parallel strategies are window split using FFT-dependent parameters for split and combine functions (Figure 4.2a) and window distribute using Round Robin (Figure 4.2b)[2]. The central execution on a single node is a reference strategy used as a basis to measure the speed-up of the parallel strategies. In all parallel strategies the measurements include the partitioning and combining phases.

The parallel strategies WD and WS were tested for degree of parallelism 2, 4 and 8. Figures 4.3 and 4.4 show the data flow graphs for degree of parallelism four.

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 the 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.

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 system performance, we ran

[2]We will use the shorter names WS and WD for window split and window distribute, respectively.

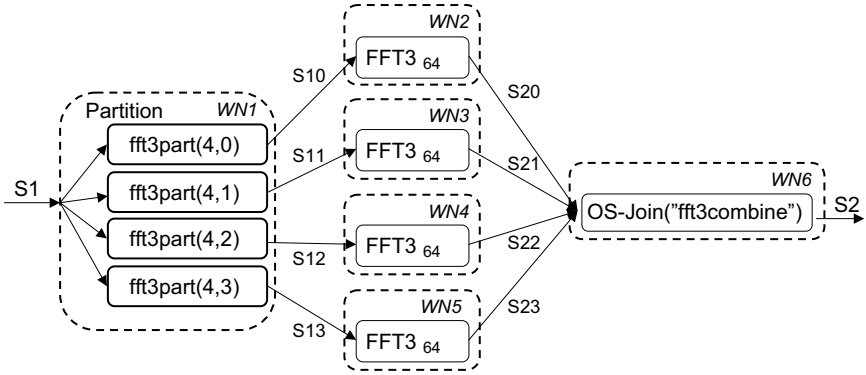


Figure 4.4: Window Split with fft3part and fft3combine: flat partitioning in four.

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 4.5 shows the execution times of FFT implementations for logical windows of different sizes.

Figure 4.6 illustrates the increase of the total elapsed time with the increase of the logical window size for the central strategy and both parallel strategies with degree of parallelism two, and in both fast and slow experimental sets. For all the strategies the FFT processing nodes are most loaded and therefore the total maximum throughput is determined by the FFT operation complexity, $O(n \log n)$. The WS2 strategy is faster than the WD2 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 WD2 strategy. Given the complexity of FFT this results in less total time in the dominating compute phase. We will present a formal analysis of this property at the end of this Section.

We observed an exception of WS2 performance for the smallest window size of 256 in the fast experimental set. The strategy has bigger total overhead than WD2 strategy due to memory management and communication of logical windows since the amount of windows it processes is bigger by a factor of two compared to the WD2 strategy, while their size is smaller by a factor of two.

Figure 4.7 illustrates the results from the slow experimental set for a degree of parallelism four. Here we compare three strategies: WD4 and WS4 with *flat* partitioning, i.e. in a single node, and WS4-Tree with *tree* partitioning.

Figure 4.8 illustrates the tree partitioning strategy, 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

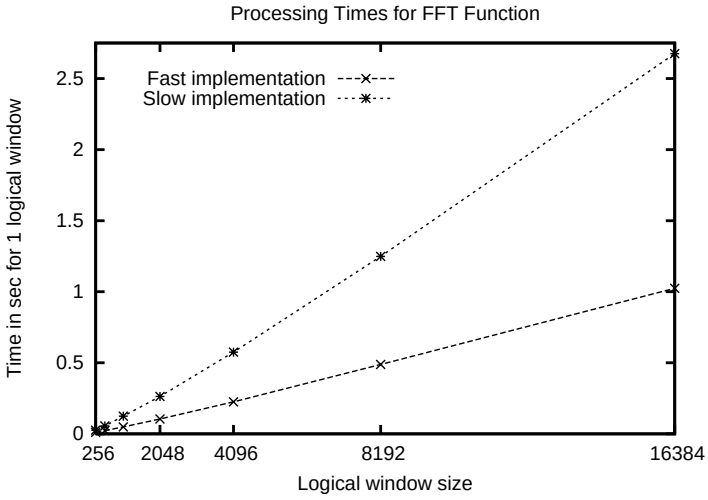


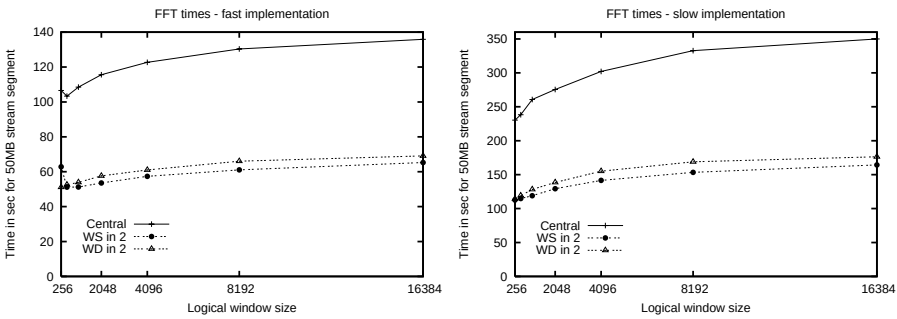
Figure 4.5: Times for FFT implementations

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 parallelism in cases when the cost in these phases limits the flow.

The load of FFT processing nodes in WD4 strategy dominates the load of the partition and combine nodes and determines the maximum throughput for all logical window sizes. The load balancing between different phases of the distributed data flow is illustrated by the diagram in Figure 4.9 which shows the total elapsed time spent in communication, computation, and system tasks.

The load of FFT processing nodes also dominates in WS-Flat strategy for logical windows of size 1024 and bigger. Hence, the processing time curve for these sizes again follows the FFT complexity behavior. However, for window sizes smaller than 1024 we observe an increase of the total execution time. This performance decrease is caused by the fact that the WS4-Flat combining node becomes most loaded (Fig. 4.9). We found two factors contributing for the high load at the combining node: first, it performs user-defined combining of windows, and second, there is a high overhead related with window maintenance. Next, we analyze these factors.

As it can be seen on the diagram (Fig. 4.9) both the partitioning and combining nodes of WS4-Flat strategy are much more loaded than the correspondent nodes in WD4. The WS4 strategies have in general more expensive user-defined splitting and combining SQFs. For example, the *fft3part* function copies vector elements in order to create partitioned logical windows and the *OS-Join* function computes result windows using *fft3combine* that executes the last step of the FFT-Radix K algorithm. The computations involve one mul-



(a) *Fast implementation.* **(b)** *Slow implementation.*

Figure 4.6: FFT times for central and parallel in 2 execution

tiplication and one sum of complex numbers for each element of the vector components of the result window.

The second source of higher load of WS nodes is that they manage bigger number of logical windows with smaller sizes compared to WD strategy. Therefore, the total overhead due to the management of logical windows is bigger for WS4-Flat than for WD4 strategy.

To summarize, higher computational cost and window maintenance overhead cause bigger load of WS4-Flat partitioning and combining nodes compared to the load of corresponding WD4 nodes. For window sizes smaller than 1024 the load of the combine node dominates the load of the FFT-computing nodes and limits the throughput. Even though the compute phase of WS4-Flat is more efficient than the compute phase of WD4, the system cannot benefit from this, because of the combine phase limitation. As a result, for those logical window sizes the WD4 strategy shows higher total throughput than WS4-Flat strategy.

The the WS4-Tree strategy overcomes the problem with the dominating load of the combine node for size 512 by distributing the load of the partition and combine phases into tree-structures. The overhead is smaller for WS4-Tree strategy than for WS4-Flat strategy since its outermost partition and combine nodes manage smaller number of logical windows with size bigger by a factor of two. Hence, WS4-Tree is the best strategy for size 512 at the price of bigger number of processing nodes. However, for the smallest size of 256 the strategy experiences the same problem as WS4-Flat and its throughput becomes below the throughput of window distribute strategy.

Figure 4.10 illustrates the results from the experimental set with fast FFT implementation for degree of parallelism four. Here we compare four strategies: WD4 and WS4 with *flat* partitioning, and WD4-Tree and WS4-Tree with *tree* partitioning.

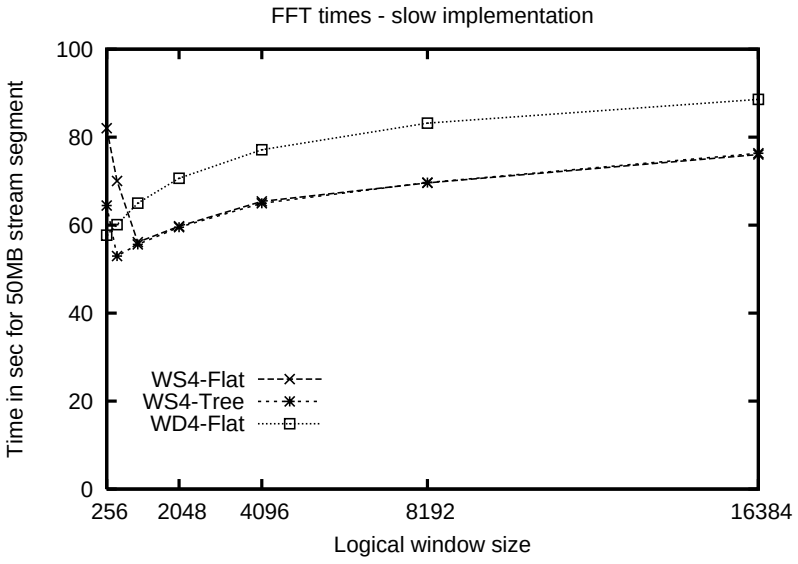


Figure 4.7: FFT times for parallel in four execution. Slow FFT implementation

The crossing point after which the WS4-Flat strategy becomes preferable over the WD4 is shifted here to the logical windows of size 8192. Again, the load of the combine phase of WS4-Flat is high and prevails the FFT-processing load for all logical window sizes (Fig. 4.11).

Which of the strategies has higher throughput in this situation depends on the proportion between the most loaded compute nodes in WD4 strategies and the dominating combine node in WS4-Flat strategy. Figure 4.11b illustrates that WS4-Flat strategy becomes preferable for logical windows of size bigger than or equal to 8192, while WD4 strategy has higher throughput for smaller window sizes, e.g. as shown for the size 2048 in Fig. 4.11a.

Similarly to the slow experimental set, the WS4-Tree strategy has less loaded distributed combine nodes than the central WS4-Flat combine node. For all window sizes bigger than 512 the throughput is limited by the FFT-computing nodes and the strategy is best for those sizes since the total load of its FFT-computing nodes is smaller than the dominating FFT-computing load in WD4. However, this best performance strategy utilizes more resources than the strategies with flat partitioning. We also observe that the improvement gained by the user-defined partitioning with respect to the RR partitioning is smaller for the fast FFT implementation than for the slow implementation, correspondingly to the FFT costs.

Both parallel strategies provide good load balancing between parallel branches assuming a homogeneous cluster environment where parallel nodes have

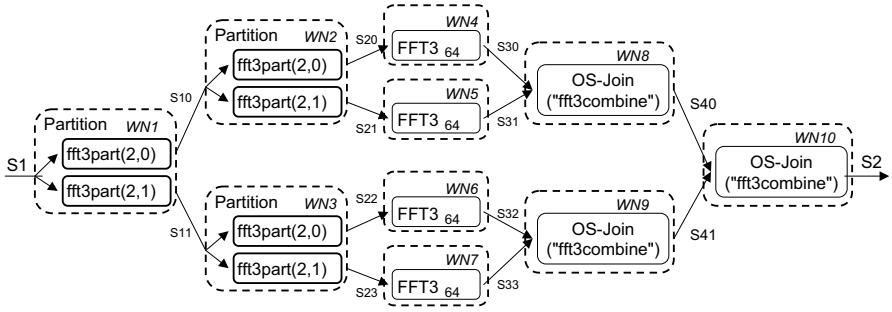


Figure 4.8: Window Distribute with tree partitioning in four

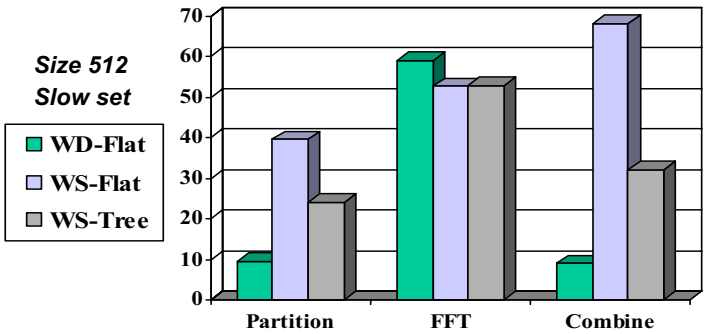


Figure 4.9: Times for window size 512, slow FFT implementation

equal capacity. Window distribute achieves this by Round Robin, while window split utilizes a user-defined splitting of a window into sub-windows of the same size. However, the experiments show that in order to achieve maximum throughput it is not sufficient to provide efficient and well balanced compute phase, but it is also necessary to achieve good load balancing between the partition, combine, and compute phases.

Table 4.1 illustrates the proportion between elapsed processing and communication times in the partition, compute, and combine phases of both strategies. The measurements are taken for the fast experimental set, logical windows of size 8192, and degree of parallelism two. We observe that WD partitioning and combining nodes as well as WS partitioning node spent most of the time communicating data. However, a substantial amount of time in

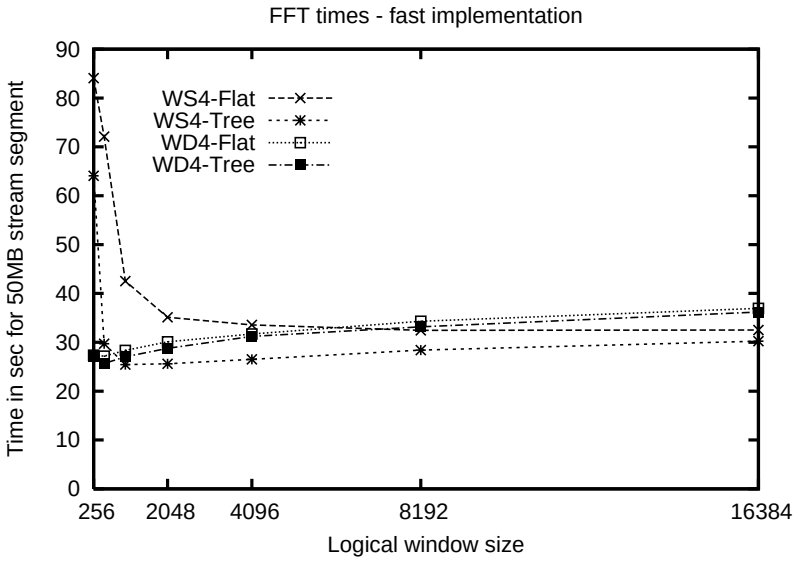
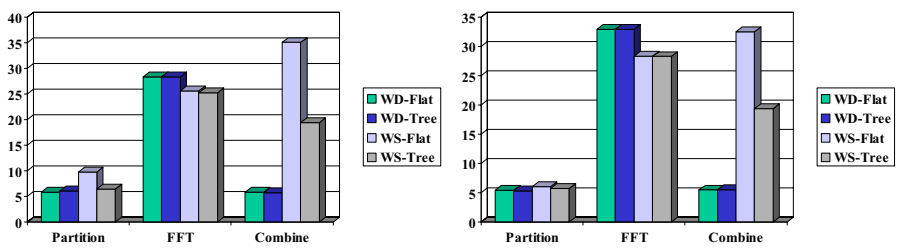


Figure 4.10: FFT times for parallel in four execution. Fast implementation

WS combining node is spent in processing of the user-defined combining of sub-streams.

The speed-up of the parallel strategies for an expensive function (slow FFT implementation) and size 8192 is presented in Figure 4.12. It clearly illustrates the cases when it is beneficial to utilize user-defined partitioning based on the SQF semantics. If the function is expensive enough (slow implementation), resources are limited (e.g., four or six nodes), and the user-defined partitioning provides more efficient dominating compute phase, the window split strategy provides better speed-up. For example, when resources are lim-



(a) *Logical window of size 2048.* **(b)** *Logical window of size 8192.*

Figure 4.11: Real time spent in partition, FFT, and combine phases of the parallel-4 strategies, fast implementation.

	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 4.1: *Communication and computational costs in different PCC phases*

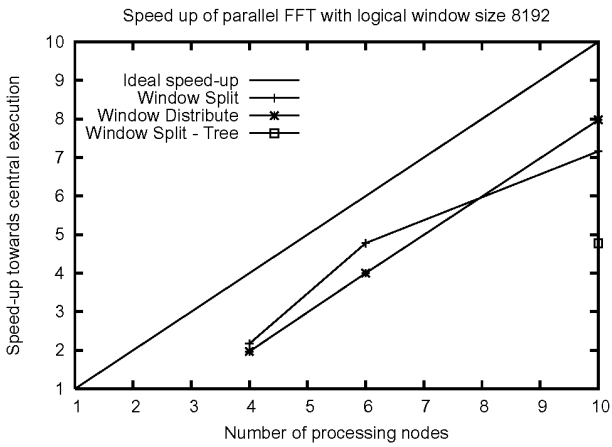


Figure 4.12: Speed up of parallel FFT strategies for window size 8192.

ited to six computational nodes, two of which are dedicated to split and join, WS4-Flat achieves a speed-up of 4.72, while WD4-Flat has a speed up of 4. For bigger number of nodes, e.g. 10 in the diagram, window distribute using RR shows better speed-up since the throughput of window split is limited by the user-defined computations and per-window overhead in the combine phase. The WS4-Tree strategy shows worst speed-up since it utilizes more resources than the flat partitioning strategies.

4.3.3 Analysis

The experiments show that the window split strategy achieves better performance in the compute phase than the window distribute strategy by utilizing an FFT-dependent partitioning. In this section we analyze the throughput of

the parallel branches of both strategies for the *fft3* SQF to complement the experimental results.

The throughput of a node can be expressed as the ratio of the size of a logical window over the total time it takes to process it.

The total processing time T for a logical window of size N includes the following components:

- Time $T_{SQF}(N)$ for executing an SQF assuming that only one SQF is executed on the node (the common case for the parallel branches);
- Communication time $T_{recv}(N)$ to receive an input logical window;
- Communication time $T_{send}(N)$ to send a result stream window;
- Time T_{sys} for performing system tasks, such as buffer management and scheduling

$$T(N) = T_{recv}(N) + T_{SQF}(N) + T_{send}(N) + T_{sys} \tag{4.1}$$

In the model above we assume that the SQF has a fan-out factor of one, i.e. produces one result window for each input logical window.

Let us consider the throughput of one partition of a strategy with degree of parallelism k and logical windows of size N. The maximum throughput of one Round Robin partition is:

$$Thr_{RR} = \frac{N}{T(N)} \tag{4.2}$$

For the same initial logical window size the window split partitioning processes k logical windows of size $\frac{N}{k}$. Hence, the maximum input throughput of one parallel window split partition is:

$$Thr_{WS} = \frac{\frac{N}{k}}{T(\frac{N}{k})} = \frac{N}{kT(\frac{N}{k})} \tag{4.3}$$

In order for the user-defined window split partitioning to have higher throughput than the RR partitioning, the following condition must be fulfilled:

$$\frac{N}{kT(\frac{N}{k})} > \frac{N}{T(N)} \tag{4.4}$$

or

$$T(N) > kT(\frac{N}{k}) \tag{4.5}$$

$$T_{recv}(N) + T_{SQF}(N) + T_{send}(N) + T_{sys} > k(T_{recv}(\frac{N}{k}) + T_{SQF}(\frac{N}{k}) + T_{send}(\frac{N}{k}) + T_{sys}) \tag{4.6}$$

Under the assumption that the system overhead and the communication

times are approximately the same, the computational complexity of the SQF determines the strategy with the best throughput. Under the above assumption the window split is better than the window distribute if and only if:

$$T_{SQF}(N) > kT_{SQF}(\frac{N}{k}) \tag{4.7}$$

The *fft3* SQF executes FFT with complexity $O(n \log n)$. If we ignore the low-order terms in the complexity formula for big enough N the above requirement takes the form:

$$cN \log N > kc\frac{N}{k} \log \frac{N}{k} = cN(\log N - \log k) \tag{4.8}$$

The last inequality is fulfilled for all values for the number of partitions $k \geq 2$.

Here we need to take into account the following factors that might cause unexpected behavior in the real experiments.

- For small values of N the low-order terms might prevail and change the inequality.
- Even when the SQF complexity analysis shows an advantage for window split, the total real advantage is smaller since communication and system tasks costs also affect the total throughput. In situations with relatively cheap SQFs and relatively expensive communication costs, such advantage might be negligible.
- As it can be seen from equation 4.7 whether user-defined partitioning would have advantage in terms of total throughput depends on the computational complexity of the SQF as well as on the existence of a particular, more efficient, method for parallel computation to be used. For example, we can expect similar results for sort algorithms with complexity $O(n \log n)$, such as merge-sort, but we cannot expect throughput improvement for SQFs with linear complexity.

5. Definition and Management of Continuous Queries

In this chapter we present the definition and management of continuous queries. First we describe the meta-data used for representation of continuous queries and data flow distribution templates at the coordinator. We present the implementation of the built-in library of templates constructors.

Long-running continuous queries require different management from stream management systems than the management of one-time queries in DBMSs. We describe the CQ management tasks performed by the coordinator during the life cycle of queries. Finally, we describe a profile-based optimization framework for automatic generation of optimized parallel plans for expensive query functions.

5.1 Meta-data for CQs

Figure 5.1 shows the meta-data describing continuous queries at the coordinator. Queries are instances of type *CQ*. Stream objects at working nodes are described at the coordinator by objects of type *Mstream* (a *meta-stream*). The *Mstream* properties *source*, *dest*, and *interface* represent the corresponding properties of the stream objects (Sec. 3.2). In addition the *stype* property stores the most specific type of a stream object, e.g., *"Radio"*, which is needed when a stream object is created at a working node. The functions *inputs* and *outputs* are defined on type *CQ* and return vectors (ordered sequences) of meta-stream objects representing the stream on which the query is executed.

The following system procedures register input and output streams to the system, respectively.

```
register_input_stream(Charstring stype,
   Charstring source, Charstring interface)
   -> Mstream s;

register_result_stream(Charstring dest,
   Charstring interface) -> Mstream s;
```

For each input stream the user specifies its type *stype*, the source address of the program or instrument sending the stream, and stream interface to be used.

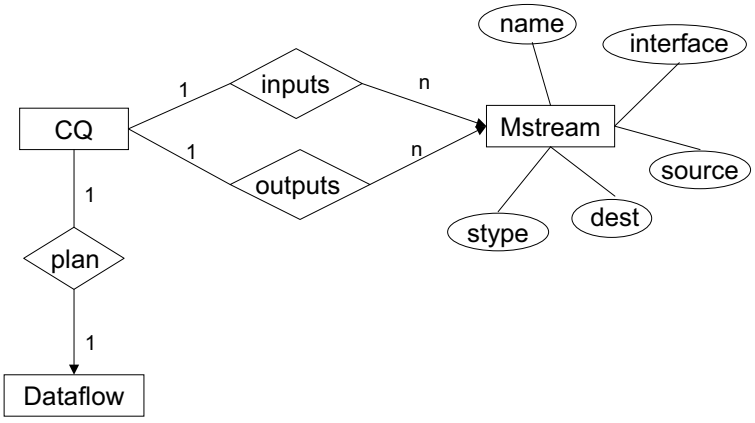


Figure 5.1: Meta-data for continuous query representation

The type of the stream must be some of the stream types defined in the system, and the interface must be valid for this stream type. The stream properties that are not specified in the above procedures, are inferred by the system. For instance, the destination of an input stream will be set to the working node where the SQFs operating on it are assigned. The name of the stream is an identifier, generated automatically when an *Mstream* object is created.

The user specifies the destination address *dest* to which the result stream should be sent and the stream interface to be used. Notice, that the type of the result stream does not need to be specified, since the system will derive it from the CQ specification. The above procedures create in the coordinator the objects of type *Mstream* representing the streams and set their attributes.

The *plan* function specifies an object of type *Dataflow* that represents the data flow graph for query execution. We describe the meta-data about the graphs in the next section.

The following system procedure is a constructor for *CQ* objects given the data flow graph, and the input and output streams:

```
cq(Dataflow d, Vector of Mstream ins,
   Vector of Mstream outs) -> Cq q;
```

An example query specification is given below:

```
set s1 = register_input_stream(
   "Radio","1.2.3.4","UDP");
set s2 = register_result_stream(
   "1.2.3.5","Visualize");
set wsplit = PCC(2,"fft3part","fft3",{},
```

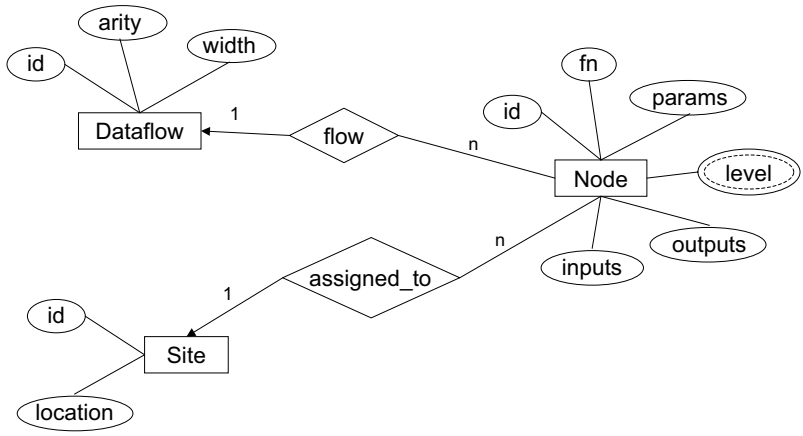


Figure 5.2: Metadata for specification of distributed data flow graphs

```
   "OS-Join",{"fft3combine"});
set q = cq(wsplit, {s1}, {s2});
```

Here the user explicitly specifies the distributed execution strategy for the query through the PCC template, providing the system with information about the degree of parallelism two and the partitioning strategy to be used, defined through the *fft3part* and *fft3combine* functions. To provide transparency to the user, the system can also automatically enumerate and select an optimized execution plan for a given query. We will address this problem at the end of the Chapter.

5.2 Data Flow Graph Definition

Figure 5.2 shows the meta-data describing data flow graphs. The graphs are represented by a type *Dataflow* with properties *arity* and *width* describing the number of input and output streams of the graph, respectively. It is assumed that the input and output streams are numbered starting from zero. The *Dataflow* type has a number of other attributes that describe, e.g., execution statistics of the graph, and are supported automatically by the system. Here we skip these attributes and present only the properties important for the data flow graph construction.

Vertices in the graph are instances of type *Node* with properties *id* that is an automatically generated unique identifier, *fn* that is the name of the SQF annotating the node, and *params* that is a vector of non-stream parameters of

the SQF. The function *flow* defined on the *Node* type specifies the data flow graph in which the node participates.

Inputs and *outputs* are special attributes that describe producer-consumer relationships between SQFs or a binding of SQFs to the input and output streams of the data flow graph. The *inputs* function returns a vector of elements that are either other nodes or input stream numbers. The *Outputs* function returns a vector of elements that are either other nodes or output stream numbers. When two nodes, $n1$ and $n2$, are connected through a producer-consumer relationship, node $n2$ has in its *inputs* function value an element equal to $n1$, while node $n1$ has $n2$ among the elements of its *outputs* function.

If a node does not consume the result of any other node in the data flow graph, it has in its *inputs* function only order numbers of input streams of the graph. We call such a node an *input point* of the graph. We use the term *output points* in a graph for nodes, such that there is no other node in the graph consuming their results. Since one node logically produces one result stream, an *output point* has in its *outputs* function only one order number of the output stream of the graph that the node produces.

However, when a graph $g1$ is connected by a pipe to another graph $g2$, it is possible to connect multiple nodes from $g2$ to consume the result stream from a single output point of $g1$. As a result the *outputs* function of the connected output point will contain references to all the consuming nodes in $g2$. Notice, that after $g1$ and $g2$ are connected, the output points in $g1$ are not any more output points for the composed graph.

The derived attribute *level* is used to store the level of a node in the topological ordering defined by the *inputs* function. The level of a node is computed as a sum of one and the maximum level of the nodes in the *inputs* function of the node. If a node is an input point, i.e. the *inputs* function contains only input stream numbers, the level of the node is set to zero.

The logical execution sites are instances of type *Site* with properties *id*, a unique identifier generated automatically by the system, and *location*, which specifies the name of the computing node to which the logical site is mapped. The mapping is performed by the system after the resources for the query have been allocated. A distributed execution is achieved by assigning graph nodes to different logical execution sites. The assignment is specified through the *assigned_to* function defined on the *Node* type returning an object of the type *Site*.

5.3 Templates Implementation

In this section we present the implementation of the data flow distribution templates from the built-in library.

```
1: function CENTRAL(fn, params, n)
2:     d ← dataflow(n, 1)
3:     s ← site()
4:     nd ← node(fn, params, vector(0, n − 1), {0}, d, s)
5:     return d
6: end function
```

Algorithm 1: Central template constructor

```
1: function PARTITION(n, tmpl, params)
2:     d ← dataflow(1, n)
3:     s ← site()
4:     for i ← 0, n − 1 do
5:         APPLY_TEMPLATE(tmpl,{n,i,params},{0},{i},d,s)
6:     end for
7:     return d
8: end function
```

Algorithm 2: Partition template constructor

5.3.1 Central Execution

The *central* template constructs a central data flow graph of one node as shown in Algorithm 1.

We use the following functions in Algorithm 1:

- *dataflow(arity, width)* is a constructor of a dataflow object setting its properties arity (number of input streams), and width (number of output streams). The central data flow in the algorithm has n input streams as the number of input streams of the SQF, and one output stream.
- *site()* is a constructor of *Site* objects;
- *node(fn, params, inputs, outputs, flow, assigned_to)* is a node constructor setting the attributes of the node with the corresponding names.
- *vector(0,k)* is a vector constructor that returns a vector of ordered elements in the interval $[0,k]$.

5.3.2 Partitioning

The *partition* template constructs a partitioning graph as in Figure 3.3a. The pseudo code is given in Algorithm 2. Algorithm 2 uses the dataflow constructor as in algorithm 1, but here the data flow graph has arity one and width n. n nodes are created and connected to different output streams of the graph to produce different partitions.

The template uses a system procedure *apply_template* (Algorithm 3) that creates a data flow graph by applying a template in its first parameter on a

```
 1: procedure APPLY_TEMPLATE(tmpl, params, inputs, outputs, d, s)
 2:     if template(tmpl) then                          ▷ Expand compute template
 3:         cd ← apply(tmpl, params)
 4:         BIND_INPUTS(cd,inputs)
 5:         BIND_OUTPUTS(cd,outputs)
 6:         for all nd ∈ nodes(cd) do
 7:             flow(nd) ← d
 8:             if s ≠ nil then
 9:                 assigned_to(nd) ← s
10:             end if
11:         end for
12:     else
13:         APPLY_TEMPLATE("central",{tmpl,params,count(inputs)},
    inputs, outputs,d,nil)
14:     end if
15: end procedure
```

Algorithm 3: Expand template or SQF invocation

vector of template parameters in the second parameter. The created nodes are included in a container graph d and assigned to a site s, the last parameters of the procedure. The *inputs* and *outputs* functions of the nodes in the graph are assigned correspondingly to the *inputs* and *outputs* parameters of the procedure.

The procedure contains a call to the predicate *template(fn)* that returns true if the parameter *fn* is a name of a template constructor. The *apply_template* procedure uses two procedures to bind the nodes of the new graph to the inputs and outputs of the container data flow graph d. *Bind_inputs* substitutes those elements in the *inputs* functions of the nodes in a graph d that are order numbers of input streams (hence, the predicate *numeric(k)*) with the stream order numbers specified in the *inp* parameter. *Bind_outputs* in Algorithm 5 has similar functionality, but for the output points. The function *nodes(d)* that returns all nodes in a data flow graph d. It is an inverse of the *flow* function. If the *tmpl* parameter is an SQF function, the procedure calls the *central* template to create a graph with one node annotated with the SQF.

5.3.3 Parallel execution

The Algorithm 6 presents the pseudo code of the *parallel* template constructor that creates s a parallel graph as shown in Figure 3.3b. Each of the parallel branches is created by a call to the *apply_template* procedure.

```
1:  procedure BIND_INPUTS(d, inp)
2:      for all nd ∈ nodes(d) do
3:          v ← inputs(nd)
4:          for i ← 0, count(v) − 1 do
5:              if numeric(v[i]) then
6:                  v[i] ← inp[v[i]]
7:              end if
8:          end for
9:          inputs(nd) ← v
10:     end for
11: end procedure
```

Algorithm 4: Binding inputs

```
1:  procedure BIND_OUTPUTS(d, outp)
2:      for all nd ∈ nodes(d) do
3:          v ← outputs(nd)
4:          for i ← 0, count(v) − 1 do
5:              if numeric(v[i]) then
6:                  v[i] ← outp[v[i]]
7:              end if
8:          end for
9:          outputs(nd) ← v
10:     end for
11: end procedure
```

Algorithm 5: Binding outputs

5.3.4 Pipelined execution

The *pipe* template takes two data flow graphs and creates a new data flow graph by copying the nodes and sites in the component data flow graphs and connecting the copies of the input points from the second component graph to the corresponding copies of the output points from the first component graph. Algorithm 7 shows the pseudocode. The following functions are used in the algorithm:

- *copy_nodes(d,map)* creates copies of all nodes and sites in the data flow graph d. The *map* parameter is a data structure that holds the mapping between an original node and its copy. Algorithm 8 presents it's pseudocode.
- *get_output(d,i)* is a function that returns the i-th output point of the data flow graph d, i.e. the node for which the value of the *outputs* function is set to $\{i\}$.

```
1: function PARALLEL(n, tmpl, params)
2:     d ← dataflow(n, n)
3:     for i ← 0, n − 1 do                                  ▷ Create branch i
4:         APPLY_TEMPLATE(tmpl,params,{i},{i},d,nil)
5:     end for
6:     return d
7: end function
```

Algorithm 6: Parallel template constructor

- *input_points(d)* is a function that returns all nodes that are input points of the data flow graph d.
- *thecopy(nd,map)* is a function that returns the copy of the node *nd* as specified by the mapping *map*.

5.4 CQ Management

Continuous queries require different management from a DSMS than the management of one-time queries in DBMS. Traditionally, a one-time query in a database finishes when the entire result set is produced having all input data processed. Since input data in stored databases are finite in size, the result is expected to be produced in a limited time. However, in the case of conceptually infinite stream data a continuous query can also run infinitely, as required for some monitoring applications such as monitoring plant environments. There are also cases when the user needs a query to run for a limited amount of time, for instance to monitor traffic conditions only in rush-hours. Therefore, a DSMS must allow a continuous query to run for a specified time interval or unconditionally. It also must provide an interface for explicit termination of CQs, which we call deactivation. The CQ management thus includes the following tasks:

1. Compilation of a data flow graph into an execution plan;
2. Starting the execution that includes installation and activation;
3. Monitoring the execution;
4. Deactivation of execution.

We will present each of these tasks in separate subsections.

5.4.1 CQ Compilation

Given the data flow graph and the input and result streams, the coordinator compiles the data flow graph into a distributed execution plan. The compilation algorithm contains the following main steps as shown in Algorithm 9:

```
1:  function PIPE(d1, d2)
2:      d ← dataflow(arity(d1), width(d2))
3:      map ← nil
4:      for all nd ∈ copy_nodes(d1, map) do
5:          flow(nd) ← d
6:      end for
7:      for all nd ∈ copy_nodes(d2, map) do
8:          flow(nd) ← d
9:      end for
    ▷ Connect copies of input points of d2 to the copies of output points of d1
10:     for all nd2 ∈ input_points(d2) do
11:         v ← inputs(nd2)
12:         for i ← 0, count(v) − 1 do
13:             nd1 ← get_output(d1, v[i])
14:             cnd1 ← thecopy(nd1, map)
15:             cnd2 ← thecopy(nd2, map)
16:             inputs(cnd2)[i] ← cnd1
17:             outputs(cnd1) ← outputs(cnd1) + cnd2
18:         end for
19:     end for
20:     return d
21: end function
```

Algorithm 7: Pipe

1. Determine node dependencies. The dependencies between nodes specified through *inputs* and *outputs* functions are analyzed and stored as a value of the *level* attribute of node.
2. Bind the nodes that are input points of the data flow graph to the input streams of the query. The binding uses the input stream numbers in the *inputs* function.
3. For each node, starting from the input points of the graph and following an increasing order of the *level* function values, compile the node. The compilation of a node is given below.
4. Connect the SQFs that are output points of the data flow graph to the result streams of the CQ.

The Algorithm 9 uses the following functions and procedures:

- *max_level(d)* returns the maximum among the levels of the nodes in a data flow graph *d*;
- *result_streams(n)* is a multi-value function that stores the result stream objects created for the node *n*

```
 1: function COPY_NODES(d, map)
 2:     for all s ∈ sites(d) do
 3:         news ← copy_site(s, map)
 4:     end for
 5:     for all nd ∈ nodes(d) do
 6:         newnd ← copy_node(nd, map)
 7:         assigned_to(newnd) ← thecopy(assigned_to(nd), map)
 8:         return newnd
 9:     end for
10: end function
```

Algorithm 8: Copying of a data flow

The compilation of a node presented by the pseudocode in Algorithm 10 has the following steps:

1. Perform a type check of the parameters of the SQF in order to derive the type of the result stream (line 2). The stream parameters are specified in *inputs* function, while non-stream parameters are given in the *params* attribute of the node. The type check requires that the input streams of the node are already bound when the node is processed. We ensure this by starting with the data flow input points with level value equal to 0 and following the SQF dependencies, i.e. increasing level values. The compiler infers the type of the result stream from the SQF definition. The type is needed by the result stream constructor.
2. Group the consumers of the compiled node according to their site assignments. For each group create a logical stream connecting the current node to the group of consumers. In this way the consumers in the group share the input stream produced by the node.
3. If both the node and the group of its consumers are assigned to the same site, the logical stream is implemented by a stream object with a main-memory stream interface assigned to the same site (lines 8-14).
   If the group of consumers is assigned on a different logical site than the current node, the logical stream is implemented by a pair of dual stream objects using inter-GSDM stream interface (lines 15-24). One of the streams (line 16), assigned to the current node's site, sends data from the current node to the site where the consumers group is assigned. The second stream object (line 17) is assigned to the consumers' site and receives data from the current node.

Algorithm 10 uses the following functions and procedures:

- *derive_result_type(n)* checks the types of the SQF parameters and returns the type of the result stream using the definition of the SQF annotating the node.

```
 1: function COMPILEDFG(d)
 2:     cq ← plan_of(d)
 3:     LEVELS(d)
                                   ▷ Connect all input points to the input streams
 4:     BIND_INPUTS(d,inputs(cq))
 5:     success ← true
 6:     for i ← 0,max_level(d) do
 7:         for all n ∈ nodes(d) do
 8:             if level(n) = i ∧ success then
 9:                 success ← compile(n)
10:             end if
11:         end for
12:     end for
13:     if success then
                               ▷ Connect the output point to the CQ result streams
14:         n ← get_output(d,0)
15:         for all outs ∈ outputs(cq) do
16:             result_streams(n) ← result_streams(n) + outs
17:         end for
18:     end if
19:     return success
20: end function
```

Algorithm 9: Algorithm for compilation of a data flow graph (DFG)

- *consumer_sites(n)* is a derived function that returns all objects of type *Site* to which some node is assigned that is a consumer of the argument node *n*;
- *nextstreamid()* is the system generator of stream identifiers;
- *mm_stream(name,stype,st)* is a stream constructor using the default parameters for streams in main memory inside a working node. The parameters specify the stream name, type, and a *Site* object representing the site where the stream object will be created;
- *inGSDM_stream(name,stype,st,src)* and *outGSDM_stream(name,stype,st,dest)* are stream constructors using the default parameters for inter-GSDM streams. The first three parameters are as described above. The last parameter is a source address for an input stream and a destination address for an output stream object. Notice, that the dual streams have the same name but are assigned to different sites;
- *subst(v,n,s)* is a procedure that substitutes the object *n* with the object *s* in the vector *v*. It is used to bind a node to an input stream produced by the node producer by the call $subst(inputs(n), nprod, s2)$.

```
1:  function COMPILE(n)
2:      stype ← derive_result_type(n)
3:      if stype then
4:          stprod ← assigned_to(n)
5:          for all st ∈ consumer_sites(n) do
6:              name ← nextstreamid()
7:              if st = stprod then              ▷ Create stream in main memory
8:                  s1 ← mm_stream(name, stype, stprod)
9:                  result_streams(n) ← result_streams(n) + s1
10:                 for all n1 ∈ outputs(n) do
11:                     if st = assigned_to(n1) then
12:                         SUBST(inputs(n1),n,s1)
13:                     end if
14:                 end for
15:             else    ▷ Create inter-GSDM stream of two dual stream objects
16:                 s1 ← outGSDM_stream(name, stype, stprod, name(st))
17:                 s2 ← inGSDM_stream(name, stype, st, name(stprod))
18:                 result_streams(n) ← result_streams(n) + s1
19:                 for all n1 ∈ outputs(n) do
20:                     if st = assigned_to(n1) then
21:                         SUBST(inputs(n1),n,s2)
22:                     end if
23:                 end for
24:             end if
25:         end for
26:         return true
27:     else
28:         return false
29:     end if
30: end function
```

Algorithm 10: Node Compilation

- *result_streams(n)* is a multi-value function that stores the result streams of a node *n*. It is used later on by the installation. The function contains a stream object for each group of consumers on different site.

5.4.2 Mapping

The compilation of a data flow graph creates an execution plan where distribution is specified in terms of logical execution sites. In order to start the execution, logical sites have to be mapped to physical computing nodes allocated by the resource manager of the coordinator. Here we assume that the computing resources have been allocated, so that the coordinator has on its disposal list of available computing nodes to be used for the query execution.

The mapping between logical sites and nodes is created by a system procedure that sets the *location* attribute of all *Site* objects in the data flow graph. Currently we always do one-to-one mapping between a logical site and a computing node to maximize the parallelism. Other mappings will be investigated as a future work.

```
1: procedure START(d)
2:     for all g ∈ sites(d) do                    ▷ All logical sites in d
3:         START_WN(location(g), id(g))
4:     end for
5: end procedure
```

Algorithm 11: Starting Working Nodes

When the mapping is set, the resource manager starts GSDM working nodes on the allocated computing nodes (Algorithm 11). When a working node starts it first registers itself to the name server and to the coordinator, and then starts its server loop in which it listens for the commands from the coordinator and data messages.

5.4.3 Installation

The installation of a continuous query consists of creating data structures and objects representing an executable plan at the working nodes. Algorithm 12 shows the main steps in the installation. There is no specific order of installation between working nodes, but internally at each node it follows node dependencies downstream (for loop in line 5).

The installation procedure is overloaded for the case when a stop condition is associated with the query. In this case, the condition is installed only at the input points of the data flow graph since the deactivation propagates downstream.

```
 1: procedure INSTALL(d)
 2:     for all g ∈ wn(d) do
 3:         maxlev ← maxlevel(d,g)
 4:         minlev ← minlevel(d,g)
 5:         for l ← minlev,maxlev do
 6:             for all v ∈ nodes(d) do
 7:                 if level(v) = l ∧ assigned_to(v) = g then
 8:                     for all s ∈ inputs(v) do
 9:                         INSTALL_STREAM(s)
10:                     end for
11:                     INSTALL_SQF(v)
12:                     for all s ∈ result_streams(v) do
13:                         INSTALL_STREAM(s)
14:                     end for
15:                 end if
16:             end for
17:         end for
18:     end for
19: end procedure
20: procedure INSTALL(d,kind,value)
21:     INSTALL(d)
22:     for all v ∈ input_points(d) do
23:         INSTALL_STOP(v,kind,value)
24:     end for
25: end procedure
```

Algorithm 12: CQ Installation

5.4.4 Activation

During the activation the SQFs in the execution plan are initialized and added to the list of active operators scanned by the scheduler. Algorithm 13 shows the activation steps. Since a node result stream is pushed to the consumers, the system must ensure that the consumers are already listening when a node starts producing data. This is achieved by activating the nodes in an order that is inverse to the node dependencies, i.e. upstream.

5.4.5 Deactivation

Deactivation of a continuous query consists of synchronized stopping of all SQFs in the query plan. The deactivation can be triggered by a stop condition associated with the query or performed on explicit command from the user.

Since the data flow graph runs in a distributed environment, we have devel-

```
1: function ACTIVATE(d)
2:     success ← true
3:     for i ← max_level(d), 0 do
4:         for all v ∈ nodes(d) do
5:             if level(v) = i ∧ success then
6:                 success ← activate(v)
7:             end if
8:         end for
9:     end for
10:    return success
11: end function
```

Algorithm 13: CQ Activation

oped a mechanism for synchronizing the deactivation on different nodes. It is initiated on all the input points of the data flow graph. Once they are deactivated, they propagate the deactivation to the consumers downstream, which on their turn are deactivated and propagate the deactivation until it reaches the output points of the execution plan.

5.5 Monitoring Continuous Query Execution

One of the important characteristics of continuous queries is that they are long-running by definition. Hence, it is highly probable that the system will experience changes in the execution environment during the runtime of a CQ due to changing rates of the source streams, changing availability of resources, or registering of new CQs that share resources with the existing ones.

Therefore, to provide efficient processing of CQs, the system should monitor the execution and adapt, if possible, the execution plans to fit better to the changed conditions. GSDM has a monitoring subsystem consisting of *statistics collector* modules in the working nodes and the coordinator. The query executor and the scheduler of working nodes invoke various statistics primitives to update the statistical meta-data. The most important parameters for monitoring in a distributed stream processing system are stream rates, CPU times for SQF executions, memory availability, memory utilization by the stream buffers, and communication costs. The statistics collector module of the coordinator periodically gathers statistical information from working nodes and performs analysis of the overall CQ performance.

5.6 Data Flow Optimization

The purpose of data flow optimization is to create an optimized data flow graph for a given continuous query. The optimization provides higher degree of transparency than the specification of queries through templates where the degree of parallelism and the partitioning strategy are specified explicitly. The optimizer automatically generates distributed execution plans and selects an optimized plan using some optimality metric. Traditionally database query optimizer functionality consists of *plan enumeration*, generating plans in the space of possible plans, and a *cost estimation model* on which the selection of an optimized plan is based.

We have developed a CQ optimizer for PCC with limited functionality that optimizes parallel execution plans for a single expensive SQF. Next, we shortly describe the components of the current CQ optimizer. The optimization of the individual SQFs relies on traditional query optimization.

5.6.1 Estimating Plan Costs

Traditional cost models rely on relatively accurate estimates of the costs of individual operators that are used to estimate the cost of the entire plan. However, in an extensible system such as GSDM executing user-defined functions over user-defined data, the cost model of individual functions might be hard to define or obtain from the author of the code. Furthermore, to allow utilization of processing resources allocated on-demand among computers with different performance parameters, the system would need to support a separate cost model for each architecture.

Therefore, the CQ optimizer selects an optimized plan based on *trial runs* that collect execution statistics rather than based on a cost estimate model. The optimizer collects statistics about the utilization time of working nodes, which is a sum of the SQF's processing times, the communication time, and the time spent in system tasks. The working node with maximum utilization time limits the throughput achievable by the data flow graph and hence we define an optimality metric (cost) of a plan as the maximum utilization time among the utilization times of working nodes is the plan. We select an optimized plan by selecting the plan with the lowest utilization time. In order to compare the statistics of several plans, the trial runs use the same cluster and work on a stream segment with equal size.

5.6.2 Plan Enumeration

A naive plan enumeration for PCC has been implemented as follows:

1. The maximum degree of parallelism is specified as a system parameter.

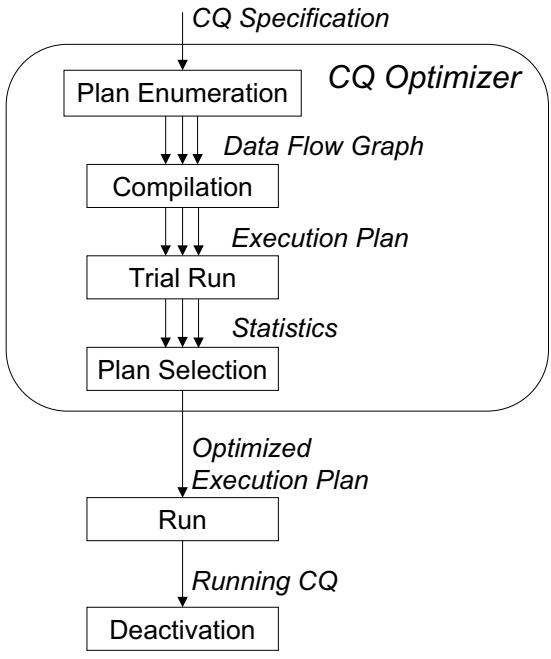


Figure 5.3: Life cycle of an optimized CQ

2. Plans for partitioned parallelism are generated using the PCC template constructor.
3. A *function registry* contains meta-data about the valid parallel strategies and their parameters for a given SQF. Both the valid stream partitioning strategies and the valid degrees of parallelism are specified in the registry.
4. The enumerator generates different plans by using the PCC template and varying the strategy (i.e. window split or window distribute) and the valid degrees of parallelism. The enumeration of plans for a given strategy stops when a plan has been generated such that either the maximum degree of parallelism or the resource limit is reached.

Each of the graphs is compiled, run in a trial mode, and statistics about the execution is stored in the coordinator's metadata by the statistics collector. The CQ optimizer then chooses an optimized data flow graph using the statistics collected and the above model for optimality. The life cycle of the CQ optimized in this way is shown in Figure 5.3.

The optimizer is implemented as a function with the following signature:

```
opt(Charstring templ, Charstring fun,
    Vector params, Vector inpstr) -> Dataflow d;
```

It takes as parameters the function *fun* to be executed, its parameters, and input streams *inpstr*. The first parameter is a template to be used for plan enumeration and currently only PCC is supported. The optimizer assigns automatically an output stream to collect statistics from trial runs. The result of the *opt* function is an optimized data flow graph for the provided parameter function *fun*. Using the CQ optimizer functionality, the continuous query on page 63 is specified by an alternative template constructor as follows:

```
set q = cq(opt("PCC","fft3",{},{s1}), {s1}, {s2});
```

The optimized plan is set by the *cq* constructor to the *plan* property of the query and used when the *run(q)* procedure starts the execution.

In the example above, plans with RR and user-defined stream partitioning are generated with different degrees of parallelism and the plan with best execution time from the trial runs is selected. In this way transparency is provided to the user, so that only the SQF *fft3* needs to be specified in the query, rather than all explicit parameters of the PCC template as in the example on page 63.

The current CQ optimizer provides automatic optimization of parallelelizable expensive SQFs using the PCC pattern. The experiments show that the optimization framework with trial runs and statistics collection is feasible, but it needs to be generalized in several important directions:

- Sophisticated plan enumeration. Since the naive enumeration of plans might create very big space of possible data flow graphs, in order to make the optimization efficient it is important to develop heuristics about which plans to generate and what order to follow during the generation. For example, enumeration strategies such as random walk of search space, binary search, or greedy can be investigated. The importance of such heuristics is even bigger in our setting than for cost-based optimization, because rather than computing the cost, the CQ optimizer runs a plan in a trial mode.
- Optimality model. Alternatively to the maximum throughput metric, other metrics such as latency and precision can be used. Furthermore, multi-criteria optimality model combining several metrics might show to fit better some applications.
- The optimization framework needs to be generalized to use different distribution templates besides the PCC template. For example, if a user specifies a pipeline of two SQFs, the CQ optimizer has to enumerate plans where each stage of the pipe is parallelized independently of the other by, possibly, different degrees of parallelism and partitioning strategies, and plans where the stages in the pipeline are executed together by defining a meta-SQF that encapsulates them, which on its turn is parallelized by some data partitioning.

6. Execution of Continuous Queries

This chapter presents the execution of continuous queries at working nodes. First, we describe the implementation of operators executing SQFs and inter-GSDM communication. Next, we present the scheduling policies used by the scheduler. Finally, we describe important observations concerning the system performance.

6.1 SQF Execution

The main execution primitive is an *execute* operator that sets up the environment and applies an SQF over its input streams and other parameters to produce logical windows in the SQF's result stream. Thus, the execution plan is composed of instances of the *execute* operator, each executing a particular SQF.

6.1.1 Operator Structure

For each SQF assigned to a working node the installation procedure creates an internal data structure (an *operator*) that contains the characteristics of the SQF as needed for execution by the *execute* operator. The structure is shown in Figure 6.1. The *id* slot stores the unique identifier of the SQF. The identifiers are generated automatically by the coordinator when the nodes in the data flow graph are created. The *SQF* slot is a pointer to an object of type *Function* representing the SQF to be executed. A *paramlist* stores a list of all parameters of the SQF. An *inputstreamlist* contains a list of stream objects that are parameters of the SQF, where for each stream there is an internal local buffer of pointers to the current logical windows in the stream. As we described in section 3.2.1 this buffer is accessed by the *window functions* in the SQF definition providing for SQF referential transparency.

The *outputstreamlist* holds a list of result streams of the SQF as determined during the compilation. Typically, the list contains one result stream object, but it is also possible to have more than one if the SQF has several consumers and they are assigned to different execution sites. In the latter case a different result stream object is created for each execution site where some SQF-

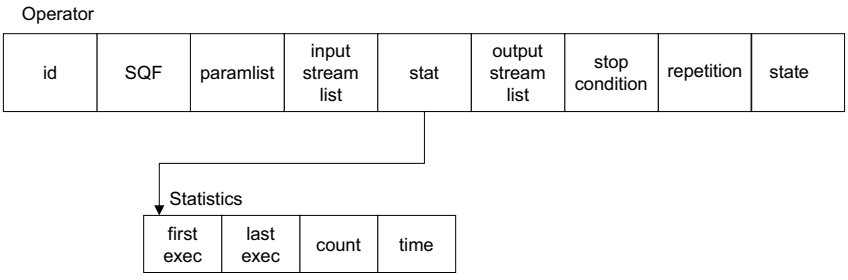


Figure 6.1: Operator structure

consumer is assigned (Algorithm 10). The logical windows from the SQF execution are inserted into each of the streams in the *outputstreamlist*.

The execution of long-running CQs is usually monitored by collecting and analyzing execution statistics. For each SQF there is a *statistics* data structure storing the initial and last time of execution, the number of executions, and the total processing time. The *stat* slot in the operator structure is a pointer to the corresponding *statistics* structure.

The *stopcondition* slot stores the stop condition associated with the SQF. Three kinds of stop conditions are supported in the current implementation: time-based, count-based[1] and unconditional. The time-based and count-based stop conditions are installed during the CQ installation by setting the *stopcondition* slot.

The *repetition* slot is used to set up an upper bound for the number of executions scheduled in one scheduling period. It is determined by the scheduling policy (to be explained below). The real number of executions might be smaller since it also depends on the data available.

Some SQFs, such as *S-Merge*, need to keep state in between subsequent executions. The *state* slot in the operator structure allows to store such state. To provide state initialization and clean-up, a pair of operations *initialization* and *cleaning-up* can be associated with the SQF. The operations are called during the activation and deactivation of the SQF, respectively.

An SQF can be in one of the following three states, illustrated in Figure 6.2: *uninstalled*, *installed*, and *active*, where a state transition is performed on a command from the coordinator or from the scheduler. As described in Chapter 2, all operator structures for SQFs installed at a working node are stored in a hash table *installed operators* with key the SQF's *id*. The *active operators*

[1] We used a count-based stop condition to provide an equal stream segment size for all experiments.

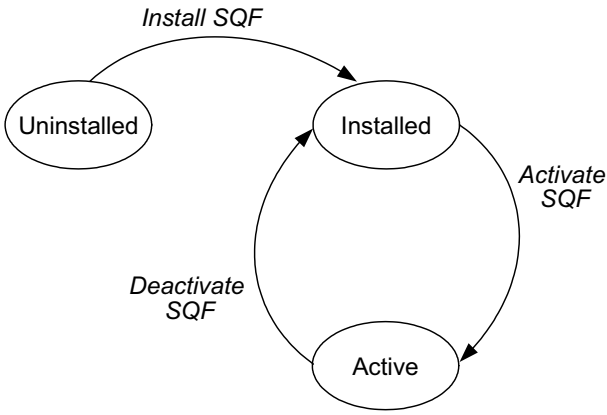


Figure 6.2: SQF states

list, used by the scheduler, contains pointers to the operator structures for the active SQFs. Assigning different states to the SQFs allows for flexibility in the execution plans. For example, changes of the execution plan on the fly are enabled by temporary deactivation of an SQF, change of its parameters, and activation or replacement with another SQF.

6.1.2 Execute operator

When the scheduler decides how many times to execute an SQF, it invokes the *execute* operator. The *execute* operator prepares an SQF for execution over the current input stream windows and executes it by a regular database query execution module to produce the next windows in the result stream.

The preparation of SQFs and the update of their result streams involve calls to the side-effect interface methods (Sec. 2.4) and, hence, are performed inside the *execute* operator.

In order to prepare an SQF for execution, the *execute* operator calls the *next* method of the stream interface for each of the input streams and sets up the content of the stream local buffers in *inputstreamlist* slot. In order to determine how many times the *next* method needs to be called, the system uses the parameters of the window functions in the SQF's definition, or values set by the *initialization* operation of the SQF if it needs state information.

For example, let an SQF is defined on a *jumping window* over a stream s, implemented as a sliding window of size n and step n ($slidingWindow(s,n,n)$). The *execute* operator prepares the stream s by calling the *next* method on it n times in order to obtain the pointers to the next n logical windows starting from the current cursor position. After the execution n pointers are dropped from the local buffer.

If an SQF is defined using a sliding window, i.e. $slidingWindow(s,n,1)$, it will be similarly prepared for the first execution by n calls to the *next* method on the stream s, but only one pointer will be dropped after the execution. All the following executions will be prepared by one call to the *next* method, since the rest $n-1$ pointers are remaining in the buffer after the previous execution. In both cases the sliding window function will return a vector of n logical windows relative to the current cursor position.

SQFs with time-based sliding window functions are prepared based on the time stamp characteristics of data. If an SQF is defined using a time-based sliding window, i.e. $timeWindow(s,span)$, the *execute* operator calls the *next* method to retrieve the pointer to the next available logical window with time stamp ts and drops from the local buffer all pointers to data with a time stamp smaller than $ts-span$.

After the data from the input stream is prepared in the local buffers, the *execute* operator executes the SQF by the regular database query engine in Amos II, providing function parameters from the *paramlist* slot of the operator structure. The produced result logical windows are added to the registered result streams of the SQF by calls to the corresponding *insert* interface methods.

If an SQF is scheduled when there is not enough stream data, the window functions return *nil* and the call to the SQF does not create result windows.

6.1.3 Implementation of S-Merge SQF

The semantics of *S-Merge* SQF is to merge data from many streams preserving the order determined by an ordering attribute, in our case a time stamp. Assuming that the order is preserved inside each of the merged streams, the execution is just a selection of the logical window with the smallest time stamp among the current logical windows of all the merged streams.

S-Merge is designed to merge arbitrary number of streams and hence its first parameter is of type *Vector of Stream*.

The semantics of *S-Merge* requires all the merged streams to have data in order to compute the smallest time stamp. However, delays due to distributed processing and communication, loss of data, or end-of-stream condition may create a situation when some of the stream buffers are empty. In order to provide non-blocking behavior in such a situation, the operator is associated with a *time-out* parameter. If data on a stream is not available, *S-Merge* waits for *time-out* time period after the first attempt to obtain it. If data is still not available after this period, *S-Merge* assumes that data has been lost and selects the logical window with the smallest time stamp among the streams with non-empty buffers. If a delayed logical window comes later on, *S-Merge* ignores it to preserve the order of already produced result stream. The *S-Merge* pseudocode is shown in Algorithm 14.

```
 1: function S-MERGE(vs,timeout)
 2:     res ← nil
 3:     empty ← check_empty_buffers(vs)
 4:     if (empty ∧ timer = nil) then
 5:         timer ← now()                ▷ Start timer if empty buffer is detected
 6:     else if (¬empty) ∨ (empty ∧ timeout_exp(timer,timeout)) then
 7:         ts ← min_ts(vs)
 8:         s ← stream_min_ts(vs,ts)
 9:         MARK(S)
10:         timer ← nil
11:         if ts ≥ last_ts then
12:             res ← currentwindow(s)
13:             last_ts ← ts
14:         end if
15:     end if
16:     return res
17: end function
```

Algorithm 14: S-Merge Algorithm

The algorithm uses the following functions and procedures:

- *check_empty_buffers(vs)* is a predicate that checks that all the buffers of streams in the vector *vs* contain data.
- *timeout_exp(timer, timeout)* is a predicate that checks for time-out expiration.
- *min_ts(vs)* finds the minimum time stamp among the time stamps of the current logical windows in the streams *vs*;
- *mark(s)* puts a marker to the stream from which the logical window with minimum time stamp is chosen. The *execute* operator uses the mark to delete the pointer in the local buffer for this stream when *S-Merge* finishes;

The selected logical window with minimum time stamp is emitted only if its time stamp does not violate the result ordering (lines 11-14). Values of *timer* and *last_ts* are kept in the *S-Merge* state in the *state* slot of the operator structure.

6.1.4 Implementation of OS-Join SQF

The semantics of the *OS-Join* SQF is to join data from multiple streams on their ordering attribute time stamp and to apply a combining function on the joint data preserving the order in the result stream. The first parameter is of type *Vector of Stream*, and the second is the combining function to be applied.

```
 1: function OS-JOIN(vs, combinefn)
 2:     res ← nil
 3:     n ← count(vs)
 4:     lw ← currentWindow(vs[0])
 5:     if lw then
 6:         for i ← 1, n − 1 do
 7:             lwi ← currentWindow(vs[i])
 8:             if lwi then
 9:                 PUT(hash(vs[i]),ts(lwi),lwi)
10:                 MARK(vs[i])
11:             end if
12:         end for
13:         params[0] ← lw
14:         t ← ts(lw)
15:         for i ← 1, n − 1 do
16:             params[i] ← get(hash(vs[i]), t)
17:         end for
18:         if ¬empty(params) then
19:             res ← combinefn(params)
20:             for i ← 1, n − 1 do
21:                 DELETE(hash(vs[i]),t)
22:             end for
23:         end if
24:     end if
25:     return res
26: end function
```

Algorithm 15: OS-Join Algorithm

OS-Join needs a policy to provide non-blocking behavior in case some of the logical windows are missing. Since a result window is produced by combining sub-windows from all streams, there are two alternatives in case of missing data: either the system waits until data come with the potential danger of blocking, or OS-Join waits for a *time-out* time period and produces a partial result using, e.g., replication of old data or approximation.

In the current implementation *OS-Join* waits for the data to come, which in practice shows non-blocking behavior. The reason is that *OS-Join* combines inter-GSDM streams created by splitting an original stream and communicated on TCP that guarantees loss-less and order-preserving receiving of streams. The pseudocode of *OS-Join* is presented in Algorithm 15. The algorithm uses the first stream as a probing source and keeps a hash table for each other stream from the vector of streams *vs* in the *OS-Join* internal state.

The following functions and procedures are used in Algorithm 15:

- *hash(s)* is a function that given a stream object returns the hash table for the object in the internal state of *OS-Join*.
- *put(h,k,val)* inserts the value *val* in the hash table *h* using the key *k*
- *get(h,k)* retrieves the element with key *k*.
- *delete(h,k)* deletes the element with key *k*.
- *empty(v)* is a predicate that checks if the vector v has elements with nil values.

In order to implement the time-out alternative that is non-blocking in general without the above assumptions holding in our case the time-out parameter should be added as in *S-Merge*. In addition there is a need for specification of partial results computations, e.g., by an additional parameter *partcombine*, that is a function called to compute the partial results.

6.2 Inter-GSDM communication

The communication between GSDM servers is message-based where the messages contain commands to be executed at the receiving server. GSDM servers can send messages *synchronously* or *asynchronously*. When a message is sent synchronously, the server executes the command contained in it and sends the result of the execution back to the server sending the request. The requesting server waits for the result before continuing its work.

When an asynchronous message is sent to a server, the sending server continues its work without waiting for the result. The message is annotated with information whether the result of the execution should be stored at the receiver to be retrieved later on. The command in the message is executed at the receiving server and the result is stored locally if specified so.

The coordinator commands to the working nodes are implemented as synchronous messages. The predefined primitives include, e.g. the installation and activation commands:

```
install_stream(Charstring stype, Charstring sname,
   Charstring srcaddr, Charstring destaddr,
   Charstring interf)-> Stream
install_SQF(Charstring id, Charstring SQFname,
   Vector strargs, Vector args)-> Boolean
install_stop(Charstring id, Charstring cond,
   Number val) -> Boolean
activate(Charstring id)-> Boolean
```

Furthermore, arbitrary commands can be sent to the working nodes, e.g. for setting some system parameters such as the size of the database image in main

memory. By sending coordinator commands as synchronous messages status information is atomically provided to the coordinator.

Stream data are *pushed* between working nodes using asynchronous messages without waiting for the result of message execution. They contain a call to the *insert* method of the stream interface for the corresponding remote inter-GSDM stream object and the data to be inserted.

6.3 Scheduling

The work of the CQ engine is controlled by its scheduler. It executes a loop where in each iteration the active SQFs are executed on the current stream data. Through this loop the CQ engine achieves continuous execution of CQs.

The scheduling policy affects substantially the system performance. It is determined by two components: how periods of work (loop iterations) are organized and how SQFs are scheduled inside a period. We call one iteration of the scheduler loop *scheduling period*. In each scheduling period the scheduler allocates resources for three groups of tasks:

- Communication: the system checks for incoming inter-GSDM messages with commands or data.
- SQFs: the *active operators* list is scanned and each SQF there is checked and eventually scheduled and started.
- System tasks: for example management of stream buffers is performed periodically and SQFs are checked for deactivation.

6.3.1 Scheduling periods

Two scheduling policies with respect to the length of the scheduling period were implemented, evaluated, and shown to fit in different situations.

The first scheduling policy has *fixed length* of the scheduling periods, where the length is called *turn-around* time. Its purpose is to control the maximum rate at which data is consumed from the external input streams and to prevent in this way overload situations at down-stream working nodes. It also provides regular inter-arrival intervals for the stream data at the consuming servers.

The policy is implemented by a special *sleep* task that causes idling of the processor to the end of the scheduling period. If the execution time for all the tasks in a period is bigger than the period turn-around time, the sleep is skipped and the scheduler sends a message to the coordinator notifying it about overload.

The second scheduling policy does not have fixed length scheduling period and hence we call it *variable-length*. The length of a given scheduling period depends on the active SQFs and the amount of unprocessed stream data. When

the engine finishes with the processing tasks in the current period, it starts a new period, checks for new data, and schedules their processing. In other words, the new period starts immediately after the work in the current period finished, so that data that has come in the meantime does not need to wait unnecessarily. By contrast, with fixed-length period new data has to wait until the beginning of the next period even if the current processing has finished and the system is idling.

6.3.2 SQF Scheduling

In the time frame of one scheduling period different policies are possible with respect to the execution order among the SQFs and the number of executions for each SQF. We call this *SQF scheduling policy*.

Currently, the execution order among several SQFs is determined by their order in the *active operators* list. Scheduling problems concerning ordering of stream operators are active area of research of stream processing engines [9, 18]. Most projects work in the context of a large number of relatively cheap operators installed at the same server, which makes the ordering of operators an important part of the scheduling. By contrast, we do not address SQF ordering problem and use a simple queue-based ordering since our focus are expensive SQFs split into multiple instances for parallel processing. In this settings, most of the servers execute a single expensive SQF over a data partition.

We implemented the following two SQF scheduling policies concerning the number of scheduled executions:

1. *Fixed-number* SQF policy determines the number of SQF executions per scheduling period as specified in the *repetition* slot of the SQF.
2. *Greedy* SQF policy checks the amount of unprocessed data in the input stream buffers and based on that determines the number of SQF executions for the current period.

The scheduling policy with fixed number of repetitions is applicable for all SQFs, but is especially useful for the SQFs that are input points of the data flow graphs. Fixing the number of executions per scheduling period together with fixing the length of the period by the *turn-around* parameter allows the system to control the execution rates of the SQFs that are input points of the data flow graph and, hence to control the rate with which the system "consumes" external stream data.

The greedy scheduling intuitively allows the system to do as much work as it can and as soon as possible. It requires that the scheduler is able to check the amount of unprocessed data and their time stamps. We combine greedy SQF scheduling with variable length periods on internal working nodes to process as much data as possible and as soon as the data has been received,

given that there are available processing resources. We do not use greedy SQF scheduling in combination with fixed-length period, since in overload situations greedy scheduling might schedule a big number of executions that exceeds the turn-around time of a fixed-length period.

The scheduling period and the SQF scheduling policy are determined by the coordinator using the above rule about the input points of the data flow graph and their assignments to working nodes.

For each SQF in the active list, the scheduler performs the following steps:

1. Checks the stop condition of the SQF. If it is true, the SQF is not scheduled.
2. Checks if the SQF has fixed number of repetitions per scheduling period. If so, the number of scheduled executions is set to the value of the repetition slot.
3. If the SQF does not have fixed number of repetitions, the scheduler uses greedy policy, i.e. determines the number of scheduled executions based on the amount of unprocessed data and the SQF parameters.

6.3.3 Scheduling of System Tasks

At the end of each scheduling period system tasks might be scheduled as follows:

- The buffer manager is called periodically to clean the stream buffers from data that all the SQFs have processed. A system parameter determines how often the buffer manager should be invoked.
- The scheduler checks the stop conditions of the SQFs and invokes the deactivation procedure if needed.

6.3.4 Effects of scheduling on system performance

We conducted several experiments to investigate the effect of different scheduling policies on the overall system performance. We measured the performance by the average response time (latency) that a logical window spends in the system, the load of the nodes, and the time spent in communication.

The purpose of the first experiment is to illustrate the effect of fixed-length and variable-length scheduler periods on the latency. We set up a data flow graph of two working nodes, WN1 and WN2, where WN1 sends logical windows regularly to WN2, which executes a very fast identity SQF. The first node uses fixed-length scheduling to provide regular inter-arrival intervals at the second node. Since an identity SQF is very cheap, the load of both working nodes is determined by the communication costs and is very low. The main source of latency in this case is the time for communication and the waiting time due to the scheduling policy.

Turn-around	Rate in LW	Load %	Lat Fixed	Lat Var
0.08	12.5	3.94	0.009	0.0044
0.1	10	3.18	0.104	0.0044
0.2	5	1.74	0.198	0.0044
0.3	3.33	1.18	0.302	0.0046

Table 6.1: *Latency with fixed and variable length scheduling period*

Table 6.1 shows the measured load and latency for logical windows of size 2048 and different input stream rates. When WN2 uses fixed-length scheduling, we measure latency that increases proportionally to the length of the scheduling period. The reason is that when a logical window comes to the working node after the scheduling period has started, it waits for the next scheduling period in order to be processed. The actual waiting time depends on the synchronization between the data arrivals and the beginning of the scheduling periods.

When WN2 uses variable-length scheduling period, the system checks the TCP sockets for incoming data messages and processes them as soon as the previous period finishes. Since the node has low load this means immediate processing of the incoming logical windows. We measured a constant latency of about 0.0044 sec. (last column in the table) for different input stream rates, which can be attributed to the communication latency.

This experiment shows that the variable-length scheduling gives stable and shorter latency than the fixed-length scheduling when working nodes have low load.

The purpose of the second experiment is to investigate whether the advantage of variable-length period still holds when the system load increases. We choose again a logical window size of 2048 (approximately 50 KB) and a data flow graph of two working nodes, where the second one executes the *fft3* SQF with processing cost of 0.107 sec. per logical window. We increased gradually the system load by increasing the input stream rate. The measurements for the load and latency are shown in table 6.2. We measured the variable-length scheduling with two versions: one with greedy SQF scheduling, and one with a number of repetitions fixed to one. The measurements of both versions are very similar[2] and presented together in the last column in table 6.2.

Again the variable length period scheduling shows smaller and stable values

[2]In the case of under load the inter-arrival time is longer than the processing time for a logical window. Hence, assuming regular inter-arrival intervals, the greedy SQF scheduling de facto schedules the SQFs for either zero or one execution, which gives the same effect as the scheduling with fixed-number repetitions equal to one.

Turn-around	Rate(LW)	Load %	Lat Fixed	Lat Var
0.3	3.3	36	0.115	0.110
0.2	5	57	0.31	0.111
0.15	6.6	73	0.259	0.111
0.12	8.33	94	0.169	0.112

Table 6.2: *Latency with fixed and variable length scheduling period*

Turn-around	Rate(LW)	Lat	Comm WN1
0.1	10	0.553	0.266
0.09	11.1	1.149	0.266

Table 6.3: *Latency at the receiver and communication overhead at the sender with fixed-number SQF scheduling*

of the latency: we measure an average latency 0.111 out of which 0.107 is the *fft3* processing cost. From this experiment we can conclude that the variable length scheduling period indeed processes logical windows as soon as they come, given load up to the measured 94%. Latency with fixed length period is bigger and not proportional to the turn-around length, but varies depending on how long logical windows need to wait until new period starts.

The goal of the third experiment is to investigate the trade-offs between fixed-number and greedy SQF scheduling when the system is overloaded. We increased the stream rate to values for which the average inter-arrival time is shorter than the processing time for a window. Tables 6.3 and 6.4 show the results for inter-arrival interval set to 0.1 and 0.09, respectively, given the *fft3* processing cost of 0.107. We observe that the greedy scheduling shows shorter latency than the fixed-number scheduling, but the communication times for sending windows increase rapidly at the sender node WN1. The reason is that since the processing cost is bigger than the inter-arrival time, during some periods more than one window come and are scheduled by greedy. This increases the length of the period and postpones the moment when next data is read from the TCP sockets. In other words, the GSDM server does not consume data from the TCP buffers in a timely manner, which results in filling the buffers and activating the TCP flow control mechanism. As a result the communication cost for sending logical windows at the first GSDM node increases and the actual rate of sending decreases, e.g. to 9.47 instead of 10 as it is set.

This experiment shows that in the case of overloading, greedy scheduling

Turn-around	Rate(LW)	Lat	Comm WN1
0.1	9.47	0.393	2.77
0.09	9.58	0.43	3.66

Table 6.4: *Latency at the receiver and communication overhead at the sender with greedy SQF scheduling*

would activate the TCP flow control mechanism and through it would eventually reduce the rate at the sender due to increased communication overhead. Therefore, an important system parameter to monitor for system overload is the communication time at sending nodes. When this time increases above some threshold the GSDM engine should take actions to reduce the input stream rate in a controllable way. Notice, that the overload at the downstream working node does not cause local loss of data since TCP is used for inter-GSDM communication. Instead, the input rate at the upstream working node reduces, i.e. new incoming data is accumulated and eventually dropped on the entry of GSDM before any processing cost to be spent on it.

If overload occurs with fixed SQF scheduling policy at the receiving node, there will be periods when the number of windows scheduled for processing is less than the number of windows received and not processed. Hence, data accumulates in the stream buffers and their latency increases as shown in Table 6.3.

Therefore the overload in this case is detected by monitoring the state of the stream buffers for overflow. The system needs a policy to apply when stream buffers are filled up, but such a policy has not been implemented yet. The most common overload policy in this case is load shedding [83] which drops some of the data without processing based on some rules. Notice, that data loss due to the shedding would occur at the receiver node after the stream windows have been processed by at least the GSDM-sending node. In the current application the source streams are received using the UDP protocol and data loss occurs at the input working nodes in case of overload.

Having investigated the effects of the scheduling on the communication and processing, we choose to use greedy SQF scheduling with variable-length periods for the internal working nodes, including the parallel working nodes. Under internal working nodes we mean those that do not have SQFs operating on external streams. In case of overload, greedy scheduling causes communication overhead at the sender - the partitioning node, which allows the system to start dropping data *before* it is processes by the remaining, more costly, part of the data flow in a way that does not affect the parallel branches.

In the same situation fixed-number scheduling at parallel nodes would start

dropping data. However, this dropping would not be synchronized among the parallel branches, thus causing higher percentage of dropped result windows for window split strategy due to dropped sub-windows. Furthermore, part of this dropping would actually happen at the combining node after the expensive work (computational SQFs) has been done.

6.4 Activation and Deactivation

The activation of an SQF includes two steps: adding the operator structure for the SQF to the list of *active operators*, and calling an *init* operation. The latter opens the SQF's input and result streams by calls to the *open* method of their stream interfaces. For SQFs with foreign implementation, the associated initialization operation is also performed that prepares the SQF internal state.

The deactivation starts at the input points of the data flow graph and propagates downstream. The propagation of the deactivation in a distributed environment is currently performed by sending a special *end-of-stream* message to the stream consumers notifying that there will not be any more data on this stream. Therefore, each SQF has an implicit stop condition that evaluates on true if all the input streams of the SQF have empty buffers and have received the *end-of-stream* control message.

At the end of each scheduling period the scheduler checks the SQF's explicit and implicit stop conditions and issues a command for deactivation of the SQF if some of them evaluates on true. The deactivation includes three steps: removing the SQF from the list of *active operators*, sending the *end-of-stream* control message to the consumers of its result stream, and calling a *clean-up* operation.

The clean-up operation closes the input and result streams by calls to the *close* method of their stream interfaces, and performs the associated with the SQF cleaning-up operation. The *close* method for a shared stream does not destroy the stream buffer, but only nullify the *cursor* in the stream buffer for this particular SQF.

After deactivation the operator structure remains in the hash table of the installed operators and the SQF can be activated again later on.

6.5 Impact of Marshaling

In a distributed stream processing system data communication is intensive. The experiments with the GSDM prototype show the importance of a carefully designed stream communication in order to achieve good system performance.

We experimented two encodings for the scientific data. In the initial implementation the encoding was character based so that all numbers had to be

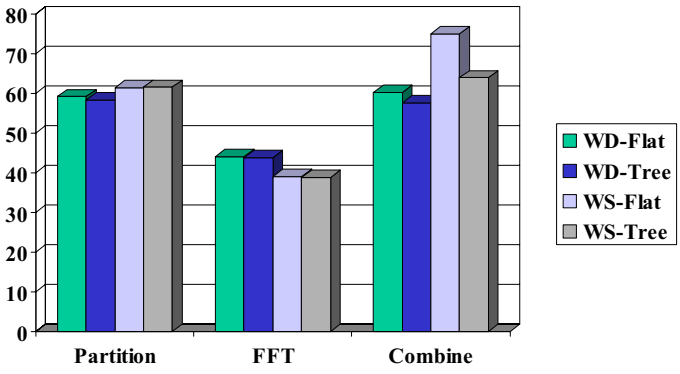


Figure 6.3: Real time spent in partition, FFT, and combine phases of the parallel-4 strategies for logical window of size 8192, fast implementation, and **character** *marshaling.*

parsed. We measured the cost for marshaling and de-marshaling of logical windows containing vectors of complex numbers into character format when sent to and received from the TCP sockets. This cost was so high in comparison with the computational costs, that the parallel execution strategies got a bottleneck in the partition and combine nodes already with degree of parallelism four. Figure 6.3 from [45] shows the saturation of the communicating partition and combine nodes for the fast FFT implementation, logical window of size 8192, with character marshaling, to be compared with Figure 4.11b for the same size and implementation, but with binary marshaling.

To address this problem, we replaced the encoding of logical windows into a binary format. As a result the size of the communicated data decreases with approximately 50% and the time spent on (de)marshaling was substantially reduced by 6 to 10 times. This is illustrated in Figure 6.4 showing the times for marshaling and de-marshaling of logical windows with binary and character encoding. The times increase linearly with the logical window size for marshaling into and from character format. We observe linear increase for marshaling into binary format for sizes bigger than 1024, while for small window sizes the overhead for calling the marshaling function for a window becomes substantial in comparison to the marshaling operation itself. The improvement achieved by using binary format is between 6 for small sizes to 10 times for big logical windows. The times measured for the binary representation included copying of a logical window from the internal GSDM format into binary format. Therefore, those times can be further improved by changing the implementation to avoid the copying. However, the current implementation is efficient enough to allow for data flow graphs with high degree of parallelism

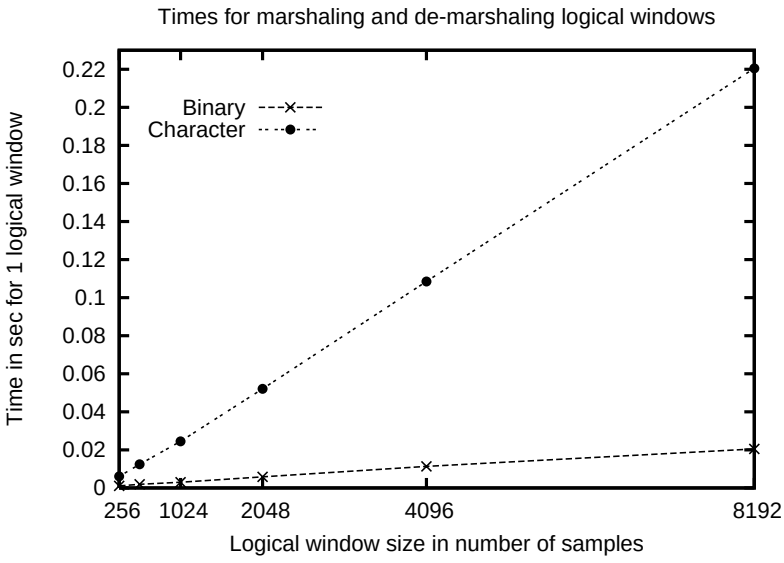


Figure 6.4: Times for marshaling and de-marshaling of logical windows with different encoding

for expensive SQFs. This experience illustrates the importance of a well-tuned communication in a distributed on-line stream processing system.

7. Continuous Queries in a Computational Grid Environment

In order to execute efficiently expensive continuous queries on high-volume data streams, GSDM needs substantial computational resources. At the same time, resource requirements are varying in time depending on the currently installed queries and their computational costs. The idea behind Grid middleware is to provide sharing of resources, such as computers, data, and instruments. In this chapter we investigate the possibilities to utilize computational Grid resources on-demand for the needs of computationally expensive continuous queries.

7.1 Overview of Grids

Grids [43, 33] enable sharing of distributed heterogeneous resources, such as computers, databases, and scientific instruments. The term *Grid* was chosen as an analogy to the electrical power grid that provides transparent, pervasive access to electric power irrespective of its source. The purpose of computational Grid middleware software is to provide an infrastructure where users have transparent access to computing resources irrespectively of where the computing cycles are generated. This idea is a basis for computational markets where service provides offer computational resources and users select a resource based on criteria including price and quality of service [65].

Several projects, such as Globus [35] and Nordugrid [61], provide tools for building computational Grid infrastructures.

In a typical usage scenario the end users interact with the Grid middleware by submitting requests, called jobs, for execution of an application program on some input data sets. In order to provide transparent execution of jobs, the middleware software must provide capabilities for dynamic discovery, selection, and allocation of grid-enabled resources. It also has to provide tools for job description, staging the application executable, transferring the necessary input data files, submitting jobs to the resource's management system, monitoring the execution, and notifying the user and transferring the results when the job completes. In order to fulfill this role, Grid middleware needs to solve a number of problems. The main challenges are heterogeneity of resources

and technologies, scalability, multiple ownerships, preservation of autonomy, and dynamics.

Over time the Grid community realized that the required capabilities of Grid middleware overlapped with the developments in the web services community. As a result, a service-oriented architecture for the Grid was proposed, known as Open Grid Services Architecture (OGSA) [32]. Based on OGSA, the Grid middleware functionality is available through service description and invocation standards of the web services community.

7.2 Integrating Databases and Grid

The initial driving forces for the development of the Grid were in the area of scientific and engineering applications. Data in these initial applications were organized as flat files and consequently the early Grid toolkits focused on file support rather than management of structured data. As the file management systems and registries associated with Grid became complex, DBMSs started to be increasingly used to store the Grid meta-data. At the same time more scientific applications emerged using the developments of, e.g., OO- and ORDBMSs to organize their complex data. The requirement of scientific collaboration created a need to be able to share such databases on the Grid.

In order to integrate databases with Grid infrastructure, Grid middleware stack needs to include service-based interfaces to databases [65]. In many cases their functionality is similar to the existing database access interfaces such as JDBC. However, more expressive interfaces are needed for request-response and publish-subscribe access to data.

Database technologies and Grid can mutually benefit of each other. Database management systems can benefit from unified access to computational and storage resources, and from facilities for dynamic resource discovery and allocation. The Grid can benefit from database technology that provides the users with data sharing, scalable management of large data sets, and declarative query languages. Data intensive tasks on Grid are typically specified as directed acyclic graphs (DAGs) in terms of jobs, i.e. programs executed on files. A higher level abstractions are needed to ease the specification of Grid data-intensive tasks. Examples of projects that provide such abstraction are GridDB and OGSA-DQP [4].

7.3 GSDM as an Application for Computational Grids

The GSDM system can benefit from Grid facilities for dynamic resource discovery and allocation. The system has highly varying resource requirements.

When a new query is submitted, containing possibly expensive SQFs, the system needs to provide on-line execution with high throughput. It can be achieved by composing a parallel execution plan to run on newly incorporated resources. When the query is stopped, the resource requirements of the system decrease correspondingly. Therefore, dedicating a fixed number of servers to the system can lead to over-provisioning and inefficient resource utilization in case the current load of the system is small. At the same time, the dedicated resources can still be insufficient to provide for an efficient execution of very expensive queries. Hence, this dynamics of resource requirements makes GSDM a good candidate for shared computational resources that are provided *on-demand* by a computational Grid infrastructure.

7.3.1 GSDM Requirements for Grids

The other side of the coin is the question how GSDM fits into the profile of the current computational Grid applications. In the following we analyze what requirements GSDM puts to a computational Grid and how they are fulfilled by the current Grid middleware.

- *Requirement I: High communication bandwidth to the processing nodes.*
  Traditionally database systems store high-volume data on disks. Hence, data locality is an important factor in query processing, in other words, it is preferable to perform the processing as close to data as possible to avoid transferring of high-volume data. In a stream processing system where stream data are not disk-stored, but communicated in real-time, this requirement changes. It is again favorable to process the stream data in proximity to the stream sources, but it is more important to use resources with high-bandwidth communication capacity for high-volume streams.
- *Requirement II: Automatic staging of executables.*
  GSDM is a main-memory database engine for stream processing and as such it can be started by simply starting the executable with the initial database image file. Therefore, in order to run the system on an arbitrary resource, staging of the executable is needed similarly to other Grid applications.
- *Requirement III: Support for parallel long-running jobs with guaranteed short start-up time.*
  Distributed execution plans for computationally expensive continuous queries require multiple nodes to work simultaneously. Therefore, parallel jobs need to be supported. CQs are also often long-running. At the same time, when a user submits a continuous query she expects *short start-up time* in order to analyze the data currently produced by an instrument, sensor, etc. A start-up with an indeterminate delay means unprocessed stream data and may lead to missed important discoveries.

- *Requirement IV: IP Connectivity of individual computing nodes.*
  Processed data are not locally stored on disks, but comes continuously from external sources, such as instruments and software on the Internet. Therefore, individual computing nodes need to have IP addresses accessible for data delivery from outside. Resources where all the communication goes through a single front-end node cannot manage high-volume streams communication.

The above requirements are not completely satisfied by the current state of computational Grids and Grid middleware. True staging of executables is not automatically provided by the current Grid middleware. Consequently, today's Grid applications are limited to use only Grid resources where the application executables have been prepared in advance, typically in a manual way.

Resource management of parallel computing resources is typically performed by batch-oriented systems, such as PBS for cluster computers. When a job is submitted, the batch system puts it to a batch queue according to job's priority and schedules it following some resource management policy. As a result the job might wait long before the resources are allocated for it. Correspondingly, the job submission using Grid middleware is *batch* oriented with unpredictable waiting times.

The resource management systems of clusters often dedicate parts of the resources for interactive remote-session jobs for the purposes of application development and testing. These interactive jobs have typically short start-up times and it is possible to acquire several nodes at once as long as there are resources available. Interactive jobs (rather than batch jobs) could be used for servicing GSDM queries because of their short start-up times. However, cluster's resource managers often limit interactive jobs to be started only from the terminal sessions which prohibits programmatic startups. Furthermore, since the interactive jobs are not considered Grid production jobs, most Grid middleware toolkits do not provide support for them.

Many computer resources accessible through the Grid infrastructure do not currently satisfy these requirements. At the same time the number of monitoring applications processing streams increases and includes new domains. It is therefore desirable to consider the requirements of such applications for computational resources when new Grid-enabled computer resources are designed.

7.3.2 GSDM Resource Allocation

In a Grid environment the GSDM resource manager module in the coordinator would:

1. Collect information about the status of the available cluster resources where the GSDM executable is pre-installed. Of particular importance is the abil-

ity to get information about the approximate waiting time for the submitted jobs.
2. Select a cluster based on the expected number of available nodes and the waiting time.
3. Submit a (interactive or batch) job to the cluster. The job contains a number of nodes as a parameter. As a result a list of available computer nodes is provided to other modules of the GSDM coordinator.
4. Multiple jobs can be submitted in a proactive way so that a minimum number of nodes is always guaranteed.

7.3.3 Multiple Grid Resources

In the presented usage scenario a CQ runs on a single cluster computer. We can also consider an execution of a CQ distributed among several clusters. Using techniques from distributed databases, the CQ can be decomposed into a number of continuous sub-queries assigned on different clusters based on their proximity to the stream sources. Such assignments would reduce the total communication traffic. Furthermore, the overall system efficiency can be optimized by installing a new CQ on clusters where other queries already process the same streams.

In order to implement such a scenario for processing distributed among several clusters, a number of problems need to be addressed, for example:

- CQ decomposition and site assignment need to take into account physical proximity of stream sources and processing resources, as well as queries currently running in the system.
- Running on-line stream sub-queries on different clusters requires very strict synchronization of resource allocation among clusters, which is currently not supported.

7.3.4 Grid Requirements for Applications

In order to run GSDM as a Grid application, the system needs to conform to Grid standards and requirements, such as:

- *Authentication and Authorization.* Users of Grids acquire resources through a *grid certificate* schema. The Grid applications run while utilizing the credentials of the user certificates, where the principle of a single sign-on is applied. Hence, it is necessary to investigate a certificate utilization schema in order to run GSDM as a Grid application. For instance, the system can use its own grid certificate, in which case the user interface needs to be augmented with user authentication and authorization mechanism to ensure that the system runs continuous queries only on behalf of authorized Grid users.

- *Security.* Grid utilizes SSL protocol for secure communication. Hence, accommodation of this protocol might be required for communication of components of a distributed Grid application.

7.4 Related Projects on Grid

In this section we present an overview of several database and stream processing projects utilizing Grid.

7.4.1 OGSA-DAI

The purpose of OGSA-DAI initiative [62] is to provide generic Grid data services for access to and integration of data held in relational DBMSs and in XML repositories, called data resource. A Grid Database Service (GDS) wraps a data resource and is capable of evaluating a collection of linked activities. The activities are one of the following kinds: database query, result transformation or result delivery. Thus, the project provides coarse-grained database interfaces allowing for grouping database access, data transformations and movement in a single request to the GDS.

OGSA-DAI addresses the problem for accessing stored collections of structured and semi-structured data, while GSDM deals with the problem for online processing of streaming data sources under Grid.

7.4.2 OGSA-DQP

OGSA-DQP [4] is a service-based distributed query processor for the Grid. It allows for declarative specification of complex applications accessing distributed data sources and computational resources in a uniform way. Thus, OGSA-DQP provides functionality for declarative service orchestration complementary to work-flow systems. The system relies on OGSA services to dynamically obtain the resources necessary for distributed query evaluation. OGSA-DQP adapts techniques from parallel databases to provide implicit parallelism for computational tasks in data-intensive requests.

The architecture includes two services: Grid Distributed Query Service (GDQS) and Grid Query Evaluation Service (GQES) whose functionality resembles GSDM coordinator and working nodes. The GDQS compiles, optimizes, partitions, and schedules distributed execution plans over multiple execution nodes. It interacts with the appropriate grid registries to obtain metadata about the data sources and computational resources. When a distributed query is posted to the system, the query evaluation system is dynamically composed from a number of GQES services. They are the execution nodes in the distributed query plan.

The OGSA-DQP is oriented for application that query high-volume stored data in science and medicine. The query processing utilizes iterative model combining push and pull-based interface between query execution modules. By contrast, we target on-line stream processing applications and utilize data-driven (i.e. push-based) data flow processing model.

OGSA-DQP re-uses the optimization, partitioning, and scheduling capabilities of Polar* engine [77] and re-implements the query execution engine in a service setting. It uses OGSA Grid Data Service that ensures that metadata and data are accessed via a standard well-defined interface. It also uses OGSA services for data transport and authentication.

Since the current Grid middleware does not support true application staging, OGSA-DQP assumes pre-installation of services software on the required Grid nodes. The service-based system architecture is used to guarantee timely acquisition of resources for the query execution, i.e. GDQS service providers either have themselves enough resources or have service level agreements with other GDS providers and computational resource providers that host query evaluation services. Therefore, the functionality of OGSA-DQP requires QoS guarantees for the participating in the distributed plans Grid data services and web services. Computational resource discovery is not automatically provided but partially supported via querying indexing services about the set of resources specified by the user.

GDQS coordinates between the query optimizer and GQES instances executing the distributed plan similarly to the GSDM coordinator controlling the working nodes. In both cases a distributed query evaluation system is constructed on the fly. However, OGSA-DQP has loosely-coupled service-based architecture, while GSDM is a tightly-coupled system. One of the places where this difference is noticeable is the communication between the system components. One of the reported OGSA-DQP performance bottlenecks is the lack of an efficient data transfer mechanism. Currently the data transport service provides for transfer of XML documents over SOAP/HTTP. High-volume streams of scientific data require efficient communication. Having the inter-GSDM communication under our control allowed to optimize it to an acceptable level where the communication is not a bottleneck.

Resource selection and scheduling algorithm that enables partitioned (intra-operator) parallelism for distributed query execution in a Grid environment is presented in [38]. The algorithm starts with a valid query plan with minimum partitioned parallelism and in each step finds the most costly operator for which there are available machines, increases the parallelism by one degree for this operator by allocating an available machine, and assesses the performance using a well-defined cost model. The algorithm stops when performance improvement is bellow a certain threshold.

We use a similar idea to create parallel plans with different degrees of par-

allelism. However, we utilize training mode to find an optimized data flow since we cannot rely on precise cost model for an extensible system with user-defined functions running in a heterogeneous environment.

The OGSA-DQP optimizer achieves partitioned parallelism using the exchange operator with pre-defined partitioning method. By contrast, we provide customizable partitioning and also a user-defined partitioning, including window split as a form of intra-object partitioning for intra-function parallelism.

Recent work [37] addresses the need for generic adaptivity framework for Grid query processing and proposes plan adaptation by changing the degree of parallelism and the distribution vector to balance the load among heterogeneous resources.

7.4.3 GATES

GATES (Grid-based AdapTive Execution on Streams) [23, 24] is a Grid middleware system aimed for distributed processing of high-volume data streams. It is designed to use the existing Grid standards and tools to the extent possible and is build on OGSA standard and Globus Toolkit 3.0 middleware. The system provides users with a high-level interface to specify algorithms for data stream processing. Another important aspect of the system is its ability to adapt the processing to achieve the best possible accuracy while maintaining the real-time constraint on the analysis.

The application developer divides the application into stages, implements the processing in each stage, and chooses one or more adjustment parameters. The adjustment parameters provide adaptivity, e.g. allow to increase the processing rate in order to achieve real-time processing at the price of reduced accuracy. The application starts using an XML configuration file that describes the application stages and their locality. The execution is performed through a number of communicating GATES Grid services that are able to contain and execute user-specified code and implement a self-adaptation algorithm.

The GATES Grid services resemble the GSDM working nodes in the sense that both are distributed execution components for stream processing started on-demand and customizable by uploading of user-defined code and SQFs. The difference is in the level of abstraction and flexibility provided. GATES application stages are fixed and pre-defined pieces of code created by the application developer and stored in a configuration file. In GSDM the user specifies continuous queries in terms of declarative SQFs and the coordinator creates dynamically a distributed execution plan.

The GATES system is designed for applications with expected high-volume communication and relatively cheap computations. Hence, the work focuses on a resource allocation algorithm minimizing the communication costs be-

tween the nodes, while parallel processing of expensive operators on a single stream is not considered at all. The distributed pattern of execution is limited to multiple input streams of the same type that undergo the same chain of operations, and are merged in the later stages of the application. We provide high-level and general framework for specifying distributed patterns through the data flow distribution templates.

Similarly to other Grid applications, GATES project intends to use the information services provided by Grid middleware for resource discovery. However, the problem how to simultaneously and with short start-up time allocate multiple Grid resources needed for distributed stream processing is not addressed.

7.4.4 R-GMA

R-GMA (Relational Grid Monitoring Architecture) [66] is a data integration system for Grid monitoring. It allows for publication of both static and dynamic stream data as well as specifying of continuous, history and latest-state queries. The system utilizes "local as view" approach for data integration through a global relational schema, providing a global view over different Grid resources, and local data sources whose schemas are views defined over the global schema. The requests for monitoring information are specified in SQL.

Monitoring information about dynamic characteristics of Grid resources is provided as data streams from information providers to the information consumers. Stream republishers aggregate streams from other providers using specifications in SQL. By constructing a hierarchy of stream republishers and using a DBMS for the latest stream values, the system allows more complex processing including joining and aggregating of latest stream values.

The design of the system is oriented very much to providing scalability, performance, and integration to handle the large amounts highly-distributed Grid monitoring data. However, the volume of the stream data is much lower than the volume of a scientific instrument stream, e.g. one data sample is generated every 30 sec, and consequently the problem how to distribute a high-volume stream for efficient parallel processing is not relevant in this context.

8. Related work

This chapter presents an overview of research projects related to the GSDM system. GSDM is a prototype of data stream management system and thus we present other DSMSs related to GSDM with respect to stream data modeling, query languages for continuous queries, processors for continuous queries, distributed processing of streams, and data stream partitioning strategies. In the first section we describe the related DSMS projects and how GSDM differs from them.

The next sections present other technology related to GSDM, namely, continuous query systems operating on stored data, parallel DBMSs providing scalable processing of non-stream data, and DBMSs used for management and analysis of scientific data.

8.1 Data Stream Management Systems

During the last years data stream processing has been an active area of research for the database community and many projects and prototypes of DSMSs have been developed and published in the literature. The earlier systems typically have central architecture where continuous queries are processed on a central server over possibly distributed stream sources. The DSMSs with central architecture are related to GSDM with respect to data modeling, query languages, and query processors for continuous queries.

Over the last three years many DSMSs were augmented with distributed and parallel processing capabilities. This capabilities are more closely related to GSDM with respect to problems such as data partitioning, distributing the continuous query processing among multiple nodes, and load balancing. When we describe such systems we start first with central architecture before presenting the distributed and parallel extensions.

Most of the stream processing systems [17, 27, 57, 59, 81], including distributed ones, have fine granularity of stream data items and relatively small cost of the stream operators per item. Hence, the distributed extensions often address the problem of an optimal operator distribution that achieves good load balancing among the machines. In contrast, the streams in the scientific applications we address have big total volume and data item size, and the user-defined functions are computationally expensive. Therefore, we address the

problem for scalable execution of expensive stream operators (SQFs) through parameterizable templates for partitioned parallelism.

8.1.1 Aurora

Aurora [17, 2, 14] is a stream processing engine with central architecture developed at Brown, Brandies, and MIT. It is a data-flow system where queries are composed utilizing a boxes and arrows paradigm from process flow and work flow systems. Data sources, such as programs or hardware sensors, generate streams that are collections of data values with fixed schema containing standard data types. The output streams are presented to applications, which are designed to deal with asynchronous data delivery.

The Aurora model is based on an extension of the relational model where stream data are of standard relational atomic data types. The cost of the operators is relatively small, so that the system has to schedule efficiently centralized processing of units with fine granularity. In contrast, we address parallel processing of computationally expensive stream operators utilizing user-defined partitioning of streams that may contain data of complex user-defined types.

A distinguishing feature of Aurora is the quality of service (QoS) support that is an integrated part of the system design. A number of convex QoS graphs can be provided by an application administrator for each result stream. The graphs specify utility of the result in terms of performance or quality metrics such as delay, percentage of dropped tuples, or result values. Scheduling algorithms utilize the QoS graphs and aim at optimizing the overall utility of the system.

Aurora's continuous queries are specified in a procedural way using a GUI. It is possible to combine multiple continuous queries, eventually for different applications, into one so-called *query network*. Thus, shared processing of CQs is supported given a specification by the application administrators.

No arrival order is assumed in Aurora's data model. Hence, order-sensitive operators are designed to handle this by either ignoring tuples out of order or introducing a slack specification, where the slack is a fixed number of tolerated out-of-order tuples. By tolerating partial disorder the system can give processing priority to tuples that contribute higher QoS utility.

The last overview of the project in [14] indicates the need for supporting different feed formats for input streams. The suggested design solution is to provide special input and output converter boxes that are dynamically linked into the Aurora process. These boxes are similar in functionality to the GSDM stream interfaces but we go further by also encapsulating in them network communication of streams. Other lessons from the Aurora experience are the need for global accessibility to the meta-data and a programmatic interface, al-

lowing to script the query network. These features are available in GSDM. For example, the coordinator creates dynamically the installation scripts to be sent to the working nodes in order to install the distributed query execution plan.

8.1.2 Aurora*, Medusa, and Borealis

Two proposals to extend the Aurora stream processing engine for a distributed environment were presented in [25], followed by the work on Borealis [1] as a second generation stream processing engine.

In Aurora* multiple single-node Aurora servers belonging to the same administrative domain cooperate to run an Aurora query network. Medusa is a distributed infrastructure for service delivery among autonomous participants where participant collaborations are regulated using economic principles, e.g., pair-wise contracts to specify a compensation for each service.

Scalable communication infrastructure is proposed using mechanisms for global name space and discovery, routing, and message transport. To provide scalable inter-node communication, streams are multiplexed to a single TCP connection. A message scheduler implements a policy for connection sharing based on QoS specifications. By contrast, in GSDM we use one TCP connection to implement an inter-GSDM stream. This choice is justified by the characteristics of the GSDM applications: high volume of data in a single stream and the coarse granularity of the expensive user-defined functions.

The Aurora* proposal includes dynamic adjustment of processing allocation among the participating nodes. Transformations of the query network are based on a stop-drain-restart model. Two basic mechanisms for load sharing among the nodes are proposed: *box sliding* and *box splitting*. Box sliding allows boxes on the edge of a sub-network on one machine to be moved to the neighbor, thus reducing the load and possibly the communication.

Box splitting creates a copy of a box intended to run on another machine. In order for a box to be split it must be preceded by a *filter* box with a predicate that partitions the input stream and be followed by one or more *merging* boxes. The merging boxes depend on the predicate in the filter box as well as on the semantics of the box to be split.

The proposed concept of box splitting has some similarities to our template for partitioned parallel execution of SQFs. The Aurora* authors point out the challenges related with the choice of a filter predicate for stream partitioning and the determination of appropriate merging boxes. However, the work does not address the problem of how to automatically create filtering and merging boxes for a given box splitting, which we provide through customizable templates. The ideas in [25] are presented at a proposal level and neither an implementation nor experimental results on box splitting are reported in the follow-up literature.

Borealis [1] is a proposal for a second generation stream processing engine that inherits core stream functionality from Aurora and distribution functionality from Medusa. It extends Aurora with support for dynamic revision of query results, dynamic query modifications, and a scalable multi-level optimization framework that strives to incorporate sensor networks with stream processing servers.

Recent work focus on two problems of distributed stream processing: load balancing [88] and fault-tolerance [15]. Since the target applications involve big numbers of relatively cheap stream operators, the load balancing problem consists of good distribution of operators among the nodes. The initial distribution of a query network utilizes localities of stored data and statistics obtained through trial runs. Further re-distribution is achieved dynamically through box sliding and correlation-based peer-wise or global re-distribution algorithms [88].

Several strategies for providing fault tolerance are presented in [15]. Besides adapting the standard active and passive stand-by approaches to the stream processing context, in upstream backup each server acts effectively as a back-up server for its downstream servers. The fault tolerance problems are not addressed currently in GSDM; this research is complementary to our work and its results can be utilized in a future work.

8.1.3 Telegraph and TelegraphCQ

The goal of the Telegraph project at UC Berkeley is the development of an adaptive data flow architecture. The Telegraph architecture includes three types of modules: query processing modules which are pipelined non-blocking versions of the standard relational algebra operators such as select, join, group etc.; adaptive routing modules, such as *eddy* [8], which are able to re-optimize the query plan while a query is running; and ingress and caching modules which are wrappers providing interface to external data sources. All the modules communicate through an inter-module communication API called *fjords*.

Two prototypes, CACQ [57] and PSoup [20], extend Telegraph with capabilities for shared processing over streams. In CACQ, standing for continuously adaptive continuous queries, an eddy can execute a "super"-query corresponding to the disjunction of all the individual continuous queries. Each tuple maintains an extra-state that serves to determine which queries should obtain a result tuple. In order to optimize selections for shared execution, a grouped filter operator is introduced that indexes predicates over the same attribute. PSoup extends the mechanisms developed in CACQ by allowing queries to access historical data and supporting disconnected operation where users can register queries and return intermittently to retrieve the latest results.

The goal of TelegraphCQ [19, 49] is shared, continuous data flow process-

ing with emphasis on adaptability. It is a result of redesign and re-implementation of Telegraph based on PostgreSQL that also uses the experiences from CACQ and PSoup.

As a part of TelegraphCQ a *flux* operator [75, 74] has been designed to provide partitioned parallelism, adaptive load-balancing, high availability, and fault tolerance. The first version of *flux* [75] provides adaptive partitioning on the fly for optimal load balancing of parallel CQ processing. General partitioning strategies, such as hash partitioning, are encapsulated in the *flux* operator. We also have a customized general partitioning and in addition handle operator-dependent window split strategies customizable with user-defined partitioning for scalable execution of expensive stream operators.

The main advantage of *flux* is the adaptivity allowing for data re-partitioning. 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 with user-defined partitioning and window distribute with Round Robin), 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* [74] encapsulates fault-tolerance logic that allows for constructing highly-available parallel data flows. The techniques involves replicated computations and mechanisms for restoring failured operators states and lost in-flight data. This work is complementary to the problems of user-defined stream partitioning presented here.

Queries in TelegraphCQ can be specified on both static and streamed data. For each stream there is a user-defined wrapper consisting of *init*, *next*, and *done* functions that are registered to the system. GSDM stream interfaces provide similar functionality. However, by utilizing object-relational modeling we put stream sources in a type hierarchy and associate the stream interfaces with stream types rather than with individual stream source. Thus, we allow to have more than one stream source using the same stream interface. Furthermore, we provide for multiple interfaces for the same stream type, so that data can be fed into the system using different communication media.

Modules in Telegraph communicate through the *fjords* API [56] that supports both push and pull connections and thereby is able to execute query plans over a combination of static and streaming data sources. In the current implementation GSDM does not provide pull-based communication between working nodes. However, stream query functions can access locally stored static data in a pull-based manner through the generic capabilities of the Amos II query processor.

8.1.4 CAPE

Continuous Adaptive Query Processing Engine, CAPE [70, 71], is a prototype system developed at Worcester Polytechnic Institute. D-CAPE [52] is a distributed stream processing framework based on CAPE and designed for shared-nothing architecture. The system is designed for highly dynamic stream environments and employs an optimization framework with heterogeneous-grained adaptivity. CAPE focuses on precise results computation by employing different optimizations and does not consider load shedding and approximation of results. CAPE utilizes *punctuations* [86] that are dynamic meta-data used to model static and dynamic constraints in the stream context. The punctuations can be exploited to reduce resource requirements and to improve the response time. Stream tuples and punctuations are assumed to be globally ordered on their timestamps recording their arrival time.

Fine-grained adaptivity is achieved by reactive query operators whose execution logic can react to the varying stream environment. An adaptive scheduling framework selects one algorithm from a pool of scheduling algorithms that best fits to the optimization goal defined as a quality of service specification combining multiple metrics. Online re-optimization and plan migration restructure the query plan at runtime including plans with stateful operators. An adaptive distribution framework allows to balance the workload among a cluster of machines and maximally exploit available CPU and memory resources. Adaptations at all levels are synchronized and invoked with different frequencies and under different conditions, where the adaption intervals increase from operator-level to the distributed processing level.

D-CAPE

In D-CAPE a number of CAPE engines perform distributed query processing and one or more *Distribution Manager* monitor the execution and initiate re-distribution when needed. Run-time statistics is periodically obtained and used to assess the processors' workload and to decide about re-allocation. A connection manager module communicates with the processors to establish operators to be executed. A *distribution decision maker* decides how to distribute the query plans using a repository of *distribution patterns*. Examples of distribution patterns are *Round Robin*, which tries to fairly assign equal number of operators to each processor, and *grouping distribution*, which tries to minimize network connections by keeping adjacent operators on the same processor.

D-CAPE monitors query performance and redistributes operators at runtime across a cluster of processors. The redistribution tries to alleviate the most-loaded machines. The algorithm that picks operators to be moved gives preferences to operators that would remove network connections in the overall distribution.

The components of the distributed GSDM architecture resembles those in the D-CAPE architecture. The GSDM coordinator functionality is performed by the CAPE distribution manager in sense of generating distributed plans, installing plans on the query processing engines, and monitoring the execution. D-CAPE's repository of distribution patterns is similar to the GSDM's library of distribution templates, which in both projects guide the generation of distributed plans.

D-CAPE's distribution patterns allow for various types of parallelism on inter-query, intra-query, and intra-operator level. Partitioned parallelism, as used in *flux* [75] and Volcano [39], is applied to query operators with large states accumulated at run time, such as multi-way joins. By contrast, we focus on partitioned parallelism for computationally expensive user-defined operators (SQFs). Our generic distribution template for partitioned parallelism is parameterizable with a user-defined partitioning strategy providing for intra-object parallelism of user-defined operators.

As in D-CAPE the GSDM architecture allows for re-optimizing parallel plans, though this functionality is not implemented in the current prototype. The statistics collector at the coordinator periodically gathers information about the cost of SQFs and communication at working nodes that allows to assess the workload.

The redistribution opportunities for plans with partitioned parallel execution of expensive operators in GSDM are somewhat limited in comparison to a general distribution framework. For example, the operator-level redistribution [52] assumes that an operator is small enough to fit on one machine. Hence, our vision about the changes that are appropriate in GSDM's partitioned parallel plans includes replacement of the partitioning strategy or increase of the degree of parallelism given an ability for additional resource allocation on-demand. The problems related with the dynamic re-optimization of parallel partitioned plans are subject of future work.

8.1.5 Distributed Eddies

The work on distributed eddies [85] puts adaptive stream processing in a distributed environment. An *eddy* [8] is a tuple router at the center of a data flow that intercepts all incoming and outgoing tuples between operators in the query plan. Eddies collect execution statistics used when the routing decisions are made. With distributed eddies each operator in the distributed plan is augmented with eddy's functionality, i.e. makes routing decisions and collects and exchanges statistics with other operators. An analytical model for a distributed query plan is constructed using a queuing network. Two performance metrics are defined, the average response time (latency) and maximum data

rate. An optimal routing using the queuing network model is computed and six practical routing policies are designed and evaluated through simulation.

The ideas of box splitting and sliding suggested in Aurora* are used to dynamically reallocate resources among operators so that more expensive operators can get more resources when needed. As we discuss, box splitting is a form of parallel processing where data partitioning among the operator instances is done as a part of the applied tuple routing policy. The policies are implemented as weighted distribution vectors. Such parallel processing is similar to our window distribute, but we customize explicitly the data partitioning strategy. Furthermore, we also provide order preservation of the result stream while distributed eddies do not guarantee that the result tuples would be ordered at the receiving sink. Finally, the GSDM window split partitioning of big stream data items does not have analogue.

Future work will investigate the application of the analytical queuing network model for generation and optimization of parallel execution plans in GSDM. For example, one possibility is to use the model to compute the expected latency or throughput with different partitioning strategies. However, several limitations restrain a direct application of the model to parallel execution plans in GSDM. For example, the assumption about well-known costs of the operators in the plan does not hold in an extensible system, such as GSDM, where the costs of user-defined functions over user-defined data types might be hard to define or obtain from the authors of the code. Hence, we need some form of test runs of plans for statistics collection purposes.

8.1.6 Tribeca

Tribeca [81] is an extensible stream database system designed to support network traffic analysis. The system uses a data flow query language where users can explicitly specify how the data flows from one operator to another. The Tribeca queries have a single source stream and one or more result streams. Hence, Tribeca is one of the first systems that practically support shared execution of analyses over the same input stream. The operators include qualifications (filters), projections, aggregates, demultiplexing (*demux*) and remultiplexing (*mux*). Demultiplexing partitions a stream into sub-streams based on the data content similarly to `GROUP BY` clause in SQL. Remultiplexing is used to combine the logical sub-streams produced by demux or unrelated streams of the same data type similarly to union operator in the relational algebra. It is not reported whether mux takes care to preserve the ordering of elements. Tribeca also supports windows on streams and a limited form of join.

Tribeca queries are compiled and optimized using many of the traditional

relational optimizations. The queries have pipelined execution and intermediate results are never unnecessarily materialized.

In GSDM stream partitioning and combining SQFs in the window distribute strategy are similar to Tribeca's *demux* and *mux* operators. However, Tribeca's partitioning is based only on data content and performed for aggregation purposes in a centralized architecture, while in GSDM user-defined partitioning is used for parallelization. Tribeca has a central architecture and queries are limited to run over a single data stream. GSDM uses a distributed architecture for parallel execution and allows for specifying data flow graphs with multiple input streams.

8.1.7 STREAM

STREAM [10, 59, 6] is a general-purpose prototype of relational-based data stream management system developed at Stanford University. The project proposes a declarative query language for continuous queries and focuses on problems such as adaptivity, approximation, and scheduling in a central processing architecture.

In [6] an abstract semantics for continuous queries is defined and implemented in CQL, a declarative query language extending SQL with window specifications from SQL-99. Queries can be specified on both streams and relations defined using a discrete, ordered time domain. The declarative continuous queries are compiled into a query plan composed of operators, queues buffering tuples between operators, and *synopses* that store the operator state. The operators belong to one of the classes relation-to-relation, stream-to-relation, or relation-to-stream. Stream-to-relation operators are based on the concept of a sliding window over a stream and expressed using a window specification, such as `[Rows n]` for count-based windows, and `[Range t]` for time-based windows.

The system has an adaptive query processing infrastructure that includes algorithms for adaptive ordering of pipelined filters and pipelined multiway stream joins [12]. In the area of operator scheduling STREAM uses a *chain scheduling* algorithm [9] to minimize runtime memory usage. When the load exceed the available system resources, STREAM provides approximate answers of continuous queries [59]. If the CPU time is not sufficient, sampling operators are introduced in the query plan that probabilistically drop elements and thus save CPU time. In the case of limited memory the approximation can be achieved by reducing the size of synopses, maintaining a synopsis sample, using histograms or wavelets, etc.

If the resources of a central system are not sufficient, STREAM addresses the problem by approximate query answering. In contrast, in GSDM we consider instead parallel execution of expensive stream operators. We do not ad-

dress problems of scheduling of many cheap operators or adaptive reordering of query plans and, thus, the work on STREAM is complementary to ours.

The problems of time management in data stream systems are addressed in [79]. The query processor of a DSMS usually needs to process elements in increasing time stamp order to provide semantic correctness of continuous queries. However, when time stamps are put at multiple distributed stream sources, the arrival order of the elements at the DSMS may differ from the increasing application time stamp order. The problem of synchronization of unordered distributed streams is solved by an input manager that buffers out-of-order stream elements and presents them ordered to the query processor through a mechanism, called *heartbeats*. The heartbeats are as a special kind of punctuations that guarantee that all the future elements on streams will have timestamps bigger than the heartbeat. The proposed algorithm for heartbeats generation models factors influencing the synchronization of distributed streams, such as application time skew and network latency and can be applied in future work in GSDM for synchronizing distributed application streams.

8.1.8 Gigascope

Gigascope [27] is a high-speed stream database specialized for IP network monitoring applications. The Gigascope query language, GSQL, provides tools to define declarative SQL-like queries and to compose them into a more complex processing chain. It is a pure stream query language, i.e. all inputs and the output of a GSQL query are data streams. The basic supported stream operators are selection, join, aggregation, and stream merge. Gigascope is an extensible systems where user-defined functions can be registered in a function registry and called in the queries. In this way Gigascope supports high-performance specialized algorithms for network analysis. Similarly we support user-defined scientific operations, e.g. FFT, for the space physics stream applications.

Gigascope has a two-level query architecture optimized for the network analysis applications. Low level queries (LLQs) access *source streams* and are intended for data reduction. High level queries (HLQ) perform more complex processing. When a GSQL query is defined over source streams, Gigascope creates one LLQ for every source stream and transforms the original query into a HLQ executing over the output streams of LLQs.

Furthermore, GSQL allows the user to associate a name with a query, which is used to access the query result stream from an application or an another GSQL query. In this way GSQL queries can compose more complex processing chains. In terms of GSDM, the named GSQL queries correspond to defined SQFs without parameters. In GSQL the name of the query identifies the result stream, but in order to specify a similar query over another source

stream, the user must define and name a separate query. The LLQs allow non-stream parameters to be defined and bound during the query instantiation.

By contrast, SQFs in GSDM are parameterized on the input streams and eventually other parameters. This allows compact parameterized specification of complex distributed processing, e.g. an SQF that is a template parameter is instantiated automatically by the template constructor, which may include multiple instantiations on different input streams. Due to the parameterization, the name of an SQF cannot identify the result stream, but instead system generated stream identifiers are used.

The GSQL processor generates C and C++ code implementing the queries. LLQs are linked into the stream manager, while HLQs run as separate processes and communicate data through shared memory. Depending on the capabilities of the computer's network interface card (NIC), some LLQs in Gigascope can execute inside the NIC. In contrast, the query executor in GSDM runs SQFs always inside of a working node following a scheduling policy.

Even though the query execution in Gigascope is composed by a stream manager and a number of high level query processes, the architecture is designed to run on a dual CPU server with shared memory communication between processes. By contrast, we address distributed and parallel execution with potentially high degree of parallelism in a shared-nothing architecture.

In order to bound the state of the stateful operators, such as join and aggregation, Gigascope uses ordering properties of the streams and predicates in the query specification instead of explicit definition of sliding windows commonly used in other projects. For example a join window is determined by a join predicate on the ordering attributes on each of the joined streams. This approach fits better the specifics of network data streams that often contain several timestamps and sequence numbers.

The merge operator in Gigascope is similar to our *S-Merge* operator. In order to handle the danger of blocking in the absence of tuples on some of the input streams, Gigascope injects ordering update tokens into the data stream. Instead, we use a time-out parameter of *S-Merge*.

8.1.9 StreamGlobe

StreamGlobe [51] is a prototype for distributed stream processing in a peer-to-peer infrastructure developed at Munich Technical University. XML data streams are queried through a windowed extention of XQuery supporting currently selection, projection, aggregation, and user-defined operations. The main focus is multi-subscription optimization where queries over the same stream source and with overlapping predicates share computation by subsumption. As a result of sharing the network traffic and computational load is reduced.

The system architecture is implemented based on OGSA where components are collaborating Grid services. The services are available according to the capabilities of the peers.

Continuous queries are distributedly executed given that the result streams of other queries can be used. The queries without overlap with other queries are executed in a centralized manner. By contrast, GSDM focuses on parallel execution of queries with expensive user-defined functions, which is not considered in StreamGlobe.

8.1.10 Sensor Networks

A number of DSMSs are oriented to process streams generated by sensor networks [16, 56, 28]. Typically, a large number of small-scale sensors, connected through a wireless communication interface, are sensing and transmitting relatively simple data units, such as temperature or pressure measurements. Since the conservation of battery power is a major concern, work in [89] proposes efficient in-network query processing capable for trading-off costly communication for cheap local computations. By contrast, scientific instruments in the GSDM target applications generate enormous amount of numerical data and we focus on scalable computations through parallelism.

Nile [41] is a prototype of a stream processing engine developed at Purdue University. It extends an object-relational DBMS, Predator, with capabilities to process continuous queries over data streams. Similarly to GSDM, stream data sources are modeled by a stream type and data are retrieved through stream type interfaces. Queries are specified in an extension of SQL. The work focuses on providing windowed operators and in particular on the in-ordered execution of time-based window joins. In contrast to GSDM, the query processing is performed on a centralized architecture.

More recent work on Nile-PDT [3] introduces an abstract data type to represent multiple streams in sensor networks. A specialized operator *Sensor Network Join (SN-join)* produces pairs of matching sensor streams based on some similarity metric. *SN-scan* operator attaches the sensor network platform to the query processing system and provides load shedding by more rare processing of sensors that do not contribute to the final results of the query oriented to phenomena detection. The operators provide relevance feedback to their predecessors that guides the predecessors in giving preferences to some of the input streams. In this way an utility-based data shedding is used to handle overload situations for centralized processing of sensor network streams. By contrast, we focus on scalable processing through parallelism and do not address data shedding.

8.2 Continuous Query Systems

The concept of continuous queries was first introduced in [84] to name queries that are issued once and run continuously. In this work an incremental evaluation approach is used to avoid repetitive computations over append-only data sources.

NiagaraCQ [22], developed at University of Wisconsin, supports scalable continuous query processing over multiple, distributed XML files. It utilizes grouping of queries based on their similarities, which leads to several benefits, such as shared computations and reduced number of query invocations. The group optimization is incremental, i.e. new queries are added to existing query groups without having to regroup already installed queries. Incremental evaluation of CQs is provided by considering only the changed portion of the updated XML files.

Continuous query processing of stored data in NiagaraCQ is triggered using an event detector that detects timer events for timer-based queries and changes of files for change-based queries. By contrast, on-line stream processing systems, such as GSDM, typically require an active scheduling mechanism that schedules continuous queries whenever new data has come and there are available processing resources, rather than waiting for a triggering event.

NiagaraCQ uses a centralized architecture for shared processing of relatively simple continuous queries over distributed XML files. In contrast GSDM has a distributed architecture to achieve scalable on-line stream processing involving expensive SQFs.

8.3 Database Technology for Scientific Applications

Scientific applications are characterized by high volumes of data with typically complex structure and non-trivial operations. Hence, they put requirements for database technology that are not met well by the main-stream relational databases oriented to business application. Therefore, scientific applications were one of the major driving forces for the development of object-oriented and object-relational DBMSs. The current ORDBMS allow for better representation of scientific data by ability to extend the type system with new base types and complex user-defined types together with user-defined operations over them.

The work in [87] proposes the idea to extend the concept of a database query with numerical computations over scientific data, and the query optimizer with transformation and implementation rules applicable to these operations. The focal point are the advantages that the algebraic database query optimization can provide to the scientific computations. In contrast to our work, the paper

does not consider on-line stream processing and assumes centralized architecture.

The work points out important observations concerning the requirements of scientific applications for database query processing. For instance, more diverse implementation techniques are available for scientific computations optimization criteria must be extended with numerical accuracy and stability. In scientific computations CPU costs are much more varied than in relational queries and often dominate the cost of specific operators (opposite to I/O cost in traditional RDBMS). Therefore, the extensibility of the system is very important so that the optimizer can utilize user-defined transformation rules and cost models. We go a step further by providing a generic mechanism to extend the system with user-defined partitioning utilizing semantics of scientific computations for scalable parallel execution.

8.4 Parallel DBMS

Data partitioning strategies for parallel databases [64] are well investigated for relational databases. Strategies, such as Round Robin and hash partitioning can be used as parameters of our window distribute strategy. What makes our stream partitioning strategies different is that the processing must preserve 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 [39] *support functions* parameterizing the *exchange* operator. In contrast, we have pairs of partition and combine operators where the combine operator preserves the stream order. While window distribute parameterized by, e.g., Round Robin is similar to the *exchange* operator, *window split* is novel.

RiverDQ [7] proposes a content-insensitive partitioning for load balancing in heterogeneous execution environment. Currently, we focus on homogeneous environments. Future work will investigate the possibilities to apply RiverDQ ideas in order to use GSDM on heterogeneous environments.

Data partitioning problems have been addressed in object-relational DBMSs with the purpose to achieve efficient parallel execution of UDFs while preserving their semantics [46, 60]. The work in [60] proposes a specification of UDFs that allows generic parallelization and classifies the partitioning strategies for user-defined operations. It presents the generic pattern of partition, compute, and combine phases for UDFs. However, the idea to specify generic and modular data flow distribution patterns through templates is to the best of our knowledge unique.

Even though the work in [60] is based upon the stream processing paradigm

for query execution, it differs from our work in the assumption that the data are stored on disk and have limited size. This assumption allows for streaming the data from the disk in different order as appropriate for the operations performed. In contrast, the on-line stream processing, including a parallel one, must conform to the ordering of the stream and cannot afford re-ordering for the operator purposes since it is a blocking operation. Our window split for parallel execution of a UDF over a single logical window does not have analogue in [60].

9. Conclusions and Future Work

In this Thesis we presented the design, implementation, and evaluation of the GSDM prototype of an object-relational data stream management system for scientific applications.

The system provides users-scientist with tools to express complex analyses of streams, generated by instruments and simulators, as continuous queries composed of user-defined operations. A distributed and parallel architecture provides for scalable execution of continuous queries with computationally expensive operations over high volume streams.

Using a generic framework the users can specify scalable parameterized distributed execution strategies by the means of data flow distribution templates. Several common distribution patterns are available through a built-in library of templates. Using the library we define a generic template for partitioned parallelism, PCC.

We defined two overall parameterizable parallel strategies: the so-called *window split* that divides a stream data items (logical window) into sub-windows to be processed in parallel by the partitions, and *window distribute* that distributes several logical windows among parallel partitions. Window split provides for intra-object parallel execution of user-defined functions, while window distribute provides for inter-object parallelism. Both overall strategies are customizable with different partitioning methods in order to instantiate a particular stream partitioning strategy. Through the customization window split has knowledge about the semantics of a user-defined function to be parallelized for the purposes of more efficient execution. Our experiments show that window split with user-defined partitioning of windows utilizing the semantics of the functions can achieve higher total throughput of the system in scenarios when expensive operations are executed on limited computational resources.

Although the problem for parallelization of user-defined functions has been addressed in general in the literature about object-relational DBMSs, we are the first to provide a generic mechanism for implementation of such parallel strategies, as well as an experimental evaluation investigating the trade-offs of the two strategies.

We developed a basic optimization framework to provide parallel transparency to the user. Utilizing meta-data about valid partitioning strategies for

user-defined functions, the continuous query optimizer enumerates parallel plans and selects an optimized one using statistics collected from trial executions.

We investigated the requirements for implementation of GSDM in a Grid-based environment.

GSDM is the first reported fully functional prototype for parallel processing of continuous queries. Leveraging upon an object-relational model, we model numerical data from scientific instruments as user-defined types and implement operations over them as user-defined functions. Types of stream data sources are organized in a hierarchy and inherit properties form a generic stream type. As part of the prototype implementation many software modules have been designed, implemented, evaluated, and improved to a level of functionality and performance that is acceptable with respect to the overall system performance and functionality. Among these are the continuous query engine based on data-driven data flow paradigm, compiler of high-level CQ specifications into distributed execution plans, stream interfaces, inter-GSDM communication primitives, statistics collector for monitoring the execution, and protocols for installation, activation and termination of the CQ execution in distributed environment. Our experiments with real scientific streams on a shared-nothing architecture show that in order for a distributed stream processing to be efficient, not only stream operators and scheduling inside of a processing node, but also stream communication between nodes, need to be carefully designed and implemented.

Although in the work on the GSDM prototype we focus on the specific needs of scientific applications, the system can be extended for other applications with expensive operations over streams with complex and high volume content. For example, analysis of MPEG streams can be specified as CQ given interface implementation for this type of streams and operation implementation to be plugged into the system.

The work on the GSDM prototype opens a number of interesting directions for future work. In the following we will enumerate some of them.

Shared execution of continuous queries

A number of work on continuous queries emphasize the need for shared execution of long-running CQ for the purposes of scalability and overall cost-efficiency of the systems. The proposed solutions [57, 22] utilize similarities in queries specifications and create shared execution plans by grouping predicates on a common attribute and evaluating simultaneously the predicates in such groups. In the presence of expensive user-defined functions we can expect even bigger benefit from shared execution of similar queries for the overall system performance. It should be investigated how to share plans that are graphs of user-defined functions.

Adaptive CQ Processing

Adaptive query processing (AQP) [42, 11] interleaves query optimization and execution, possibly multiple times over the running time of the query, in order to adapt the processing to the changing execution conditions.

Long-running queries often experience during their life time changes in the execution environment and characteristics of input streams. Hence, adaptive execution of CQs is an important desirable property of CQ processing. A variety of adaptation techniques appeared recently in the literature spanning from the operator-level adaptation, plan migration to another execution plan, changing the operator scheduling policy, or adapting distributed plans by re-distributing the load among the resources currently available [71, 12, 88].

Future work will investigate possibilities for adaptation of parallel execution plans, such as changing the degree of parallelism and replacing one partitioning strategy with another. Such adaptation can response to changes in the available resources assuming relatively stable rates of streams generated by scientific instruments.

Integration with Grid Infrastructure

The ability of computational grids to provide computational resources on-demand can be very beneficial for GSDM running expensive CQs with varying resource requirements. The current development of Grid middleware does not provide the resource allocation functionality as needed for the long-running parallel GSDM jobs with guaranteed start-up time. Future work on this problem depends on the future developments of the Grid middleware.

CQ Optimization

The current optimization framework uses exhaustive parallel plan enumeration for an expensive SQF. To avoid generation and evaluation of possibly very big spaces of data flow graphs, the functionality of the optimizer needs to be enhanced with heuristics such as random walk or binary search of the search space.

Furthermore, methods for optimizing of CQs composed out of several SQFs have to be developed and evaluated. For example, investigating when it is worth to encapsulate a pipeline of two SQFs into one large SQF to be executed in parallel and when it is better to parallelize each of them separately, possible by using different degrees of parallelism and partitioning strategies.

Finally, the optimality metric used to select the optimized plan can be further developed. The metric we used in the presented experiments estimates the total throughput of the system through the maximum utilization time of the nodes. In order to address the needs of applications with different quality of service requirements, we plan as a future work to evaluate the distributed

strategies by other metrics, such as latency and result precision, as well as to combine several metrics for multi-criteria optimization.

Summary in Swedish

Skalbar sökning av strömmande mätdata

Vetenskapliga instrument som satelliter, digitala antenner, digitala radioteleskop och simulatorer, genererar mycket stora volymer data var innehåll kan vara komplext. Dessa instrument producerar hela tiden data i form av ordnade och kontinuerliga sekvenser av dataelement, s.k. *dataströmmar*. För att kunna undersöka innehåll och upptäcka intressanta mönster i sådana dataströmmar behöver forskare utföra analyser av innehållet i dessa. Analyserna innefattar avancerade och dyrbara numeriska beräkningar. Dataströmmarna är i regel oändliga, och kan därför aldrig lagras i sin helhet. I stället görs bearbetningarna över ändliga delar av strömmarna som flyttas framåt hela tiden, vilka vi kallar *logiska fönster*. För att få hög precision i beräkningarna är det ofta önskvärt att ha så stora logiska fönster som möjligt.

Databashanterare (eng. Database Management Systems, DBMS) har under lång tid använts för hantering av stora mängder data. För att lätt och snabbt kunna hitta data i stora databaser tillhandahåller databashanteraren ett *frågespråk* vilket är ett högnivåspråk. Frågespråk tillåter användaren att lätt söka efter data i stora databaser utan att ange i detalj hur sökningen skall gå till. Emellertid är existerande databashanterare inte väl lämpade för de specifika krav som uppkommer vid bearbetning av frågor över strömmade data. Frågor över dataströmmar kallas *kontinuerliga frågor* eftersom de körs kontinuerligt över en tidsperiod under vilken de hela tiden returnerar nya frågeresultat allteftersom nya data anländer. Kontinuerliga frågor skiljer sig från vanliga databasfrågor där användaren skickar en fråga åt gången till databashanteraren som sedan omedelbart returnerar ett ändligt svar från varje begärd fråga innan nästa fråga bearbetas. När användare vill ha svar på en kontinuerlig fråga startas en resultatdataström som inte avslutas förrän användaren begärt att frågan skall stoppas.

I denna avhandling presenteras en ny ansats att utveckla databastekniker för tillämpningar som bearbetar stora dataströmmar innehållande dyrbara beräkningar. Vi har gjort detta genom konstruera ett utbyggbart system för generell hantering av stora dataströmmar. Vi kallar systemet GSDM (Grid Stream Data Manager). GSDM gör det lätt för användare att uttrycka och effektivt utföra omfattande vetenskapliga beräkningar med kontinuerliga frågor över strömmade data. I GSDM modelleras vetenskapliga data i termer av tillämpning-

sorienterade dataobjekt och funktioner över dessa objekt. GSDM är utformat som ett distribuerat system där olika delbearbetningar kan köras samtidigt på många olika datorer som kommunicerar med varandra via ett kommunikationsnätverk. Detta möjliggör skalbarhet för både stora datavolymer och komplicerade beräkningar.

GSDM ger användaren ett allmänt ramverk för att specificera strategier för parallell körning av kontinuerliga frågor över strömmar. En strategi uttrycker hur data och program delas upp mellan datorerna i det distribuerade systemet. Strategierna uttrycks som *dataflödesdistributionsmallar* eller bara *mallar*. Ett inbyggt bibliotek av mallar i GSDM tillhandahåller byggstenar för att bygga mer komplicerade distribuerade strategier. Vi definierar en generell mall för ett vanligt sätt att parallellt utföra dyrbara bearbetningar, kallad PCC (Partition-Compute-Combine). Med PCC definierar vi två olika strategier för att dela upp strömmar för skalbara, parallella och komplexa beräkningar på olika datorer i ett nätverk:

I den första strategin, som kallas *fönsteruppdelning*, kan användaren specificera hur fönstren i en ström skall delas upp i mindre fönster innan den parallella bearbetningen utförs på olika datorer. Beroende på den bearbetning man vill göra i en fråga kan det vara betydligt billigare att parallellt bearbeta mindre fönster än att göra motsvarade bearbetning över de stora originalfönstren. Hur man gör sådan uppdelning av strömmar beror på tillämpningen. Användaren tillhandahåller därför tillämpningsberoende funktioner för att dela upp stora fönster i mindre. I våra experiment har vi tillämpat fönsteruppdelning på distribuerad transformering av signaler med FFT (Fast Fourier Transform) och visat att fönsteruppdelning för denna tillämpning ger effektiva parallella beräkningar.

Den andra strategin, *fönsterdistribution*, sänder hela fönster till olika datorer för parallell bearbetning. I detta fall behövs ingen tillämpningsberoende uppdelning av fönstren utan hela fönster bearbetas parallellt av olika datorer. Fönsterdistribution påminner om strategier som används i distribuerade databaser. Utmärkande för fönsterdistribution i GSDM är att systemet måste ta hänsyn till att strömmarna är kontinuerliga sekvenser av obegränsad längd medan konventionella databaser arbetar med ändliga mängder där ordningen inte är viktig. Det finns flera sätt att distribuera fönster till olika datorer. I systemet kan egna distributionsfunktioner definieras.

Vi är först med att utveckla en generell och utbyggbar mekanism för skalbar exekvering av strömmade frågor. I avhandlingen undersöks för- och nackdelar med de olika strategierna för parallellisering. Det visar sig att den optimala strategin bl.a. beror av tillgängliga beräkningsresurser (dvs. antal datorer som deltar i beräkningen). Baserat på dessa resultat har vi utvecklat en optimerande mall. Med denna mall väljer systemet automatiskt vilken strategi som skall användas för parallellisering en kontinuerlig fråga och hur många datorer som

skall användas. Detta val sker genom att systematiskt variera strategierna och mäta vilken som är bäst för en given kontinuerlig fråga.

GSDM är ett fullt fungerande system för parallell bearbetning av kontinuerliga frågor över dataströmmar. GSDM innehåller följande komponenter: - En generell motor för utförande av kontinuerliga frågor baserade på mallar, - En kompilator för kontinuerliga frågor som genererar exekveringsplaner som tolkas och utförs av många kommunicerande motorer. - Programmerargränssnitt för start och stopp av kontinuerliga frågor. - Ett generellt gränssitt för att koppla upp olika sorters dataströmmar med systemet. - Övervakning av hur de kontinuerliga frågorna utförs. - Dynamisk uppstart och nedkoppling av GSDM noder.

Vi har gjort experiment med GSDM med verkliga vetenskapliga data från digitala radioteleskop. Experimenten gjordes i ett distribuerat gridbaserat datorkluster där alla noder arbetar oberoende av varandra och inte delar data. Våra experiment visar vikten av att ha en skalbar och distribuerad arkitektur med effektiv kommunikation mellan noderna för att hantera strömmar med stor datavolym och med avancerade beräkningar över stora fönster.

Acknowledgements

First and foremost I would like to thank my advisor Professor Tore Risch for giving me the opportunity for doctoral studies under his supervision. I am grateful to him for sharing his knowledge with me and being always eager to discuss new ideas and research problems. His enthusiastic attitude and high requirements have always stimulated me to learn more, to do better and *scalable*. I deeply appreciate his willingness to help and I am very grateful for his valuable suggestions and comments during the writing of the thesis.

I am in debt to my fellow graduate student Timour Katchaounov who introduced me to Tore while I was seeking for a PhD position. Timour was always ready to answer my questions and give an advice. Thanks to the current and former members of the UDBL lab, and especially to Ruslan Fomkin, Johan Petrini, and Erik Zeitler for helping and sharing with me difficulties and gladness.

I would like to thank our collaborators from the LOIS project Professor Bo Thidé, Walter Puccio, Roger Karlsson, Jan Bergman, Lars Daldorff, and the other members of the team, for providing scientific data and discussing with us interesting application problems.

My appreciation to Datalogi leaders, and to Marianne Ahrne and Anne-Marie Nilsson for their help with the administrative and financial issues. Special thanks to Maya Neytcheva for being both a friend and an advisor in the parallel algorithms issues. I am grateful to Ivan Christoff for being supportive and cheering me up. Kostis Sagonas has helped me a lot with his ability to point my attention to the right questions. My grateful thoughts to Anna Sandström, Elena Fersman, Pritha Mahata, Rafal Somla, Leonid Mokrushin, Pavel Krćal, and other colleagues at the IT Department for making the time of my studies more enjoyable.

I am grateful to Gunilla Klaar and Eva Enefjord who were very helpful with all the administrative and practical problems when I started my studies at the Department of Information Science. There I was lucky to meet and make friends with Brahim Hnich, Zeynep Kiziltan, and Monica Tavanti, who gave an international perspective to my view of life. We shared wonderful moments and I am grateful for their friendship and support.

I am grateful to my sister Pavlina, who embolden me to study computer science, and her family for their love and support. Special thanks to my friend

Zoya Dimitrova, who encouraged me to continue with studies for doctoral degree, and to my doctors for body and soul Mariana Angelcheva, and Radostina Miteva for their endless support, understanding, and encouragement. I am also grateful to Natalia Ivanova, Nasko and Olga Terzievi, Nina and Volodya Grantcharovi, Elli Bouakaz, Enny Sundell, Anna Velikova, and Silvia Stefanova for creating a bulgarian home atmosphere. I am forever grateful to Violeta Danova, who was both a friend and as a mother for me during the most difficult times, and helped me a lot with taking care of my son. Special thanks to Kester Simm for his love, care, and support, and for calming me down during the last stressful months. Whatever the future, I am happy we met.

This thesis is dedicated to my parents, who have always encouraged me to study, and to my son Yordan for his generous patience to have an always busy mother.

This work was funded by the Swedish Agency for Innovation Systems (VINNOVA) under contract #2001-06074.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, et al. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[2] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, et al. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[3] M.H. Ali, Walid G. Aref, Raja Bose, A.K. Elmagarmid, et al. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. In *VLDB Conference*, 2005.

[4] M. Nedim Alpdemir, Arijit Mukherjee, Norman W. Paton, Paul Watson, et al. Service-based distributed querying on the grid. In *First International Conference on Service-Oriented Computing - ICSOC*, pages 467–482, 2003.

[5] Amos II user's manual, http://user.it.uu.se/ udbl/amos/doc/amos_users_guide.html.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, 2003.

[7] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, et al. Cluster I/O with River: Making the fast case common. In *IOPADS*, pages 10–22, 1999.

[8] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.

[9] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *VLDB J.*, 13(4):333–353, 2004.

[10] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[11] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, 2005.

[12] Shivnath Babu, Rajeev Motwani, Kamesh Munadata, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pages 407–418, 2004.

[13] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

[14] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Çetintemel, et al. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[15] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD Conference*, pages 13–24, 2005.

[16] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *2nd Intl. Conf. on Mobile Data Management*, 2001.

[17] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, et al. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[18] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[20] Sirish Chandrasekaran and Michael J. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB J.*, 12(2):140–156, 2003.

[21] N. Chaudhry, K. Shaw, and M. Abdelguerfi (ed.). *Stream Data Management*. Springer, 2005.

[22] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[23] Liang Chen and Gagan Agrawal. Resource allocation in a middleware for streaming data. In *Proceedings of the 2nd workshop on Middleware for grid computing*, pages 5–10, 2004.

[24] Liang Chen, Kolagatla Reddy, and Gagan Agrawal. GATES: A grid-based middleware for processing distributed data streams. In *HPDC*, pages 192–201, 2004.

[25] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, et al. Scalable distributed stream processing. In *CIDR*, 2003.

[26] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

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

[28] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.

[29] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[30] R.G.G. Cattell (ed.). *The Object Database Standard: ODMG-99*. Morgan Kaufmann, 1999.

[31] Andrew Eisenberg, Jim Melton, Krishna G. Kulkarni, Jan-Eike Michels, and Fred Zemke. SQL: 2003 has been published. *SIGMOD Record*, 33(1):119–126, 2004.

[32] Ian T. Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, 2002.

[33] Ian T. Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid - enabling scalable virtual organizations. *Int. J. Supercomputer Applications*, 2001.

[34] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems. The Complete Book*. Prentice Hall, Inc., 2002.

[35] The Globus project, http://www.globus.org.

[36] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[37] Anastasios Gounaris, Norman W. Paton, Rizos Sakellariou, and Alvaro A. A. Fernandes. Adaptive query processing and the grid: Opportunities and challenges. In *DEXA*, pages 506–510, 2004.

[38] Anastasios Gounaris, Rizos Sakellariou, Norman W. Paton, and Alvaro A. A. Fernandes. Resource scheduling for parallel query processing on computational grids. In *5th International Workshop on Grid Computing (GRID 2004)*, pages 396–401, 2004.

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

[40] Peter M.D. Grey, Larry Kerschberg, Peter J.H. King, and Alexandra Poulovassilis (Eds.). *The Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*. Springer, 2003.

[41] Moustafa A. Hammad, Mohamed F. Mokbel, M. H. Ali, Walid G. Aref, et al. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.

[42] J. Hellerstein, M. Franklin, et al. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*

[43] C. Kesselman (eds.) I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[44] Milena Ivanova and Tore Risch. Customizable parallel execution of scientific stream queries. In *VLDB*, 2005.

[45] Milena Ivanova and Tore Risch. Customizable parallel execution of scientific stream queries. *Techical Report 2005-012 from the Dept. of Information Technology, Uppsala University, Sweden*, April 2005.

[46] Michael Jaedicke and Bernhard Mitschang. On parallel processing of aggregate and scalar functions in object-relational dbms. In *SIGMOD Conference*, pages 379–389, 1998.

[47] Christian S. Jensen, James Clifford, Shashi K. Gadia, Arie Segev, and Richard T. Snodgrass. A glossary of temporal database concepts. *SIGMOD Record*, 21(3):35–43, 1992.

[48] Milena Koparanova and Tore Risch. High-performance grid stream database manager for scientific data. In *European Across Grids Conference*, pages 86–92, 2003.

[49] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, et al. TelegraphCQ: An architectural status report. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.

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

[51] Richard Kuntschke, Bernhard Stegmaier, Alfons Kemper, and Angelika Reiser. StreamGlobe: Processing and sharing data streams in grid-based P2P infrastructures. In *VLDB Conference*, 2005.

[52] Bin Liu, Yali Zhu, Mariana Jbantova, Bradley Momberger, and Elke A. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, 2005.

[53] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowl. Data Eng.*, 11(4):610–628, 1999.

[54] LOFAR, http://www.lofar.nl/.

[55] LOIS - the LOFAR outrigger in Scandinavia, http://www.lois-space.net/.

[56] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

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

[58] Jim Melton. *Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann Publishers, Inc., 2003.

[59] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

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

[61] NORDUGRID: Nordic testbed for wide area computing and data handling, http://www.nordugrid.org/.

[62] Open Grid Services Architecture - Data Access and Integration, http://www.ogsadai.org.uk/.

[63] Michael A. Olson, Wei Hong, Michael Ubell, and Michael Stonebraker. Query processing in a parallel object-relational database system. *IEEE Data Engineering Bulletin*, 19(4):3–10, 1996.

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

[65] Norman W. Paton. Databases and the grid: Jdbc in wsdl, or something altogether different? In *First International IFIP Conference on Semantics of a Networked World (ICSNW)*, pages 1–13, 2004.

[66] Relational grid monitoring architecture, http://www.r-gma.org.

[67] Tore Risch and Vanja Josifovski. Distributed data integration by object-oriented mediator servers. *Concurrency and Computation: Practice and Experience*, 13(11):933–953, 2001.

[68] Tore Risch, Vanja Josifovski, and Timour Katchaounov. *Functional Data Integration in a Distributed Mediator System*, pages 211–238. In [40], 2003.

[69] Tore Risch, Milena Koparanova, and Bo Thidé. High-performance grid database manager for scientific data. In *WDAS*, pages 99–106, 2002.

[70] Elke A. Rundensteiner, Luping Ding, Timothy Sutherland, Yali Zhu, Brad Pielech, and Nishant Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[71] Elke A. Rundensteiner, Luping Ding, Yali Zhu, Timothy Sutherland, and Bradford Pielech. *CAPE: A Constraint-Aware Adaptive Stream Processing Engine*, pages 83–111. In [21], 2005.

[72] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A model for sequence databases. In *ICDE*, pages 232–239, 1995.

[73] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.

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

[75] 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.

[76] David W. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6(1):140–173, 1981.

[77] Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, et al. Distributed query processing on the grid. In *Third International Workshop on Grid Computing*, 2002.

[78] Richard T. Snodgrass and Ilsoo Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, 1986.

[79] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.

[80] Michael Stonebraker and Paul Brown. *Object-Relational DBMSs*. Morgan Kaufmann Publishers, Inc., 1999.

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

[82] Alexander S. Szalay, Jim Gray, Ani Thakar, Peter Z. Kunszt, et al. The SDSS skyserver: public access to the sloan digital sky server data. In *SIGMOD Conference*, pages 570–581, 2002.

[83] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[84] Douglas B. Terry, David Goldberg, David Nichols, and Brian M. Oki. Continuous queries over append-only databases. In *SIGMOD Conference*, pages 321–330, 1992.

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

[86] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, 2003.

[87] Richard H. Wolniewicz and Goetz Graefe. Algebraic optimization of computations over scientific databases. In *19th International Conference on Very Large Data Bases*, pages 13–24, 1993.

[88] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the Borealis stream processor. In *ICDE*, pages 791–802, 2005.

[89] Yong Yao and Johannes Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.