The Design and Implementation of the Conquest Query Execution Environment

Frank Fabbrocino, Eddie Shek, and Richard Muntz
Data Mining Laboratory
Computer Science Department
University of California, Los Angeles
Los Angeles, California 90024
{frank, shek, muntz}@cs.ucla.edu

Abstract

Given the enormous amount of geo-scientific data that has been collected and is currently being collected for study, a system for the rapid, efficient and extensible query processing of such data that effectively hides its vastness, heterogeneity and location is needed. The UCLA Data Mining Laboratory is developing Conquest, a system that provides a user-extensible, operator-based, parallel and distributed query execution system within OASIS, an environment for the collaborative study of potentially distributed and heterogeneous geo-scientific data. This paper presents the design and implementation of Conquest and its subsystems, and how it meets these goals. The focus of this paper is on the dynamic query execution environment within Conquest and the lessons and experiences learned during its development.

1 Introduction

Every day, huge amounts of geo-scientific data are collected by scientific institutions from instruments such as satellites orbiting the earth, and through experiments and simulations. Unfortunately, many obstacles prevent scientists from effectively studying this data. For example, scientists must determine where the data they need is located and how it can be obtained, and then how to convert the data into a format compatible with their scientific software. Furthermore, scientists must contend with the difficulties of processing the data, given the lack of interoperability and extensibility of scientific software in general. In summary, scientists must tackle the vastness, heterogeneity and distributed nature of scientific data, and then the inadequacies and inefficiencies of scientific software before they can even begin to analyze and interpret it.

Consider the complexities of cyclone tracking, where sea-level pressure (SLP) data is combined with wind directional data to track the paths of low-pressure zones across the earth over time. Scientists may wish to track cyclones over arbitrarily long periods of time, but given that such data has been collected over long periods, the potential for crossing data set boundaries and the resulting format incompatibilities complicates processing in conventional systems. Furthermore, identification of low-pressure zones involves minima extraction and refinement over very large multidimensional array data sets, and conventional database systems are inadequate for such mathematically intensive processing on large data sets. Finally, geo-scientific study is an evolving technology, and scientists may wish to make subtle changes to processing algorithms and parameters that are not easily supported by conventional systems.

OASIS [4] is a system under development at the Data Mining Laboratory at UCLA that attempts to address the vastness, heterogeneity, and distributed nature of geo-scientific data and provides an environment for the analysis, knowledge discovery, visualization and collaborative study of such data. Within OASIS, Conquest provides an extensible, parallel, geo-scientific query processing system that is fully interoperable with OASIS services and data objects, as well as conventional data repositories. Furthermore, it addresses the above complexities of studying geo-scientific data by providing an extensible, parallel and distributed system for rapid query development, refinement and execution.

In this paper, the design and implementation of Conquest is presented, giving particular attention to the implementation and operation of the dynamic query execution environment. The lessons learned and difficulties encountered during its implementation, as well as a background of the systems and technologies that influenced its design and implementation will also be discussed.

The remainder of this paper is outlined as follows. Section 2 gives a brief overview of Conquest and its architecture. Section 3 describes the implementation of the query execution environment in detail. Section 4 presents a discussion of the significant difficulties encountered during Conquest's implementation. Section 5 discusses the related work that has influenced the design and implementation of Conquest. Finally, Section 6 concludes the paper.

2 Conquest

Conquest is an extensible, parallel, geo-scientific query processing system designed to be interoperable with OASIS services and data objects. It harnesses the power of distributed processing nodes, from workstations to massively parallel machines, for the processing of geo-scientific data. Conquest consists of three key ingredients that together provide the basis for meeting these goals:

- *Data Modeling:* Conquest provides a data model that is both simple and expressive for representing the geo-scientific data both structurally and semantically. A collection of basic data types and an associated mapping allows the modeling of virtually all scientific phenomena.
- *Extensible Operator-Based Processing:* Queries in Conquest are built by interconnecting operators that each implements some unique functionality, but together they produce the desired results. The actual execution of these operators can be distributed across a network of processing nodes providing both intra- and inter-operator parallelism. The capability to define and implement new operators is provided.
- *Distributed Architecture:* Through its subsystems, Conquest provides the needed functionality for client interaction, query creation, compilation, and management, and finally, the dynamic execution environment for geo-scientific queries across distributed processing nodes. Through the use of CORBA

inter-process communication, hardware, software and network transparency is obtained.

The next 3 subsections examine each of these key components of Conquest in more detail, with the first examining the data model, the second examining the operators, and the third presenting the overall distributed system architecture.

2.1 Data Model

The *Conquest Data Model* is the representation of geo-scientific data in the system. The great diversity of geo-scientific phenomena appears to require an equally complex data model in the context of scientific software. However, Conquest provides a semantically rich but conceptually simple data model based on the observation that virtually all scientific phenomena can be modeled as a collection of basic data types and an associated mapping called a *field*.

In cyclone tracking for example, sea level pressure points are modeled as three dimensional coordinates with the first dimension latitude, the second dimension longitude and the third dimension time. Furthermore, each coordinate instance maps to a sea-level pressure value. With such a canonical form as the basis for processing, the heterogeneity of geo-scientific data representations is resolved and processing primitives can be implemented independently of the actual representation.

2.2 Operator-Based Processing

Given the task of designing a system that is both fast, efficient and extensible, Conquest utilizes the concept of *operators* as the basic building-block of queries. Each operator implements some unique processing function, and has one or more input streams but only one processed output stream. Users combine operators into a tree structure or *data flow expression* that represents the user's query and produces the required results. The leaves in this tree consist of operators that generate or retrieve data from such repositories as databases and flat files. The internal nodes are where the data is processed and the root node conducts some final processing of the data before it is returned to the user. Operators that are siblings in the tree operate in parallel for increased performance whereas those connected in a sequence form a parallel processing pipeline.

For example, a data flow expression for the cyclone tracking example mentioned in Section 1 is given in Figure 1. Four different operators are used in the query, one for reading sea level pressure from a database, one for extracting minima (points of low pressure) from this data, one for reading wind directional data from flat files, and finally, one for the actual cyclone tracking by processing the extracted sea-level pressure minima with the wind directional data. Including more advanced forms of processing or adding new functionality such as saving the results of a query is as simple as replacing or adding more operators to the data flow expression.

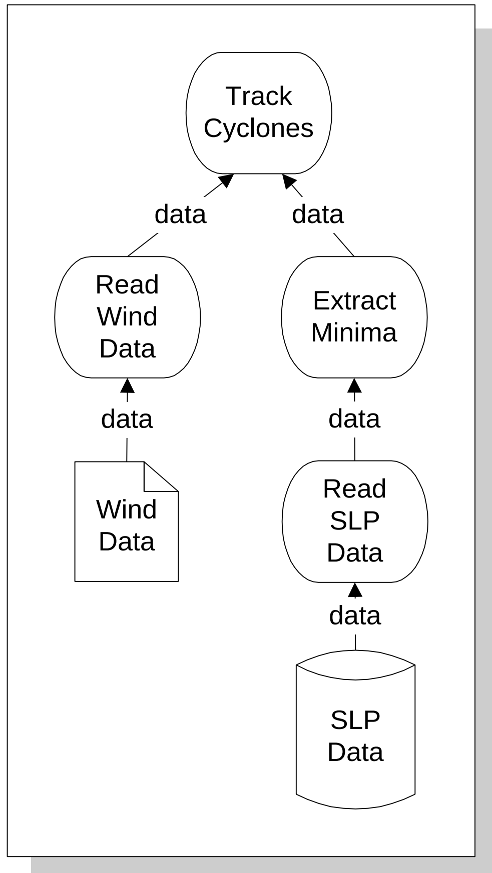

Figure 1: A data flow expression for cyclone tracking.

In order to combine operators to form queries in such a manner, each operator exposes the exact same interface of the following five methods and their associated semantics:

- `init()`: Initializes the operator with the passed arguments
- `open()`: Prepares the operator to begin processing
- `next()`: Returns the next granule of data processed by the operator
- `close()`: Gracefully stops the operator from processing
- `abort()`: Aborts current processing and closes the operator immediately

With this standard interface, the Conquest execution environment need not concern itself with what a particular operator does, and views the execution of a query as simply a collection of connected operators. Furthermore, users can add operators of arbitrary new functionality very easily, provided that they appropriately implement the required interface methods above. These required methods are referred to as the *Standard Operator Interface*.

For example, consider the "Read SLP Data" operator that reads the SLP data from a database system as illustrated in Figure 1. A call to `init()` would initialize the operator instance with the name of the database and the relation to be read from, as well as a reference to the consuming operator "Extract Minima". A call to `open()` would cause the operator to obtain a connection to the database, open the indicated relation, and prepare for retrieving the data. Calls to `next()` would cause the operator to retrieve data incrementally from the database and process it accordingly to extract the SLP data and send it to the "Extract Minima" operator. A call to `close()` would cause the operator to close the database connection, and finally, a call to `abort()` would immediately cause the operator to suspend all processing.

In the implementation of Conquest, there are actually two different types of operators, physical and exchange, although they implement the same Standard Operator Interface described above. The next two subsections explains what each type of operator does and why this distinction is necessary.

2.2.1 Physical Operators

Physical Operators implement the logical and algebraic operations necessary to perform computations and functions on geo-scientific data. For example, all of the operators illustrated in Figure 1 are Physical Operators. Each physical operator has attributes associated with it that are used during query construction, optimization and processing, including rules, optimization properties, and physical conditions. In order to achieve the goal of extensibility, Conquest provides the ability for users to add new, previously undefined operators to the system, provided that they adhere to the Standard Operator Interface and the Conquest Data Model. Furthermore, users are able to inherit behaviors and attributes of existing operators to simplify new operator development.

2.2.2 Exchange Operators

Given the distributed nature of Conquest, the functionality for an operator to send and receive data to and from other operators on other processing nodes is provided by the *Exchange Operator*. Essentially, the Exchange Operator hides the complexity of network communication by being placed between operators that are executing on different nodes and need to communicate. The Exchange Operator is actually a logical concept, with its implementation in two parts: an *Exchange Consumer* that resides on the sending node with facilities for sending data across the network, and an *Exchange Producer* that resides on the receiving node with complimentary facilities for receiving data from the network. In essence, an Exchange Operator is split in half, interfacing with Physical Operators on both nodes but utilizing inter-process communication between the two halves. This is illustrated in Figure 2 with the right side showing the logical view and the left side showing how the Exchange Operator encapsulates network communication.

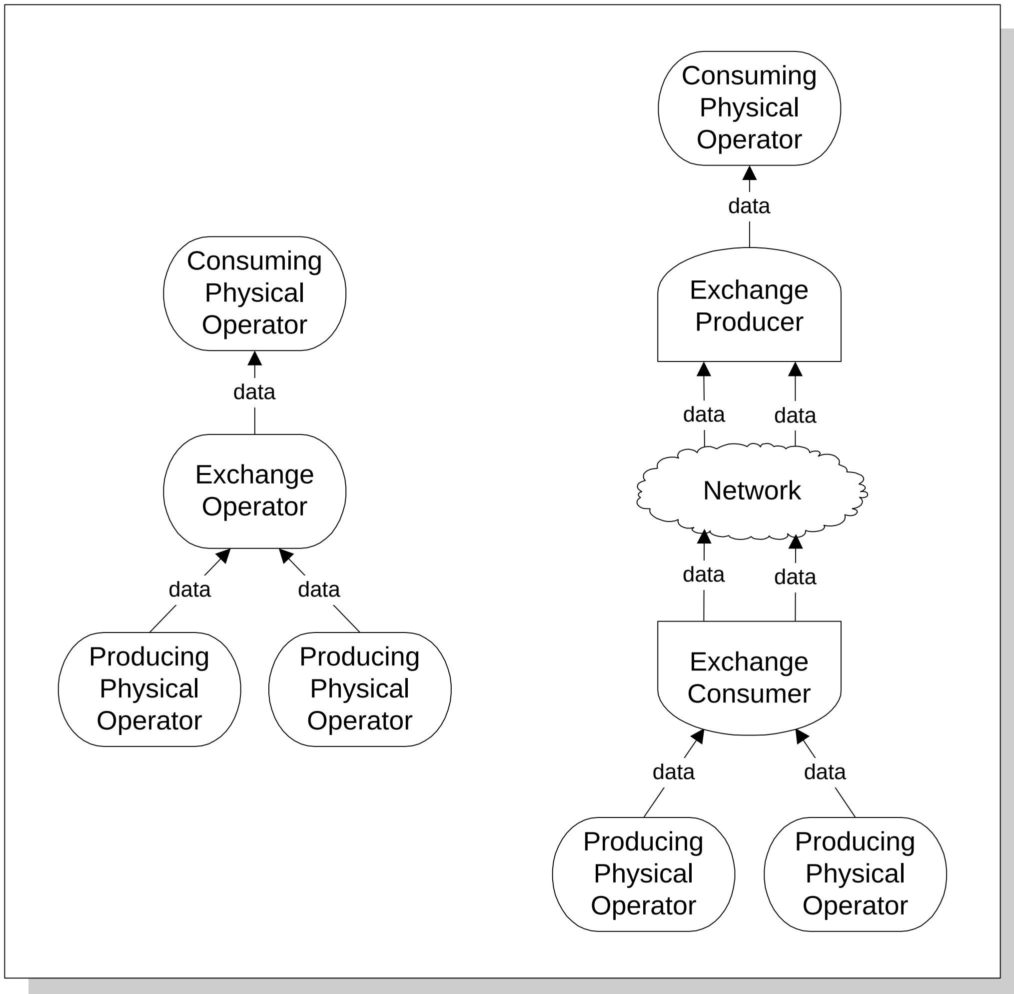

Figure 2: How the Exchange Operator hides network communication complexity.

2.3 Distributed Architecture

The architecture of Conquest is divided into four main sub-systems of potentially geographically distributed services:

- System Catalog
- Query Management
- Query Compilation
- Query Execution

Briefly, the System Catalog maintains metadata used by both the system and its users. Query Management provides users of Conquest with the ability to interact with the system to compose queries, locate and specify the input data-sets, and control the actual execution of the query. Query Compilation provides the services to compile and optimize queries that users have composed. And finally, the Query Execution sub-system consists of the dynamic, run-time environment within which Conquest queries execute. The next four subsections briefly present each of these sub-systems.

2.3.1 System Catalog

The System Catalog is a metadata repository used by both Conquest and its users. The System Catalog is divided into three areas: a Data Dictionary, an Operator Catalog, and a System Dictionary. The Data Dictionary contains information about both conventional and OASIS data resources that can be accessed in Conquest queries, such as the type and schema of the geo-scientific data the resources contain. The Operator Catalog maintains information about both predefined and user-defined operators and is used by users during building of queries and by the system during compilation and optimization. Finally, the System Dictionary maintains configuration information on Conquest itself, such as resource information on the processing nodes that are available for query processing. Virtually all parts of Conquest depend on the System Catalog and typically only one exists in a network of machines that can process Conquest queries. Figure 3 illustrates the internal structure of the System Catalog.

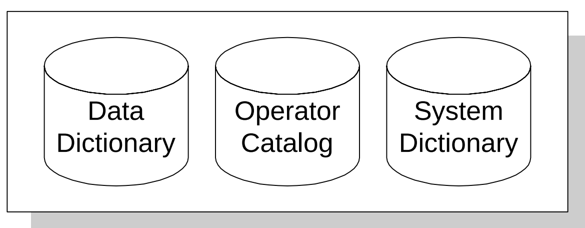

Figure 3: Components of the System Catalog

2.3.2 Query Management

The Query Management sub-system provides users with the functionality for initially building new queries or retrieving existing queries. The sub-system consists of two services, a Query Manager and a number of Query Agents. The Query Manager represents the entry-point to Conquest, from which users are able to build queries or retrieve existing queries. Once a query has been compiled successfully, the Query Manager creates a Query Agent that manages the execution of that query on behalf of the user. Query Agents are created on a per-query basis, and its interface allows users to begin, abort, pause and resume the execution of queries. This sub-system is illustrated in Figure 4.

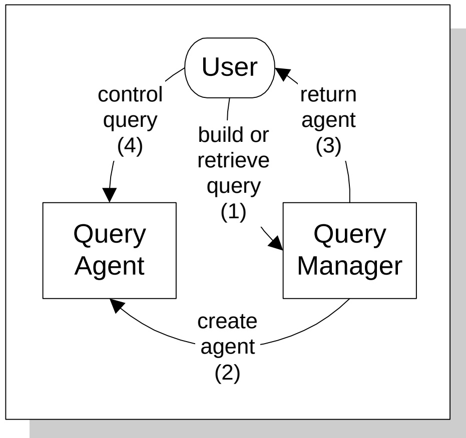

Figure 4: The Processes of Query Management

2.3.3 Query Compilation

The Query Compilation sub-system represents the services for the compilation and optimization of a Conquest query. The query a user has created using the Query Manager is simply a data flow expression with internal nodes representing operators and the leaves representing data sources. Before the query can be processed however, this expression must be converted into one that specifies exactly how the query is to be processed. This information includes not only the operators used, but the actual nodes on which the operators execute and the degree of parallelism on each node and between nodes. The resulting compiled and optimized query is called a *query execution plan.*

To accomplish this task, the sub-system consists of two parts, the Query Parser and the Query Optimizer. The Query Parser, given a query by the Query Manager, verifies that the query is both syntactically and semantically correct, and if so, passes it to the Query Optimizer, which compiles and optimizes the query into a query execution plan. By applying transformation and rewrite rules, the Query Optimizer produces a number of alternative query execution plans. Then heuristics are used to select the query execution plan with the least processing cost from the alternates. Once this process is complete, execution can be initiated by the user via the Query Agent that has been created by the Query Manager. Figure 5 illustrates this process, showing the interaction between the Query Manager and the Compilation sub-system.

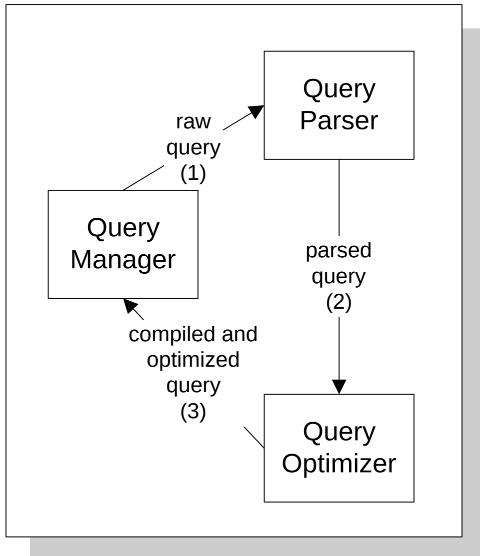

Figure 5: The Compilation/Optimization Process

2.3.4 Query Execution

The Conquest Query Execution sub-system, whose design and implementation is the central focus of this paper, provides the services for the dynamic, parallel execution of Conquest queries across a potentially heterogeneous, distributed network of computers. Given a query execution plan, the Query Execution sub-system analyzes it and creates the appropriate processes on each node involved and establishes the data communication paths between them. This process is called *materialization*. Once a query execution plan has been materialized, actual execution can be invoked by the user via the Query Agent. Materialization is a complex process that involves a number of services within the Query Execution sub-system, the creation of a number of servers, and a number of recursive calls. The next section will examine the Query Execution sub-system in more detail.

3 Query Execution Environment in Detail

Given the distributed nature of Conquest, the processing of a query is considerably more complex than in conventional systems since it involves the services of a number of potentially heterogeneous and geographically distributed processing nodes. To accomplish this, Conquest utilizes the dynamic object server creation and communication facilities of CORBA [5] extensively. The next four subsections examine the Execution environment in great detail, with the first subsection examining the overall structure, the following two subsections examining the two principle services used during query execution, and the last subsection describes actual query processing.

3.1 Structure

Each node available for processing a Conquest query is said to be *Conquest-enabled* when a single, static Scheduler Service has been installed on it and the appropriate information has been registered in the System Dictionary of the System Catalog. The Scheduler's function is to prepare the node for query processing by creating one or more Executor Servers. The Executor provides the framework in which one or more operators execute and the control and communication facilities and primitives for operators on different nodes to exchange data. These communication facilities are implemented using CORBA inter-process communication, thus concealing the hardware, software and network heterogeneity of the underlying systems. The collection of interconnected Executors represents the query processing environment.

When a user initiates the processing of a compiled query, the Query Agent contacts the Scheduler of the node at the root of the query execution plan. This Scheduler is known as the *Root Scheduler* and begins the materialization of the query execution plan by analyzing the query execution plan and contacting Schedulers on the indicated processing nodes, passing to each a subset of the query execution plan, or *fragment*. The fragment specifies only the part of the query execution plan that runs on the Scheduler's processing node. The contacted Schedulers analyze the fragment and create and initialize Executors appropriately, which in turn analyze the fragment to determine:

- which operators to dynamically load and initialize
- which Executors are the producers of this Executor's input stream
- which Executors are the consumers of this Executor's output stream

Executors that are created using the same fragment are said to belong to the same *Executor Group*. Executors that are consumers of another Executor's data are known as *Consumer Executors,* and those that are producers of data for another Executor are known as *Producer Executors*. Note that this is only a role distinction for the purpose of indicating relations between Executors, and in fact, most Executors are both Consumer Executors and a Producer Executors.

The actual algorithm for materialization implements a depth-first-search traversal of the query execution plan, contacting Schedulers to create Executors during the top-down phase, and establishing the producer/consumer communication paths during the bottom-up phase. Thus the overall Conquest execution environment can be viewed as a tree, where the leaf nodes are the Executors that implement operators that generate or retrieve the input data, the internal nodes are Executors that process this data, and the root node is the final Executor where the data receives some final processing and is returned to the user. Furthermore, within each of the Executors exists a tree of operators that represents that Executor's processing pipeline.

An example Query Execution structure is illustrated in Figure 6 for the cyclone tracking query given in Figure 1. There are three Executor Groups (denoted by the dotted boxes), one for reading sea-level pressure data and extracting minima, another for reading wind data, and finally, another for tracking cyclones. Because minima extraction is more computationally expensive than reading wind data, there are three Executor instances (denoted by solid boxes) in the minima extraction Executor Group, and thus overall processing benefits from the parallelism. However, many possible combinations of Executors and Executor Groups are possible, with the optimal configuration determined by the Query Optimizer at query compilation time. The next two subsections examine the Schedulers and Executors in even more detail.

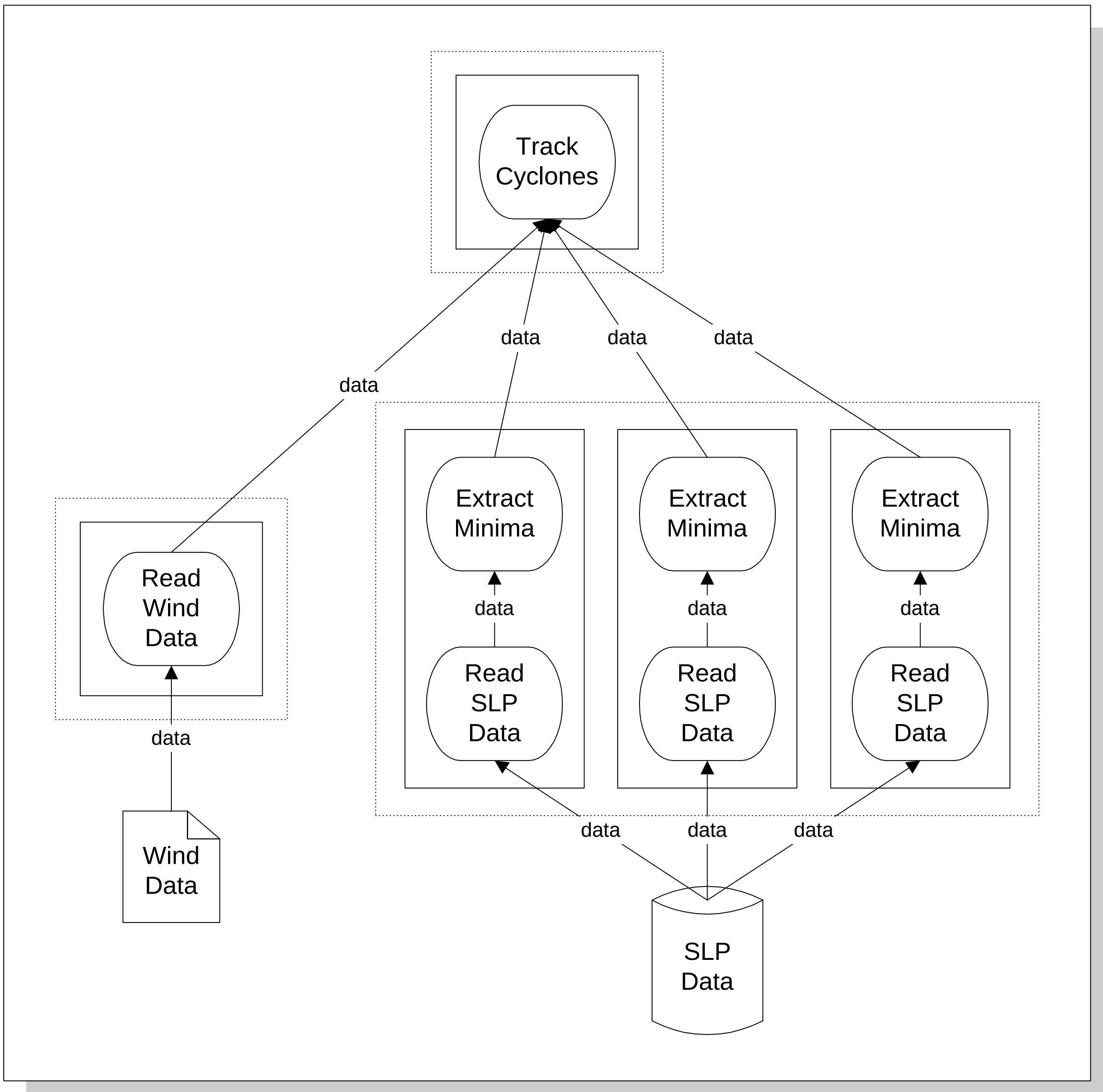

Figure 6: An example query execution environment for cyclone tracking.

3.2 Schedulers

Each Conquest Scheduler manages the query execution environment of a single processing node. During materialization, its job is to analyze the query execution plan and create the Executors in which one or more instances of operators execute and establish the communication paths between these created Executors and their corresponding Producer and Consumer Executors at other processing nodes. Once a query has been materialized and the user initiates query processing, the Scheduler performs no other functions until either the user aborts query processing or until query processing has completed. While a query is processing however, a scheduler may process other materialization requests from other Query Agents.

A number of Schedulers are involved in the materialization of a Conquest query but not all Schedulers contacted perform the same functions. Some Schedulers including the Root Scheduler are contacted to materialize Executor Groups and Executors, whereas some are contacted only to materialize Executors. This distinction in role is achieved through two similar but different public interface methods defined in the Scheduler's interface. The two methods are `materializeExecutorGroup()` and `materializeExecutor()`, with the former responsible for the materialization of an Executor Group, and the later responsible for the materialization of one or more Executors on a single node. Clearly, a call to `materializeExecutorGroup()` will result in one or more calls to `materializeExecutor()`. The two method prototypes, using simplified CORBA IDL, are:

```
NamedExchangeConsumerSeq materializeExecutorGroup(
                in QueryExecutionPlan qep,
                in NamedExchangeConsumerSeq consumers,
                in QueryAgent agent );

NamedExchangeConsumer materializeExecutor(
                in QEPFragment qep_fragment,
                in NamedExchangeConsumer consumer,
                in QueryAgent agent );
```

The arguments to `materializeExecutorGroup()` include the query execution plan, a sequence of object references to Consumer Executors, each with a sequence of port identifiers that indicate its Exchange Consumers, and an object reference to the Query Agent that is controlling the query's execution. The algorithm for `materializeExecutorGroup()` is implemented as two nested `for` loops where the outer iterates over each fragment in the query execution plan, and the inner contacts the appropriate scheduler invoking `materializeExecutor()`. The return value is a sequence of object references to Executors created and for each Executor, a sequence of port identifiers that indicate its Exchange Consumers.

The arguments to `materializeExecutor()` are very similar to those of `materializeExecutorGroup()`, except that information relevant to only a single Executor is passed. Thus, the caller passes a fragment, an object reference to a single Consumer Executor with a sequence of port identifiers that indicate its Exchange Consumers, and an object reference to the Query Agent. The algorithm then locates the parallelism information for the processing node and creates one or more Executors as indicated using the same fragment. For each Executor created, the Scheduler initializes it by passing the fragment that indicates the appropriate operators and communication information. The return value is an object reference to the Executor created with a sequence of port identifiers that indicate its Exchange Consumers.

After the call to `materializeExecutorGroup()` on the Root Scheduler completes, the query execution plan has been materialized and the Execution Environment is ready to begin processing the query's input. At this point, the top level Executor, or *Root Executor*, waits for the signal to begin query processing. When a query has been either aborted or completed, the Root Scheduler initiates the *de-materialization* of the Execution Environment, which is essentially the reverse of materialization: destroying the Executors created and freeing any allocated resources.

3.3 Executors

A Conquest Executor manages the environment in which a group of operators execute. The facilities it provides includes the ability to dynamically load operators and the functions for communication between its operators and those in the relevant Producer and Consumer Executors located on other processing nodes. In essence, the Executor provides the functionality of both the Exchange Producer and the Exchange Consumer to operators executing within it.

Schedulers create one or more Executors based on the parallelism information contained in the relevant fragment. Once an Executor is created, it is initialized with the fragment, and the fragment is analyzed by the Executor to load the indicated operators. The operators themselves are implemented as dynamically loaded libraries which, during initialization, are loaded into a C++ operator class that uses virtual functions to provide the Standard Operator Interface. Thus, during query processing, the operators are able to interoperate using the Standard Operator Interface and the Conquest Data Model without regard to what the operator does, and whether or not they are communicating with an Exchange Operator or Physical Operator.

To manage the communication between operators within an Executor and operators in other Executors, an Executor implements the functionality of the Exchange Producer Operator by including the Standard Operator Interface methods in its interface, and one or more Exchange Consumer Operators through the thread-per-invocation facilities of CORBA. When an operator in a Consumer Executor requests information from an operator in a Producer Executor, the former simply invokes the `next()` method defined on the Producer Executor. The implementation of `next()` by the Executor utilizes CORBA inter-process communication facilities, thus providing architecture, network and location transparency. In the case of Exchange Consumers, the Executor receives data asynchronously and places it into an input buffer for its Physical Operators to process.

The two public interface methods that an Executor implements, in addition to the Standard Operator Interface, are:

```
ExchangeConsumerPortSeq materializeQEP(
             in ConqQueryWKS::QEPFragment qep_fragment,
             in ExchangeConsumerPortSeq consumer,
             in ConqManagement::QueryAgent agent );
void destroy();
```

The method `materializeQEP()` implements the initialization functionality that the Scheduler invokes after the Executor is created. The arguments to it are a fragment, an object reference to a single Consumer Executor with a sequence of port identifiers that indicate its Exchange Consumers, and an object reference to the Query Agent. This method's algorithm is similar to that used in materialization in that it uses a depth-first-search traversal to dynamically build a tree of operators by loading the corresponding dynamically loaded libraries during the top-down phase and establishing the communication links between operators during the bottom-up phase. Then, the `init()` method is invoked on each operator instance with the arguments indicated in the query execution plan. The return value of `materializeQEP()` is a sequence of port identifiers that indicate the Executor's Exchange Consumers, if any.

The method `destroy()` implements the reverse of the initialization function. The method is invoked by the Root Scheduler when either the query processing completes or when it is aborted by the user through the Query Agent. In either case, the caller is the Scheduler that created the Executor via `materializeExecutor()`. Its implementation releases all resources allocated by the Executor and then destroys the Executor server instance itself.

3.4 Query Processing

Given the Standard Operator Interface and the Conquest Data Model, query processing is very simple. Once the query execution plan has been materialized, the user initiates query processing by signaling the Query Agent to begin. The Query Agent contacts the Root Executor and invokes the `open()` method on it to prepare the materialized Executor hierarchy for processing. This invocation travels recursively down the operator hierarchy until all the operators in this Executor are ready for processing to begin. Then, the call travels to the Producer Executors of the Root Executor and so on, until `open()` has been invoked on all operators.

Once completed, the Agent then invokes the `next()` method on the Root Executor to actually begin the processing of data. This call is also recursively invoked throughout all Executors and their operators in the Executor hierarchy. Once the bottom-most operators have received the `next()` invocation, they immediately begin to retrieve or generate the input data as appropriate and send it back up the operator and Executor hierarchy for processing, to the Root Executor, and ultimately to the user.

The stream of data within an Executor is *demand-driven*, meaning that the root operator explicitly requests data from its descendent operators. The stream of data between Executors, however, is *data-driven*, meaning that the Exchange Producers always attempt to transmit data to their Exchange Consumers, blocking only when there is no more data to transmit or when the Exchange Consumer's input buffer is full. The advantage of this distinction is the reduction of the communication latency between Executor instances, especially when considering the data marshaling costs and the potential communication latency, while still providing fine grain control over data streams within an Executor instance.

4 Discussion

In order to meet the goals outlined in Section 2, a number of difficulties surface that required non-trivial solutions. These difficulties included:

- automating system configuration
- operator environment
- operator encapsulation
- exchange consumer identification

The next four subsections discusses each of these difficulties in detail, the justification for the solutions chosen, and the resulting tradeoffs.

4.1 Automating System Configuration

Given the complexity in a potentially large-scale, geographically distributed system like Conquest, it was realized early on in its implementation that keeping system administration overhead to a minimum was an absolute necessity. In terms of the Execution Environment, this means reliably automating as much as possible the dynamic configuration information the system depends on. For a Conquest-enabled processing node to be considered for use in a query, it must be registered in the System Dictionary of the System Catalog. Once registered, the Optimizer can choose it as a processing node in a query execution plan, unless performance reasons disqualify it. The difficulty thus became, how can Scheduler registration be coupled with creation and destruction since these events determined when a Scheduler became available or not?

Thanks to the Lifecycle Services provided by the Object Development Framework in the Sun implementation of CORBA, it was possible to reliably automate Scheduler registration and de-registration with the System Catalog. The Lifecycle Services provided the necessary hooks by allowing the invocation of methods during object creation and destruction. In other words, it became possible to automatically invoke a function immediately after a Scheduler is created and destroyed. The method, `_initialize_new_ConqScheduler()`, automates registration by locating the System Catalog using the CORBA Naming Service, and then, after determining resource information such as hostname, type, number of CPUs, etc., invoking the `registerScheduler()` method on the System Catalog with this information to add it to the database. This information is then used by the Query Optimizer when comparing processing node capabilities and resources during optimization. Destruction of a Scheduler similarly invokes `unregisterScheduler()` on the System Catalog to remove this information from the database.

4.2 Operator Environment

With the features and communication facilities that CORBA provides, it seems obvious to implement operators as full-fledged CORBA objects. Therefore, Exchange Operators would no longer be needed since its functionality would become an integral part of all Physical Operators. Whether an operator was communicating data to an operator on the same processing node, or to a node across the network, the communication complexity would become completely transparent to the operators involved. However, this was decided against because CORBA inter-process communication facilities, although successful in providing architecture, network, and location transparency, involved too much overhead to provide the level of performance needed in such a data intensive application—even communication overhead between objects on the same processing node was significant! One study [1] showed that CORBA inter-process communication performance averaged 40 megabits per second, whereas sending the equivalent data using streams averaged 80 megabits per second.

Thus the concept of an Executor was born that would provide operators with the benefits of CORBA inter-process communication but without this overhead between operators executing on the same node. Operator communication within an Executor would result in only function call overhead, but only the communication between operators in different Executors would incur the CORBA inter-process communication overhead. Unfortunately, communication performance between Executors is still a serious concern, but work is underway to address this inadequacy.

4.3 Operator Encapsulation

In an attempt to reuse the operators implemented in an earlier version of Conquest (discussed in Section 5.2), it became necessary to find a way to easily implement the required Standard Operator Interface. The earlier version of Conquest specified that the operator be implemented as a library and provide the Standard Operator Interface, but prefix each function name with the name of the operator. For example, the "TrackCyclones" operator illustrated in Figure 1 would implement the interface: `TrackCyclones_open()`, `TrackCyclones_close()`, and so on. Thus the difficulty arose of how could operators be invoked via the Standard Operator Interface when their method names do not follow the Standard Operator Interface convention?

What was needed was a way to automatically mask out the prefixed function names so that a call to `open()` invoked `TrackCyclones_open()`, a call to `close()` invoked `TrackCyclones_ close()`, and so on. This was accomplished by using a combination of C function pointers and C++ classes. A C++ class, `physicalOpExec`, was defined with the Standard Operator Interface methods as function pointers. When an instance of this class is created, an initializer method dynamically loads the operator library and sets the interface function pointers to point to the appropriate methods. In the case of "TrackCyclones", `open()` would point to `TrackCyclones_open()`, `close()` would point to `TrackCyclones_close()`, and so on. Unfortunately, a level of indirection is introduced during method calls but at the benefit of not having to reimplement existing operators.

4.4 Exchange Consumer Identification

After having decided to load one or more operators within a single CORBA server, a problem arose in how to uniquely refer to the potentially numerous Exchange Consumer Operators a single Executor instance could have. A materialized Consumer Executor contains within it a tree of operators with Exchange Consumer Operators logically as leaves. These Exchange Consumer Operators receive data from one or more Producer Executors, but each Producer Executor sends data to a specific Exchange Consumer Operator. Given that these Producer Executors run on different machines, they would be asynchronously sending data to their Consumer Executor's Exchange Consumer Operators. However, if there is more than one Exchange Consumer Operator in the Consumer Executor, how can the Consumer Executor know for which Exchange Consumer Operator the data is for since the senders of the data have only an object reference to the Executor? Worse, an Exchange Consumer can receive data from multiple Producer Executors, so how could the Consumer Executor identify which Producer Executor the data is from?

Thus, some way was needed for an Executor to indicate not only *which* Exchange Consumer Operator in a Consumer Executor was to receive its data, but a way to uniquely identify the Executor to the Exchange Consumer. For simplification, it was decided that each Exchange Consumer would have one or more *ports* through which data from an Producer Executor would be received. Each port would have a unique number, even if the Executor contained more than one Exchange Consumer. Thus, the problem became, then how can the port number be conveyed when a Producer Executor sends data to a Consumer Executor?

Two potential solutions were explored, the first using CORBA sub-objects IDs and the second by passing an integer indicating a port along with the data. CORBA sub-object IDs provided the capability to use a portion of the object reference to store the port number. When an invocation is received, this information is automatically extracted by the ORB and passed as an argument to the function. The second solution is essentially an inexpensive implementation of sub-object IDs, in this case the caller explicitly passes the port number to the receiver. The second solution was chosen because it is simpler to implement, and provides the necessary functionality and adds only the cost of marshaling an integer.

5 Related Work

The design and implementation of Conquest incorporated ideas from a number of systems. The next four subsections briefly describe its four primary influences:

- Volcano, a query execution engine that introduced the concept of the exchange operator and the basic query execution model
- Conquest-PVM, an earlier implementation of Conquest using the PVM message-passing model
- CORBA, the object-oriented, distributed system standard
- OASIS, an environment for the study of geo-scientific data

5.1 Volcano

Volcano [2], recognizing the need for both high functionality and high performance in emerging database applications, provides a query execution engine that utilizes parallelism and operator-based processing extensively. Operators can be combined to form arbitrarily complex query evaluation plans. Furthermore, to provide for the distribution of such operators, Volcano introduces the exchange operator to hide operators from the complexities of process allocation, inter-process communication, flow control, and architectural differences.

Volcano also provided the basis for Conquest's query execution model, and Conquest extended it with the support of scientific data models including relational data, scientific data fields, and multidimensional arrays, and with operators specifically for geo-scientific data mining. Furthermore, support for various external file formats, such as HDF and netCDF, was added as well as database systems such as Illustra.

5.2 Conquest-PVM

Conquest-PVM [3] is an earlier implementation of Conquest utilizing the PVM message-passing facility for inter-process communication. Its primary goal was the development of an exploratory data mining system for the rapid expression and execution of geo-scientific queries in a network of single processor nodes and even massively parallel machines. The architecture consisted of three subsystems: a Scientist Workbench for query specification, a Query Manager for query compilation and optimization, and a Query Execution Server for the actual processing of queries. Queries in Conquest-PVM are operator based, providing both inter- and intra-operator parallelism.

5.3 CORBA

The Common Object Request Broker Architecture (CORBA) specification provided the facilities for building Conquest as a fully distributed system. It is the result of efforts from a consortium of computing technology institutions and vendors called the Object Management Group (OMG), whose ultimate goal is to specify an architecture for an open software bus via which object components provided by different vendors can interoperate across heterogeneous systems and networks, and the services such objects might require. With objects that are fully CORBA compliant, implementers will not have to deal with the issues of location and communication in potentially heterogeneous systems and networks. The architecture itself is an object-oriented, client/server model that consists of four main elements: a message-passing protocol, an object model, a virtual communication backbone, and a number of object services.

CORBA has proven to be a successful foundation upon which to build distributed systems, and was used extensively in both OASIS and Conquest. The facilities that it provides have significantly reduced the amount of development effort needed when compared to conventional methods such as Remote Procedure Call (RPC). Furthermore, given its efficient abstraction of remote invocations, even users with little understanding of distributed system concepts have been able to produce correct implementations. However, from the experiences in implementing both OASIS and Conquest, CORBA implementations must improve the performance of inter-process communication and provide more fully compliant implementations before CORBA can become a more general and widely used solution. Furthermore, the quality of the development environment leaves much to be desired.

5.4 OASIS

OASIS is a flexible, extensible, and seamless environment for scientific data analysis, knowledge discovery, visualization, and collaboration. It is funded by NASA under the Earth Observation System program, whose goal is to develop the next generation systems and technologies for the geo-scientific study of Earth's environment. The major design goals of OASIS include the development of an object hierarchy for the precise representation of geo-scientific data, a query processing facility for complex scientific queries involving large data-sets (which Conquest provides), a standard and efficient method for accessing a wide variety of data repositories, and finally, a basis for the integration of data analysis and visualization with data management. The foundation for the distributed architecture of OASIS is the CORBA standard.

The success of OASIS both as a distributed system and as a geo-scientific tool provided the impetus for re-implementing Conquest using CORBA, and integrating Conquest with OASIS. Thus, users of Conquest can access the features and services that OASIS provides, as well as the geo-scientific data repositories that it provides uniform access to.

6 Conclusion

Conquest is a distributed, geo-scientific query processing system that has successfully incorporated distributed object technologies to allow users to efficiently study geo-scientific data without concern for its vastness, heterogeneity and distributed nature. Furthermore, its extensibility allows users to address the continually changing and evolving requirements of geo-scientific study by providing the framework in which to incorporate new query processing functionality. Such extensibility and performance is necessary to provide scientists with the facilities for rapid query development, refinement and execution.

Bibliography

[1] A. Gokhale and D. C. Schmidt, **"Measuring the Performance of Communication Middleware on High-Speed Networks"**, *Proceedings of ACM SIGCOMM 1996*, August 1996.

[2] G. Graefe, **"Volcano, an Extensible and Parallel Dataflow Query Processing System"**, *IEEE Transactions on Knowledge and Data Engineering*, 1994.

[3] E. Mesrobian, R. R. Muntz, E. C. Shek, J. R. Santos, J. Yi, K. Ng, S. Y. Chien, C. R. Mechoso, J. D. Farrara, P. Stolorz, and H. Nakamura, **"Exploratory Data Mining and Analysis Using Conquest"**, *IEEE Pacific Rim Conference on Communications, Computers, Visualization, and Signal Processing*, Victoria, British Columbia, Canada, May 1995.

[4] E. Mesrobian, R. R. Muntz, E. C. Shek, S. Nittel, M. Kriguer, M. La Rouche, and F. Fabbrocino, **"OASIS: An EOSDIS Science Computing Facility"**, *International Symposium on Optical Science, Engineering, and Instrumentation, Conference on Earth Observing System*, Denver, Colorado, Aug. 1996.

[5] The Object Management Group, **The Common Object Request Broker: Architecture and Specification**, OMG Document 95.12.29 Revision 2.0, 1995.