River Design

Tobias Mayr (mayr@cs.cornell.edu)
Jim Gray (gray@microsoft.com)
12/13/2000

1 Introduction

We describe the design and the implementation of a relational river system for parallel query execution on a cluster. Rivers allow the exchange of relational data among dataflow operators executing on the different sites of a cluster. This allows both partition and pipeline parallelism, while offering a simple record iterator interface to the data processing code. Rivers are based on the data flow paradigm [DG90,DG92] and implement a form of exchange operators [G90,G93] and the rivers of [A+99,B+94].

Partitioned parallel data processing relies on an underlying mechanism that redistributes data among the parallel nodes. In the classical data-flow paradigm, relational operations are executed in parallel on a different subset of the data, a different *partition*, on each node. The partitioning of the data among the nodes is often specific to the executed operation, guaranteeing that the union of the results of the operations executed locally on each node is equivalent to the result of the operation executed on all data. For example, while sorting records, each node would get a specific range of values, or while joining two relations, each node would get a specific hash bucket.

The key to this paradigm is that operations are designed without regard to later parallelization. Each node executes the operation *sequentially* on *local* streams of data. Rivers encapsulate all aspects of parallelism and make it transparent to the operators, by offering simple, non-parallel record iterator interfaces.

Figure 1 shows how data processing is parallelized in the classical data flow paradigm. The same operation is executed on different subsets of the data on different nodes. Before the next operation is processed, the data are repartitioned among the nodes (arrows in the figure), either to optimize data flow or to satisfy semantic requirements of the next operation.

Our goal now is to build a simple river system that can be used as the communications layer for data-intensive applications that want to process large sets of data in parallel. Parallel applications do not need to be written from scratch; instead, embedding them in a river environment can parallelize existing systems. Our focus is the exchange of data between operators, across nodes, and not the many other aspects of parallel systems, like distributed metadata, parallel optimization, and distributed transactions.

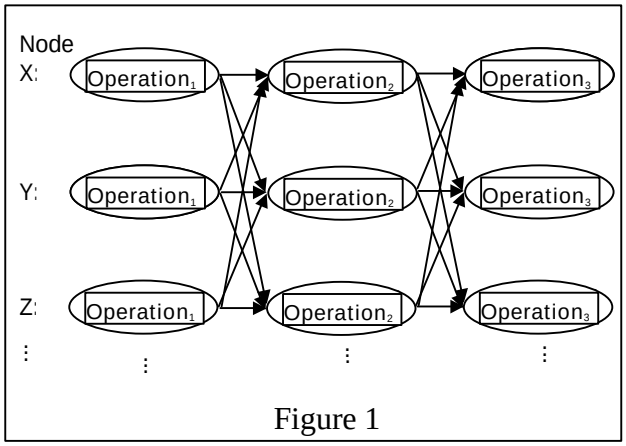

Figure 1

At this date, the communications mechanisms of the river system a fully implemented, while the launch mechanism and the XML parser is

incomplete. Nevertheless we present the projected design in Sections 3.5 and 3.6.
Section 2 describes river systems conceptually, while Section 3 describes the design that we chose to implement this conceptual framework.

2 River Concepts

Rivers are used to construct systems that execute a data processing application in parallel on multiple machines. The application is organized into separate *operators* that each consume and produce data. Different operators can be executed on different *nodes* exchanging data through rivers, or multiple *instances* of one operator can be executed on multiple sites processing different *partitions* – different subsets of the data as partitioned by the river system (partitioning is examined in the next section).
River systems view all data as composed of *records*. These records are organized into homogeneous *streams of records* – all records of a stream have the same type. Operators are programs that consume and produce record streams. Thus, each operator accesses the river system as a set of *record stream endpoints*. Endpoints are either *sources* or *sinks*. *Sources* offer a 'get record' iterator interface to a consumer of records. *Sinks* offer a 'put record' iterator interface to a producer. Figure 2 shows the abstract view of a river system.

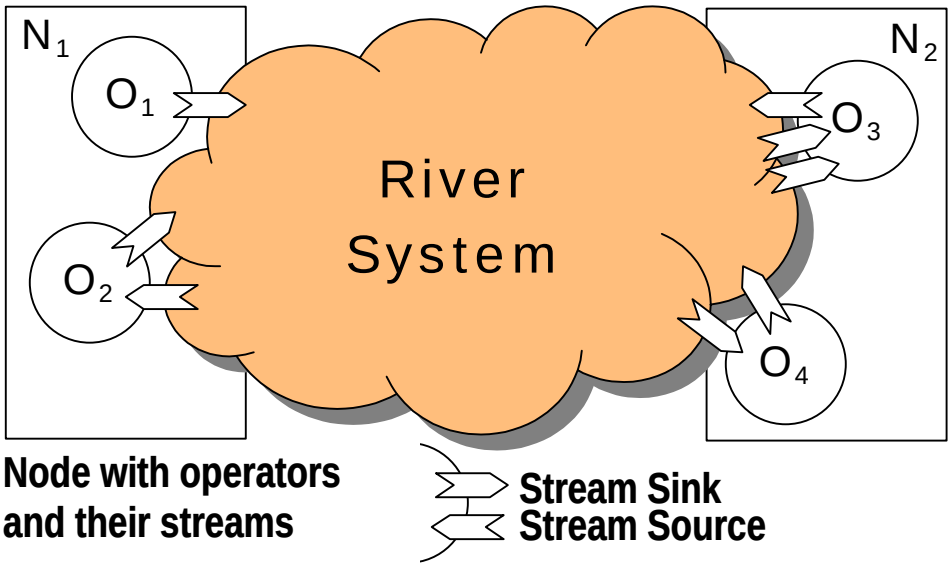

Figure 2: Abstract View of a River

A river system manages multiple *rivers*, each consisting of a set of record stream endpoints with records of the same type. Records are only exchanged between the sinks and sources of the same river. Different rivers are independent from each other and do not interact directly.
Operators can be composed using rivers to form pipelines – the output of one operator is sent to the inputs of another operator. This allows different programs to process the same data sequentially, introducing *pipelined parallelism.*
Multiple instances of one operator can be composed as consumers of one shared river and as producers of another. The parallelism between these programs, executing the same code on different partitions of the data, is known as *intra-operator parallelism*, or

partitioned parallelism. Both pipelined and partitioned parallelism are encapsulated in the river and transparent to the data processing programs themselves.
Figure 3 shows the role of rivers in the data flow example of Figure 1. There are two rivers involved, introducing parallelism within the operations, in the figure along the vertical dimension, and pipelined parallelism between producers and consumers of data along the horizontal dimension.

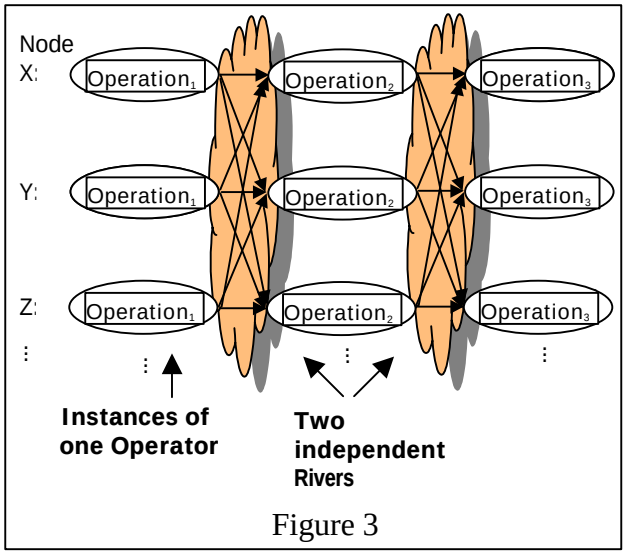

Figure 3

2.1 Partitioning of Record Streams

The record streams that are consumed through sources and produced through sinks are *exchanged* through the river. There are many variations on this: A sink can be *one-to-one connected* to a source, which outputs exactly the sequence of records that is received by the sink. Multiple sinks can be *n-to-one connected* to a source, which outputs an interleaving of the record sequences that were consumed by the sinks. A sink can be *one-to-n connected* to multiple sources by distributing its records among the sources. Each record consumed by the sink is output by one and only one source[1]. Each source's output is a subsequence of the record sequence consumed by the sink. Finally, multiple sinks can be *n-to-n connected* to multiple sources. Each sink's record sequence is distributed among all sources as in the one-to-n case, and each source interleaves all sequences that it thus receives as in the n-to-1 case. Some sources and sinks are not connected to others at all, but read from or write to local files.
The two cases 1-to-n and n-to-n involve the distribution of records from one sink among multiple sources. There are many different ways in which this can be done: round robin, range-partitioning, etc. We categorize methods in which the values of a record determine its receiver as *value-based* while we call all other methods *flow-based.* Section 2.3.3 and Section 3.2.2 examine how the distribution of records can be specified.

2.2 River Topologies

Rivers form an effective encapsulation for parallelism because the data processing operators are insulated from the issues of data movement and distribution. This happens through high-level communications abstractions, like n-to-n connected sources and sinks. Naturally, the price of this simplification of the operators is increased complexity in the implementation of the river system and its run-time parameterization.
A river system consists of a set of rivers, sets of operator instances with their location on the nodes of the system, and for each river a set of endpoints that connect it with operators. Additionally, each river has a specific connectivity of its endpoints within that river. For the sake of simplicity, we always describe all sinks of each river as n-to-n

[1] Some systems replicate records for multiple consumers. This currently not part of our design.

connected to all sources of that river. The other forms of connectivity are derived as special cases.
Formally, a river system is given by the following elements:

- *V* – a set of possible record values.
- *N* – a set of participating nodes.
- *R* – a set of rivers, with *SRC(r) = {r-src1, r-src2, …}* the set of sources and *SNK(r) = {r-snk1, r-snk2, … }* the set of sinks of the river *r in R*. We write *SRC(R)* for *union(r in R)(SRC(r))* and *SNK(R)* for *union(r in R)(SNK(r))*.
- *O* – a set of operators, with *|o|={o1,o2,…}* the set of instances of operator *o in O*. We write *|O| = union(o in O)(|o|)* for the set of all operator instances. Each operator has a set of ports *P(o) = {p1, p2, …}*, we write *P(|O|)* for the set of all ports of all instances.
- *L : |O| → N* – a mapping of operator instances into the set of nodes *N*: *L(oi)=n* means that instance i of operator *o* is located on node *n*.
- *U : (SRC(R) union SNK(R)) → P(|O|)* – a mapping of sources and sinks of all the rivers onto the ports of operator instances. The location of an endpoint *e in (SRC(R) union SNK(R))* can be defined as *L(e)=L(U(e))*. Thus the location mapping is extended to *L : |O| union (SRC(R) union SNK(R)) → N*
- For each river *r in R*, and each sink in that river *s in SNK(r)*, a mapping *Pr,s: V → SRC(r)* which maps record values to the sources of that river.

The last mapping, called a source's partitioning, determines which source will output a specific record that was given to the sink. There is an individual mapping from values to sources for each sink. Only value-based partitioning can be reflected in this model – dynamic, flow-based partitioning would be much harder to formalize. Despite of this, our design will allow for it.

2.3 Application-Specific Functionality

This section discusses components of the river system that are implemented as part of the application. These components are operators, record formats, and partitionings.

2.3.1 Operators

The data processing application runs on top of a river system as a cooperating group of operator instances. These operators interact with each other exclusively through rivers. Their access to rivers is limited to consumption and production of record streams through record sources and sinks in the river. Vice versa, the river system as an execution environment initiates and controls the execution of operators.
Arbitrary programs are allowed as operators as long as they implement a control interface that allows the river system to initialize, run and control them. The river system makes the required endpoints available the operators during their initialization.

2.3.2 Record Formats

As mentioned, records can have arbitrary application-specific formats. Because the river design should support a wide range of applications, the system should not commit to a particular physical record layout. Instead, river systems should rely on application-specific implementations of the record format.
The only requirement on the interface is that rivers must be able to recognize records in a stream of incoming bytes. For a given byte sequence, the river must know how many

bytes of it constitute each valid record. With this functionality the river system can segment a byte stream into a stream of records.

2.3.3 Partitioning

The third application-specific component is the partitioning of records for one-to-n or n-to-n connected sinks. The river has to forward each record to one of the connected sources. The choice of the source is made by an application-specific function.

3 River Components

This section describes the internal design top-down: river endpoints are implemented by record stream sources or sinks, which themselves can be mergers or partitioners of multiple underlying record streams. Record streams can also be translated to or from streams of fixed length byte buffers. These byte streams can be transferred through the network or stored and retrieved from the local file system.

Handling data transfers as streams of records requires knowledge about the physical layout of records. We first present the interface of record formats, before we describe how streams of such records are handled and how they are merged and partitioned. Finally we describe the translation between record streams and byte buffer streams and their interface to network and file services. Additional sections discuss the execution environment and XML specifications for river systems, although these components are not yet completed.

The following interfaces can be classified as *internal*, *external*, and *application-specific*. External interfaces are directly used by the application that uses the river system, while internal interfaces are not exported – they are presented here only to illustrate the internal river design. Application-specific interfaces are implemented by the application and are used by the river system to access application code. The following table summarizes the three categories.

External	Application-Specific	Internal
• River Sources • River Sinks	• Record Formats • Operators • Partitionings	• Merger/Partitioner • Byte Stream Record Endpoints • Byte Stream Endpoints

3.1 Record Formats

River-based applications handle data in the form of records, while network and file system only handle raw bytes. Rivers have to impose the abstraction of records onto processed byte streams. At the same time, the physical layout of records should be up to the application and not be dictated by rivers. In our design, applications contribute an implementation of the following record interface to the river system. Rivers only use record formats through the included methods.

Record Formats are an *application-specific interface.*

```
class RecordFormat {
public:
       virtual UINT GetRecordLength( );        // 0 if length is variable
       virtual UINT GetRecordLength( const BYTE* Record,
                                     UINT  MaxLength );
```

```
        virtual UINT GetNumberOfFields( );      // 0 if field is variable
        virtual UINT GetFieldLength( UINT FieldIndex );
        virtual UINT GetFieldLength( UINT FieldIndex ,
                                     const BYTE* Record );
        virtual BYTE* GetFieldValue( UINT FieldIndex ,
                                     BYTE* Record);
};
```

Because the river code itself does not manipulate records, it simply handles them as byte extents. Consequently, there is no specific class for records. This allows us to use blocks of bytes as contiguous sequences of records without additional copying.

An object of the class `RecordFormat` embodies all operations that are specific to a particular format. Record formats encapsulate both the byte layout and the schema information for records. It has functions to determine the fixed length of records, returning zero if the record size varies. In this case the length can be determined only for a specific record. The maximum length parameter allows us to apply the function to a potentially incomplete record. Zero is returned if not enough bytes are available to determine the length. Analogously, the length of a field can be determined in general or for a particular record. The final function returns a pointer to a particular field within the record.

As a sample implementation, the current code provides a fixed length record format with fixed length byte array fields.

3.2 River Sources and Sinks

Records in the river are accessed through record stream endpoints of rivers – sources or sinks. All record sources and sinks offer an iterator interface over record batches[2]. Both, sources and sinks, must be opened before the first and closed after the last request for records. Sources allow a check for 'end of stream', returning true if no more records will be returned. Both, a source's `GetNextRecords` and a sink's `PutNextRecords` have a boolean `Blocking` parameter. If it is set to true they block until the requested number of records have been processed. If not, the endpoint will return after processing as many records as possible without blocking. This allows the consumer or producer of records to 'try' if a source or sink is available. The static `WaitForSources` method of the source class also lets an operator block on multiple sources until the first one has any data available. These features make it easier to adapt to the flow of data: Data from available endpoints can be used first before accessing blocking endpoints.

An important design choice for iterator interfaces is *memory management*: Who deallocates the records that originated from a source or that were consumed by a sink? In our design, the records returned by a call to `GetNextRecords` are deallocated by the source during the next call to that function. So the consumer has to process the records or make a copy between two iterator invocations. The sink always makes its own copy or processes records before it returns from `PutNextRecords`. This only concerns the records that were reported as processed through the transient parameter `ActualNumberOfRecords`.

[2] Based on the experiences described in [B+94] and confirmed by experiments we did with batch sizes, it seems clear that per-record invocations of the iterator interface would come at a significant cost. Batch processing adds complexity to the operator but allows the river system to work more efficiently. Our variable batch sizes allow a tradeoff between both factors.

This choice of *implicit memory management* allows the standard iteration of getting records from a source and giving them to a sink without ever explicitly allocating or deallocating them. As an example, consider the sample implementations of the record pumps.

Record sources and sinks are an *external* interface.

```
class RecordSource {
public:
	void Open();   // establishes connections, dispatches asynchronous IO
	void Close();  // waits for outstanding IO, closes connections
	GetNextRecords(     // batch iterator request
		BOOL     Blocking,
		UINT     RequestedNumberOfRecords,
		BYTE**   Records,
		UINT*    ActualNumberOfRecords );
	BOOL EndOfStream(); // is more data available?
	static DWORD WaitForSources(            // Which sources will not block?
			RecordSource** Sources,
			ULONG NumberOfSources,
			BOOL Blocking );
}
class RecordSink {
public:
	void Open();   // establishes connections
	void Close();  // waits for outstanding IO, closes connections
	PutNextRecords(     // batch iterator request
		BOOL     Blocking,
		UINT     RequestedNumberOfRecords,
		Record** Records,
		UINT*    ActualNumberOfRecords );
}
```

Different implementations underlie the river record sources and sinks, depending on the connectivity within the river:

- Merging record sources: Record sources might internally merge record streams coming from several underlying internal record sources
- Buffer stream record sources: Record sources might internally construct their records from a stream of buffers coming from an internal byte source.
- Partitioning record sinks: Record sinks might internally partition their records onto several underlying record sinks.
- Buffer stream record sinks: Record sinks in the river might internally translate their records into a byte buffer stream that they output through an internal byte sink.

The following subsections present the different subclasses of `RecordSource` and `RecordSink` that implement these tasks. Merger sources interleave records coming from multiple sources, partitioner sinks distribute records onto multiple sinks, and bytes stream record sources respectively sinks translate byte buffer streams into record streams and vice versa.

3.2.1 Merger Record Sources

A merger record source offers a stream of interleaved records produced by a set of underlying record sources. The record format and the set of record stream sources are the

parameters of the merger. The access interface is that of the parent class. Whenever new records are requested, the merger will query its underlying sources and deliver records from some of the sources that have them available. It will never block on one source while others are available.
Merger record sources are an *internal* interface.

```
Class MergerRecordSource : public RecordSource {

       MergerRecordSource(
              RecordFormat*  Format,
              RecordSource** ArrayOfSources,
              UINT           LengthOfArray  );

}
```

3.2.2 Partitioner Record Sinks

A River sink accepts a stream of records and distributes each record, according to its partition, to one of the underlying record sinks. Parameters are the list of used record stream sinks and the partitioning function. The partitioning function determines for each record the index of its target partition. The partitioner only blocks when one of the underlying sinks that receive some of the records is blocking.
Partitioner record sinks are an *internal* interface.

```
Class PartitionerRecordSink: public PartitionerRecordSink {

PartitionerRecordSink (    DP_RecordFormat* RecordFormat;
                           RecordSink**     Sinks;
                           UINT             NumberOfSinks,
                           UINT (*pPartitionFunction)
                                  (PartitionerRecordSink* This,
                                   BYTE* Record) );
}
```

3.2.3 Byte Stream Record Sources and Sinks

These are record sources and sinks of the river that internally translate from or to buffer streams. They are implemented on top of buffer stream sources and sinks described in section 3.3. They assume that the buffer stream is a contiguous sequence of records – although records may span buffers. Their parameters are the used bytes stream source respectively sink and the used record format.
Byte stream record sources and sinks are an *internal* interface.

```
class ByteStreamRecordSource : public RecordSource {

       ByteStreamRecordSource( RecordFormat* , ByteStreamSource* );

}

class ByteStreamRecordSink : public RecordSink {

       ByteStreamRecordSink ( RecordFormat* , ByteStreamSink* );

}
```

3.3 Byte Buffer Sources and Sinks

Byte streams are handled in the form of a sequence of fixed length buffers because they are used for asynchronous I/O operations (i.e. disks, networks). The fixed buffer size corresponds to the size of a single asynchronous I/O request. Internally, these buffer streams are generated by various types of sources and processed by various types of sinks. As a simple connection, a buffer stream pump allows a direct transfer of buffers

between sources and sinks. Byte streams can be translated to record streams, allowing the use of record functionality (see Section 3.2.3).
Like record streams, sources and sinks offer an iterator interface over buffers. Both, sources and sinks, must be opened before the first and closed after the last request. Sources allow a check for 'end of stream', returning true if no more buffers will be returned. Both, a source's `GetNextBuffer` and a sink's `PutNextBuffer` have a boolean `Blocking` parameter. They only block until the request is processed if it is set to true. If not, the endpoint will return without blocking and without processing the request. This allows the consumer or producer of a byte stream to 'try' if a source or sink is available.
Memory management is done through a buffer pool interface that forces the source and sink users to explicitly deallocate buffers returned from `GetNextBuffer` and allocate buffers given to `PutNextBuffer`. Internally used buffers for new read requests in the source or from finished write request in the sink are automatically allocated respectively deallocated. Buffers returned from a source can be directly given to the sink, avoiding unnecessary copies or the mentioned explicit calls to the buffer pool.
Byte stream sources and sinks are an *internal* interface.

```
Class ByteStreamSource

{      void    Open();
       void    Close();
       Buffer* GetNextBuffer(BOOL Blocking);
       BOOL    EndOfStream();
}

class ByteStreamSink

{      void    Open();
       void    Close();
       BOOL    PutNextBuffer(Buffer* Buffer, BOOL Blocking);
}
```

The following sections describe different implementations of this interface. The two main ones are network and file endpoints, transferring the fixed-length buffers across the network respectively to a local file system. Additionally, null endpoints generate and consume data at insignificant CPU cost to allow testing and performance measurements.
We did very thorough performance studies for these components [MG00].

3.3.1 Network Sources and Sinks

Network endpoints are receiving and sending buffers through TCP/IP connections. On this level, there are only one-to-one connections. N-to-N connections are constructed using multiple network connections and record partitioners and mergers. Consequently, the functionality on this level is fairly simple. Parameters are the name of the remote host and the used port number.

```
Class ByteStreamSocketSource : public ByteStreamSource

{      ByteStreamSocketSource(
              LPCSTR HostName ,
              USHORT PortNumber ); }

Class ByteStreamSocketSink : public ByteStreamSink

{      ByteStreamSocketSink(
              LPCSTR HostName ,
              USHORT PortNumber ); }
```

3.3.2 File Sources and Sinks

File sources produce data read from a local file, while file sinks consume data and write them to file. There is no structure to the file, it simply contains the sequence of bytes consumed respectively produced by the endpoint. The only parameters are the local file names.

```
Class ByteStreamFileSource : public ByteStreamSource
{      ByteStreamFileSource(
              LPCSTR FileName ); }
Class ByteStreamFileSink : public ByteStreamSink
{      ByteStreamFileSink(
              LPCSTR FileName ); }
```

3.3.3 Null Sources and Sinks

These endpoints merely simulate data sources and sinks without significant resource usage. They produce buffers by simply allocating them and consume them by deallocation. The actual bytes in the buffer are never read or written by the endpoint. Still, the event synchronization mechanisms used for asynchronous IO are also used for these endpoints to make their behavior similar to that of file and network endpoints. A null source has the number of generated bytes as an argument, while the sink has no arguments.

3.4 Operators

So far we have seen data sources and sinks, handling either unstructured byte buffer streams or structured record streams. But, so far there is no way to couple sources and sinks. *Operators* are the universal way to combine the data of different rivers. An operator uses sources and sinks; it consumes data from the sources, processes them and produces results on the sinks. Operators implement the application that uses the river system. Consequently they are not part of the river code base. Nevertheless there are a few very basic operators that implement generic functionality and that can serve as examples for how operators work. The most basic function is to forward data between a source and a sink – we call an operator that does this a *data pump*. More specifically, an operator that forwards buffers from a buffer source to a buffer sink is called a *byte pump* and one that forwards records a *record pump*.

The shared interface of all data pumps is shown in the following. It requires initiation through an open call and final clean-up after a close call. The run function executes the pump: Synchronous execution means execution within the calling thread while asynchronous execution creates and uses a separate thread. While the pump is running, other threads can poll progress reports through the feedback function.

Operators are an *application-specific* interface.

```
class DP_Operator {
       void Open();
       void Close();
       void Run(bool Synchronous);
       DOUBLE GetFeedback();
}
```

3.5 River Specifications

In this section we will outline how the topology of a rive system can be specified using XML documents. This is just an illustration of river specifications, since the XML parsing and the related launch mechanisms have not been implemented yet. Appendix A shows a sample XML document that specifies a simple sort as a river system.
The four elements on the top level are:

- Nodes: Specifying the necessary information about the participating nodes, each node has a unique identifier and an IP address to allow TCP/IP addressing.
- Record Formats: Specifying each used record formats. The specifications consist of an identifier, a reference to an implementation and parameters that are specific to the implementation.
- Rivers: Each river has an identifier, a type, a reference to the used record format and lists of its sources and sinks.
    - Sources and Sinks: Have an identifier, a reference to the connected operator instance and internal information specific to the type of the river.
    - River can be of type 'FromFiles', 'ToFiles', and 'BetweenNodes'
    - 'FromFiles' are rivers without sinks that serve files through sources on the file's site. The specification contains the file path for each source.
    - 'ToFiles' are rivers without sources that write sink data to files on the sink's site. The specification contains the file path for each sink.
    - 'BetweenNodes' are the main form of rivers, partitioning data from each sink to all the sources according to an application-specific partitioning function. The specification contains an implementation and its parameters for each sink's partitioning.
- Operators: Each operator has an implementation identifier, parameters and a list of instances. The instances have identifiers, references to the node on which they are located and lists of sources and sinks of different rivers that they are using. They also have additional parameters that are specific to the execution node.

Whenever the document references an implementation, either for an operator, for a record format or for a partitioning, the used identifier is matched against a list of implementations available as static or dynamic libraries. This allows application developers to add new implementations to the system. The implementation references are always accompanied by a parameter field that is interpreted by the implementation code.
Thus all information that is needed to set up and execute a river system is given through an XML document. The next section describes how a centralized launcher uses this information to setup and control a river system.

3.6 Executing Rivers

Launching and synchronizing distributed computations is a crucial part of parallelizing an application. The river system uses a central controller process that launches and controls local river instances on each node. It distributes the parameters to the local river programs during their launch. The local programs interpret their site-specific parameters during their initialization. In regular intervals, the central controller polls progress information from each node until finally every node is done.
We explored several mechanisms for launching the local components remotely from the central controller. The solution we chose is to run them as distributed COM applications.

This allows the controller to simply construct and access them as COM objects. This allows easy start-up and monitoring by pulling information through method calls.
If rivers are used as part of an existing data processing application, the remote access mechanisms of that application might suggest a more appropriate launch and control mechanism. For example, database systems could be run as independent servers on each node, controlled by a controller that acts as a shared client. On the other hand, the delivered DCOM mechanism can distribute even applications that allow no remote access whatsoever, for example data processing libraries.
A central controller program, the *launcher*, creates instances of DCOM objects for all operator instances on their remote sites. One object is created per operator instance, but all instances at a node, along with the necessary river sub-structure run as individual threads within the same process. Each object has the operator interface described in section 3.4. The objects are initialized with river source and sink objects that implement the particular IO routines necessary to produce or consume the records from the used rivers.
For example, the operator 'Sort1' in the XML example above would produce its source records from a merger of data from two network connections with the sinks of 'Pump1' and 'Pump2'. Its sink records would be written to the specified file of the sink. Node A and node B would each run two objects - pump and sort - in individual threads of the same process. The launcher will poll progress information from each object in regular time intervals. The objects are shut down once the processing is completed.
The precise steps during initialization are as follows:

- The launcher constructs operators on each site, giving them all them all available parameters. The operators are given the needed river sources and sinks.
- As sinks are created, they return local connection information, like port numbers which the launcher passes on to the connected sources.
- The operators are started and perform all their local processing independently.
- The launcher polls progress information until every operator is done.
- The launcher shuts down the operators.

Our implementation relies crucially on Windows support in constructing the remote DCOM objects. DCOM component services and the used river objects must be installed on every site of the system. The remote object interface allows method calls with arbitrary arguments to the remote objects but not vice versa. This is why polling is used to track progress, while signaling of progress and termination by the objects might form an even better alternative. Only experience will show if the DCOM mechanisms are reliable and efficient enough to justify their use.
This design is still in the implementation phase and is presented here merely as an illustration.

References

[A+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, Katherine A. Yelick: Cluster I/O with River: Making the Fast Case Common. IOPADS 1999: 10-22

[B+94] Tom Barclay, Robert Barnes, Jim Gray, Prakash Sundaresan: Loading Databases Using Dataflow Parallelism. SIGMOD Record 23(4): 72-83 (1994)

[DG90] David J. DeWitt, Jim Gray: Parallel Database Systems: The Future of Database Processing or a Passing Fad? SIGMOD Record 19(4): 104-112 (1990)

[DG92] David J. DeWitt, Jim Gray: Parallel Database Systems: The Future of High Performance Database Systems. CACM 35(6): 85-98 (1992)

[G90] Goetz Graefe: Encapsulation of Parallelism in the Volcano Query Processing System. SIGMOD Conference 1990: 102-111

[G93] Goetz Graefe, Diane L. Davison: Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution. TSE 19(8): 749-764 (1993)

[MG00] Tobias Mayr, Jim Gray: Performance of the 1-1 Data Pump. See http://www.research.microsoft.com/~gray/River

Appendix A: Sample XML Specification

This example shows an XML specification for a simple sort application based on rivers. There are three rivers, `Input`, `Exchange`, and `Output`. `Input` and `Output` are file rivers that simply make local file data available as record streams. `Exchange` is a n-to-n connected river that repartitions data into sort buckets on the two involved nodes. Between `Input` and `Exchange`, instances of a simple record pump forward the streams. Between `Exchange` and `Output`, instances of the `Sort` operator sort the local buckets. Figure 4 shows the design.

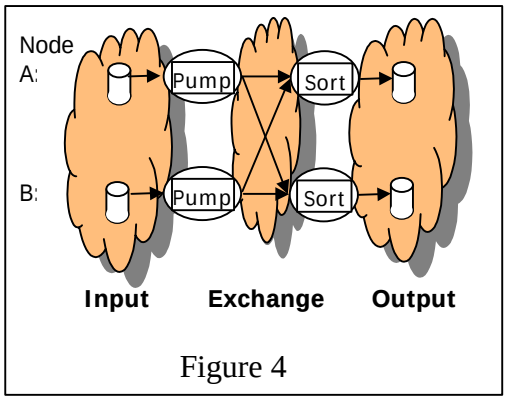

Figure 4

```
<?xml version="1.0" encoding="utf-8"?>
<root>
   <Nodes>
      <Node ID="A" IP-Address="157.57.184.42"/>
      <Node ID="B" IP-Address="157.57.184.43"/>
   </Nodes>
   <RecordFormats>
      <RecordFormat ID="Standard"
       Implementation="FixedLengthByteArray">
         <Parameters>
            <Fields>
               <Field Length=" 10 "/>
               <Field Length=" 90 "/>
            </Fields>
         </Parameters>
      </RecordFormat>
   </RecordFormats>
   <Rivers>
      <River ID="Input" Type="FromFiles "
       RecordFormat="Standard">
         <Sinks/>
         <Sources>
```

```
            <Source ID="Input.Source.1"
             ConnectedOperator="Pump1">
               <File Path="C:\data\partition1.data"/>
            </Source>
            <Source ID="Input.Source.2"
             ConnectedOperator="Pump2">
               <File Path="C:\data\partition2.data"/>
            </Source>
         </Sources>
      </River>
      <River ID="Exchange" Type="BetweenNodes"
       RecordFormat="Standard">
         <Sinks>
            <Sink ID="Exchange.Sink.1" ConnectedOperator="Pump1">
               <Partitioning Implementation="RangePartitioning">
                  <Parameters Ranges="[0,0.5*max,max]"/>
               </Partitioning>
            </Sink>
            <Sink ID="Exchange.Sink.2" ConnectedOperator="Pump2">
               <Partitioning Implementation="RangePartitioning">
                  <Parameters Ranges="[0,0.5*max,max]"/>
               </Partitioning>
            </Sink>
         </Sinks>
         <Sources>
            <Source ID="Exchange.Source.1"
             ConnectedOperator="Sort1"/>
            <Source ID="Exchange.Source.2"
             ConnectedOperator="Sort2"/>
         </Sources>
      </River>
      <River ID="Output" Type="ToFiles " RecordFormat="Standard">
         <Sinks>
            <Sink ID="Output.Sink.1" ConnectedOperator="Sort1">
               <File Path="C:\data\results1.data"/>
            </Sink>
            <Sink ID="Output.Sink.2" ConnectedOperator="Sort2">
               <File Path="C:\data\results2.data"/>
            </Sink>
         </Sinks>
         <Sources/>
      </River>
   </Rivers>
   <Operators>
      <Operator Implementation=" RecordPump">
         <Parameters/>
         <Instances>
            <Instance ID="Pump1" Node=" A">
               <Parameters/>
               <Sources Source="Input.Source.1"/>
               <Sinks Sink="Exchange.Sink.1"/>
            </Instance>
            <Instance ID="Pump2" Node=" B">
               <Parameters/>
               <Sources Source="Input.Source.2"/>
               <Sinks Sink="Exchange.Sink.2"/>
            </Instance>
         </Instances>
```

```
      </Operator>
      <Operator Implementation=" Sort">
         <Parameters SortFieldIndex="0" SortDirection="Ascending"
          VariousParameters="VariousValues"/>
         <Instances>
            <Instance ID="Sort1" Node=" A">
               <Parameters Range="[0,0.5*max]"/>
               <Sources Source="Exchange.Source.1"/>
               <Sinks Sink="Output.Sink.1"/>
            </Instance>
            <Instance ID="Sort2" Node=" B">
               <Parameters Range="[0.5*max,max]"/>
               <Sources Source="Exchange.Source.2"/>
               <Sinks Sink="Output.Sink.2"/>
            </Instance>
         </Instances>
      </Operator>
   </Operators>
</root>
```