Clustera: An Integrated Computation And Data Management System

David J. DeWitt
Erik Paulson
Eric Robinson
Jeffrey Naughton
Joshua Royalty
Srinath Shankar
Andrew Krioukov

Computer Sciences Department, University of Wisconsin-Madison, Madison, WI USA 53726
{dewitt, erobinso, srinath, epaulson, naughton, krioukov, royalty}@cs.wisc.edu

Abstract

This paper introduces Clustera, an integrated computation and data management system. In contrast to traditional cluster-management systems that target specific types of workloads, Clustera is designed for extensibility, enabling the system to be easily extended to handle a wide variety of job types ranging from computationally-intensive, long-running jobs with minimal I/O requirements to complex SQL queries over massive relational tables. Another unique feature of Clustera is the way in which the system architecture exploits modern software building blocks including application servers and relational database systems in order to realize important performance, scalability, portability and usability benefits. Finally, experimental evaluation suggests that Clustera has good scale-up properties for SQL processing, that Clustera delivers performance comparable to Hadoop for MapReduce processing and that Clustera can support higher job throughput rates than previously published results for the Condor and CondorJ2 batch computing systems.

1. Introduction

A little more than 25 years ago a cluster of computers was an exotic research commodity found only in a few universities and industrial research labs. At that time a typical cluster consisted of a couple of dozen minicomputers (e.g. VAX 11/750s or PDP 11s) connected by a local area network. By necessity clusters were small as the cost of a node was about the same as the annual salary of a staff member with an M.S. in computer science ($25K).

Today, for a little more than the annual salary of a freshly minted M.S., one can purchase a cluster of about 100 nodes, each with 1,000 times more capacity in terms of CPU power, memory, disk, and network bandwidth than 25 years ago. Clusters of 100 nodes are now commonplace and many organizations have clusters of thousands of nodes. These clusters are used for various tasks including analyzing financial models, simulating circuit designs and physical processes, and analyzing massive data sets. The largest clusters, such as those used by Google, Microsoft, Yahoo, and various defense laboratories, include over 10,000 nodes. Except for issues of power and cooling, one has to conclude that clusters of 100,000 or even one million nodes will be deployed in the not too distant future.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand

While it is always dangerous to generalize, clusters seem to be used today for three distinct types of applications:

- computationally intensive tasks
- analyzing large data sets with techniques such as MapReduce
- running SQL queries in parallel on structured data sets

Applications in the first class usually run as a single process on a single node. That is, rather than using multiple nodes to cooperatively run a single simulation, the general pattern is that the nodes execute independent instances of the simulation, each exploring a different portion of the parameter space. This style of usage is what Livny [1] refers to as "high throughput" computing.

The second type of application is typified by Google's MapReduce [2] software. This software has revolutionized the analysis of large data sets on clusters. A user of the MapReduce framework needs only to write map and reduce functions. The framework takes care of scheduling the map and reduce functions on the nodes of the cluster, moving intermediate data sets from the map functions to the reduce functions, providing fault tolerance in the event of software or hardware failures, etc.

There are two key differences between the class of jobs targeted by the MapReduce framework and those targeted by the first class of cluster software. First, MapReduce jobs run in parallel. For example, if the input data set is partitioned across the disks of 100 nodes, the framework might run 100 coordinated instances of the job simultaneously. The second major difference is that MapReduce is targeted towards data intensive tasks while the former is targeted towards compute intensive tasks.

The third class of cluster applications is one for which the database community is most familiar: a SQL database system that uses the nodes of a cluster to run a SQL query in parallel. Like MapReduce, such systems are targeted towards running a single job (i.e., a SQL query) in parallel on large amounts of data. In contrast to MapReduce, the programming model is limited to that provided by SQL augmented, to a limited degree, by the use of user-defined functions and stored procedures.

Despite their obvious differences, all three types of cluster systems have significant similarities. All have a notion of a job and a job scheduler. For Condor-like batch systems the job is an

executable to be run on a single node. For MapReduce it is a pair of functions. For database systems the job is a SQL query. All three also have a job scheduler that is responsible for assigning jobs to nodes and monitoring their execution. In the case of MapReduce and SQL, the scheduler also is responsible for "parallelizing" the job so that it can run on multiple nodes concurrently.

With this background, we turn our attention to the focus of this paper, which is to present the design, implementation, and evaluation of a new cluster management system called Clustera. Clustera is different from all previous attempts to build a cluster management system in three important ways. First, it is designed to run the three classes of jobs described above efficiently. Second, its design leverages modern software components including database systems and application servers as its fundamental building blocks. Third, the Clustera framework is designed to be extensible, allowing new types of jobs and job schedulers to be added to the system in a straightforward fashion.

The remainder of this paper is organized as follows. In Section 2 we present related work. Section 3 describes Clustera's software architecture including the three classes of abstract job schedulers we have implemented so far. In Section 4 we evaluate Clustera's performance, including a comparison of Clustera's MapReduce implementation with that provided by Hadoop on a 100-node cluster. Finally, our conclusions and future research directions are presented in Section 5.

2. Related Work

2.1 Cluster Management Systems

The idea of using a cluster of computers for running computationally intensive tasks appears to have been conceived by Maurice Wilkes in the late 1970s [3]. Many different cluster management systems have been developed including LoadLeveler [5], LSF [6], PBS [7], and N1 Grid Engine (SGE) [8]. Like Condor, LoadLeveler, LSF and PBS use OS files for maintaining state information. SGE, optionally, allows the use of a database system for managing state information (job queue data, user data, etc.). In contrast to these "application-level" cluster management systems that sit on top of the OS, some vendors offer clustering solutions that are tightly integrated into the OS. One example of this class of system is Microsoft's Compute Cluster Server [9] for Windows Server 2003. The focus of the GridMP [10] is to provide a framework within which developers can "grid-enable," or "port to the grid" pre-existing enterprise applications.

2.2 Parallel Database Systems

Parallel database systems have their roots in early database machine efforts [11,12,13]. MUFFIN [14] was the first database system to propose using a cluster of standard computers for parallelizing queries, a configuration that Stonebraker later termed "shared-nothing". Adopting the same shared-nothing paradigm, in the mid-1980s the Gamma [15] and Teradata [16] projects concurrently introduced the use of hash-based partitioning of relational tables across multiple cluster nodes and disks as well as the use of hash-based split functions as the basis of parallelizing the join and aggregate operators. Today parallel database systems are available from a variety of vendors including Teradata, Oracle, IBM, HP (Tandem), Greenplum, Netezza, and Vertica.

2.3 MapReduce

Developed initially by Google [2], and now available as part of the open source system Hadoop [17], MapReduce has recently received very widespread attention for its ability to efficiently analyze large unstructured and structured data sets. The basic idea of MapReduce is straightforward and consists of two functions that a user writes called **map** and **reduce** plus a framework for executing a possibly large number of instances of each program on a compute cluster.

The map program reads a set of "records" from an input file, does any desired filtering and/or transformations and then outputs a set of records of the form (key, data). As the map program produces output records a "split" function partitions the records into M disjoint buckets by applying a function to the key of each output record. The map program terminates with M output files, one for each bucket. In general, there are multiple instances of the map program running on different nodes of a compute cluster. Each map instance is given a distinct portion of the input file by the MapReduce scheduler to process. Thus, with N nodes in the map phase each producing M files there is a total of N * M files.

The second phase executes M instances of the reduce program. Each reads one input file from each of the N nodes. After being collected by the MapReduce framework, the input records to a reduce instance are grouped on their keys (by sorting or hashing) and fed to the reduce program. Like the map program, the reduce program is an arbitrary computation in a general-purpose language. Each reduce instance can write records to an output file, which forms part of the "answer".

2.4 Dryad

Drawing inspiration from cluster management systems like Condor, MapReduce, and parallel database systems, Dryad [18] is intended to be a general-purpose framework for developing coarse-grain data parallel applications. Dryad applications consist of a data flow graph composed of vertices, corresponding to sequential computations, connected to each other by communication channels implemented via sockets, shared-memory message queues, or files. The Dryad framework provides support for scheduling the vertices constituting a computation on the nodes of a cluster, establishing communication channels between computations, and dealing with software and hardware failures.

In many ways the goals of the Clustera project and Dryad are quite similar to one another. Both are targeted toward handling a wide range of applications ranging from single process, computationally intensive jobs to parallel SQL queries. The two systems, however, employ radically different implementation strategies. Dyrad uses techniques similar to those first pioneered by the Condor project based on the use of daemon processes running on each node in the cluster to which the scheduler pushes jobs for execution.

3. Clustera Architecture

3.1 Introduction

The goals of the Clustera project include efficient execution of a wide variety of job types ranging from computationally-intensive, long-running jobs with minimal I/O requirements to complex SQL queries over massive relational tables. Rather than "prewiring" the system to support a specific type of job, Clustera is designed to be extensible, enabling the system to be easily extended to handle new types of jobs and their associated schedulers. Finally, the system is designed to scale to tens of 1000s of nodes by

exploiting modern software building blocks including applications servers (e.g., JBoss) and relational database systems.

In designing and building this system we also wanted to answer the question of whether a general-purpose cluster management system could be competitive with one designed to execute a single type of job. As will be demonstrated in Section 4, we believe that the answer to this question is "yes".

3.2 The Standard Cluster Architecture

Figure 1 depicts the "standard" architecture that many cluster management systems use. Users submit their jobs to a job scheduler that "matches" submitted jobs to nodes as nodes become "available". Examples of criteria that are frequently used to match a job to a node include the architecture for which the job has been compiled (e.g. Intel or Sparc) and the minimum amount of memory needed to run the job. From the set of "matching" jobs, the job scheduler will select which job to run according to some scheduling mechanism. Condor, for example, uses a combination of "fair-share" scheduling and job priorities. After a job is "matched" with a node, the scheduler sends the job to a daemon process on the node. This process assumes responsibility for starting the job and monitoring its execution until the job completes. How input and output files are handled varies from system to system. If, for example, the nodes in the cluster (as well as the submitting machines) have access to a shared file system such as NFS or AFS, jobs can access their input files directly. Otherwise, the job scheduler will push the input files needed by the job, along with the executable for the job, to the node. Output files are handled in an analogous fashion. We use the term "push" architecture in reference to the way jobs get "pushed" from the job scheduler to a waiting daemon process running on the node.

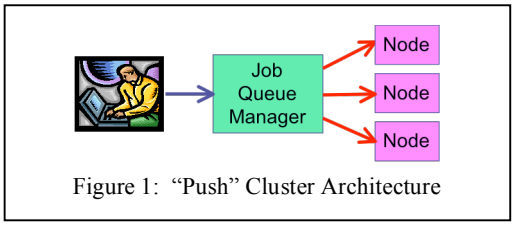

Figure 1: "Push" Cluster Architecture

Examples of this general class of architecture include Condor, LSF, IBM Loadleveler, PBS, and SunN1 Grid Engine. There are probably dozens of other similar systems. Despite this high-level similarity, there are differences between the systems, too. Condor, for example, uses a distributed job queue and file transfer to move files between the submitting machine and the node.

3.3 Clustera System Architecture

Figure 2 depicts the architecture of Clustera. This architecture is unique in a number of ways. First, the Clustera server software is implemented using Java EE running inside the JBoss Application Server. As we will discuss in detail below, using an Application Server as the basis for the system provided us a number of important capabilities including scalability, fault tolerance, and multiplexing of the connections to the DBMS. The second unique feature is that the cluster nodes are web service clients of the Clustera server, using SOAP over HTTP to communicate and coordinate with the Clustera server. Third, all state information about jobs, users, nodes, files, job execution history, etc. is stored in a relational database system. Users and system administrators can monitor the state of their jobs and the overall health of the system through a web interface.

The Clustera server provides support for job management, scheduling and managing the configuration and state of each node in the cluster, concrete files, logical files, and relational tables, information on users including priorities, permissions, and accounting data, as well as a complete historical record of each job submitted to the system. The utility provided by maintaining a rich, complete record of job executions cannot be overstated. In addition to serving as the audit trail underlying the usage accounting information, maintaining these detailed historical records makes it possible, for example, to trace the lineage of a logical file or relational table (either of which typically is composed of a distributed set of physical files) back across all of the jobs and inputs that fed into its creation. Similarly it is possible to trace forward from a given file or table through all of the jobs that – directly or indirectly – read from the file or table in question to find, for example, what computations must be re-run if a particular data set needs to be corrected, updated or replaced.

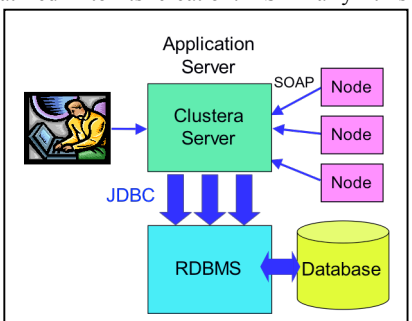

Figure 2: Clustera Architecture

Users can perform essential tasks (e.g., submit and monitor jobs, reconfigure nodes, etc.) through either the web-service interface to the system or via a browser-based GUI. Additionally, users can access basic aggregate information about the system through a set of pre-defined, parameterized queries (e.g., How many nodes are in the cluster right now? How many jobs submitted by user X are currently waiting to be executed? Which files does job 123 depend on?). While pre-defined, parameterized queries are quite useful, efficiently administering, maintaining, troubleshooting and debugging a system like Clustera requires the ability to get real-time answers to arbitrary questions about system state that could not realistically be covered by even a very large set of canned reports. In these situations one big benefit of maintaining system state information in an RDBMS is clear – it provides users and administrators with the full power of SQL to pose queries and create ad-hoc reports. During development, we have found that our ability to employ SQL as, among other things, a very high-powered debugging tool has improved our ability to diagnose, and fix, both bugs and performance bottlenecks.

The Clustera node code is implemented as a web-service client in Java for portability (either Linux or Windows). The software runs in a JVM that is forked and monitored by a daemon process running on the node. Instead of listening on a socket (as with the "push" model described in the previous section), the Clustera node software periodically "pings" the Clustera server requesting work whenever it is available to execute a job. In effect, the node software "pulls" jobs from the server.

While the use of a relational DBMS should be an obvious choice for storing all the data about the cluster, there are a number of benefits from also using an application server. First, application servers (such as JBoss [20], BEA's WebLogic, IBM's WebSphere, and Oracle Application Server) are designed to scale

to tens of 1000s of simultaneous web clients. Second, they are fault tolerant, multithreaded, and take care of pooling connections to the database server. Third, software written in Java EE is portable between different application servers. Finally, the Java EE environment presents an object-relationship model of the underlying tables in the database. Programming against this model provides a great deal of back-end database portability because the resulting Java EE application is, to a large extent, insulated from database-specific details such as query syntax differences, JDBC-SQL type-mappings, etc. We routinely run Clustera on both IBM's DB2 and on PostgreSQL.

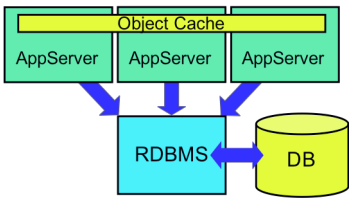

Figure 3: Clustered Application Server

As illustrated in Figure 3, application servers can also be clustered on multiple machines for enhanced reliability and scalability. In a clustered environment the application server software can be configured to automatically manage the consistency of objects cached on two or more servers. In effect, the Java EE application is presented the image of a shared object cache.

3.4 Concrete Files, Concrete Jobs, Pipelines, and Concrete Job Scheduling

Concrete jobs and concrete files are the primitives on which higher-level abstractions are constructed. **Concrete files** are the basic unit of storage in Clustera and correspond to a single operating system file. Concrete files are used to hold input, output, and executable files and are the building blocks for higher-level constructs, such as the logical files and relational tables described in the following section. Each concrete file is replicated a configurable number of times (three is the default) and the location for each replica is chosen in such a way to maximize the likelihood that a copy of the file will still be available even if a switch in the cluster fails. As concrete files are loaded into the system (or created as an output file) a checksum is computed over the file to insure that files are not corrupted.

As database practitioners we do the obvious thing and store the metadata about each concrete file in the database as illustrated in Figure 4. This includes ownership and permission information, the file's checksum and the location of all replicas. Since nodes and disks fail, each node periodically contacts the server to synchronize the list of the files it is hosting with the list the server has. The server uses this information to monitor the state of the replicas of each file. If the number of replicas for a concrete file is below the desired minimum, the server picks a new node and instructs that node (the next time it contacts the server) to obtain a copy of the file from one of the nodes currently hosting a replica.

A **concrete job** is the basic unit of execution in the Clustera system. A concrete job consists of a single arbitrary executable file that consumes zero or more concrete files as input and produces zero or more concrete output files. All information required to execute a concrete job is stored in the database including the name of the executable, the arguments to pass it, the minimum memory required, the processor type and the file IDs and expected runtime names of input and output files.

A **pipeline** is the basic unit of scheduling. Though the name implies linearity, a pipeline is, in general, a DAG (directed acyclic graph) of one or more concrete jobs scheduled for co-execution on a single node; the nodes of the pipeline DAG are the concrete jobs to execute and the edges correspond to the inter-job data-dependencies. For large graphs of inter-dependent jobs, the concrete job scheduler will dynamically segment the graph into multiple pipelines. The pipelines themselves are often sized so that the number of executables in the pipeline matches the number of free processing cores on the node executing the pipeline.

The inputs to and the outputs from a pipeline are concrete files. During pipeline execution, the Clustera software transparently enables the piping (hence the term "pipeline") of intermediate files directly in memory from one executable to another without materializing them on disk. Note that the user need not tell the system which jobs to co-schedule or which files to pipe through memory – the system makes dynamic co-scheduling decisions based on the dependency graph and enables in-memory piping automatically at execution time. (In the near future we plan to extend piping of intermediate files across pipelines/nodes.) This dynamic, transparent piping of data between jobs is similar to that employed in Dryad [14]. The appendix provides more details on how file access patterns can alter pipelining decisions.

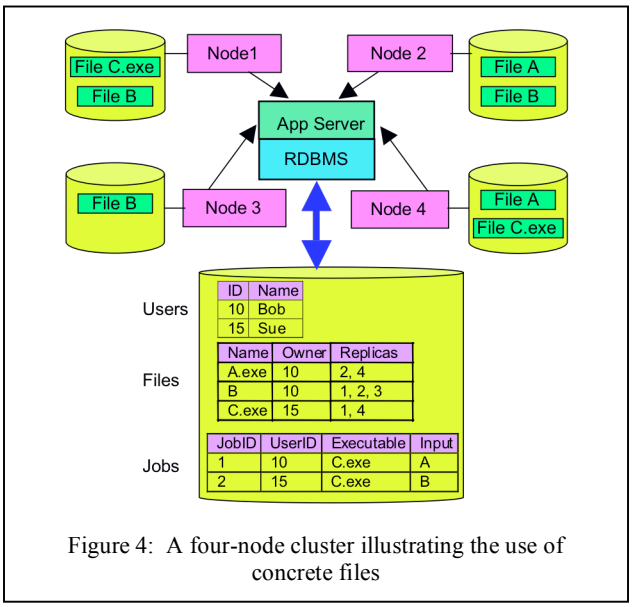

Figure 4: A four-node cluster illustrating the use of concrete files

The Clustera server makes scheduling decisions whenever a node "pings" the server requesting a pipeline to execute. Matching, in general, is a type of join between a set of idle nodes and a set of jobs that are eligible for execution [19]. The output of the match is a pairing of jobs with nodes that maximizes some benefit function. Typically this benefit function will incorporate job and user priorities while avoiding starvation. Condor, for example, incorporates the notion of "fair-share" scheduling to insure that every user gets his/her fair share of the available cycles [21]. In evaluating alternative matches, Clustera also includes a notion of what we term "placement-aware" scheduling which incorporates the locations of input, output, and executable files. The "ideal" match for a node is a pipeline for which it already has a copy of

the executable files and all input files. If such a match cannot be found then the scheduler will try to minimize the amount of data to transfer. For example, if a pipeline has a large input file, the scheduler will try to schedule that pipeline on a node that already has a copy of that file (while avoiding starvation).

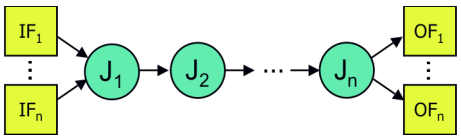

Figure 5: A very common class of pipeline consisting of a linear chain of n concrete jobs.

Figure 5 shows a linear pipeline of n jobs in which the outputs of one job are consumed as the inputs to the successor job. For this very common type of pipeline, only the first job in the pipeline consumes any concrete files as input and only the last job in the pipeline produces any concrete files as output; the rest of the data passes through in-memory pipes transparently to the executables.

Figure 6 shows a four-job DAG that runs as a single complex pipeline. In contrast to the Figure 5 pipeline, the Figure 6 pipeline illustrates that pipelines can be arbitrary DAGs. This single pipeline plan would likely be an excellent choice for execution on a four-core machine that already hosts copies of files LF1 and LF2. Figure 7 illustrates an alternative execution plan in which the four-job DAG of Figure 6 is segmented into three separate linear pipelines. Because they can run independently of each other, the first two pipelines are immediately eligible for execution. The third pipeline is dependent upon output files from the first two and is only eligible for execution after those output files have been materialized.

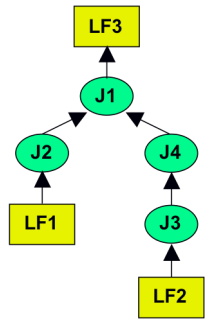

Figure 6: A 4-job DAG that could be executed as a single pipeline

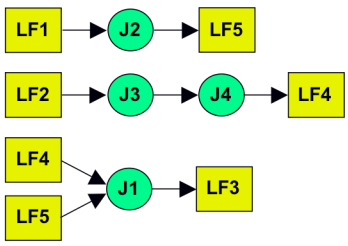

Figure 7: Possible alternative set of pipelines resulting from segmenting the Figure 6 DAG into three separate linear pipelines with materialized intermediate files.

In contrast to Figure 6, the segmented plan of Figure 7 is potentially more suitable when files LF1 and LF2 are large and do not reside on the same host. In this situation, the first two pipelines can be executed without any data transfer required. Since the decision on where to schedule the third pipeline can be deferred until that pipeline is actually ready for execution, the concrete job scheduler can wait to see which of the two intermediate files (LF4 and LF5) is larger and minimize the network bandwidth usage and data transfer time by scheduling the third pipeline to run on that node.

Figure 8 illustrates the set-up and execution of a single-job pipeline whose executable is File C, which is stored on node 1. In response to the request for work from node 1 the Clustera server returns a job identifier, the name of the executable and input files, and a list of locations of the replicas for each file. Node 1 discovers that it already has a local copy of C but not a copy of File A. Using HTTP, it requests a copy of file A from Node 2 (which includes an HTTP server). After retrieving File 1 from Node 2, Node 1 stores it locally and then uses JNI to fork C.exe.

3.5 Logical Files and Relational Tables

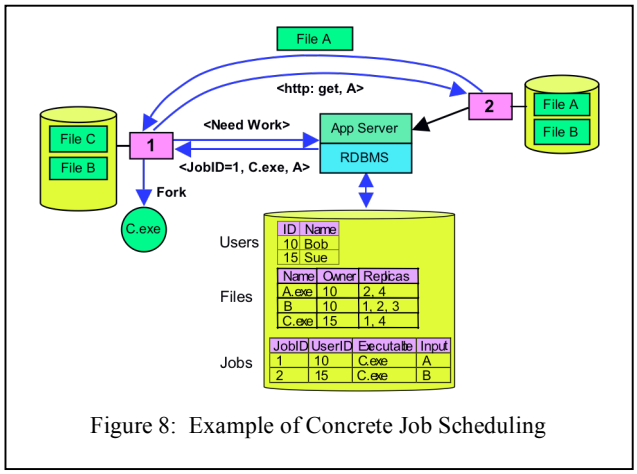

Figure 8: Example of Concrete Job Scheduling

In general, users interact with the system in terms of logical files and tables rather than concrete files. A logical file is a set of one or more concrete files that, together, form a single logical unit. When a logical file is first created a partitioning factor is specified which indicates the desired degree of declustering. For example, if the creation specifies a partitioning factor of 100, as the logical file is loaded into the system, it will be automatically partitioned into 100 concrete files. Each concrete file, in turn, will be placed on a different node in the cluster. The Server automatically takes care of placing the set of concrete files (and their replicas) across the nodes and disks of the cluster. As with concrete files, the database is used to keep track of everything including the file ownership and permissions as well as the identifiers of the concrete files that constitute the logical file. For more details on logical file management, please refer to the Appendix

A relation in Clustera is simply a logical file plus the associated schema information. We store the schema information in the Clustera database itself; an alternative approach is to store the schema information in a second logical file. We chose the first approach because it provides the SQL compiler with direct access to the required schema information, bypassing the need to access node-resident data in order to make scheduling decisions.

3.6 Abstract Jobs & Job Scheduler

3.6.1 Introduction

Abstract jobs and abstract job schedulers are the key to Clustera's

flexibility and extensibility. Figure 9 illustrates the basic idea. Users express their work as an instance of some abstract job type. For example, a specific SQL query such as *select * from students where gpa > 3.0* is an instance of the class of SQL Abstract Jobs. After optimization, the SQL Abstract Job Scheduler would compile this query into a set of concrete jobs, one for each concrete file of the corresponding students table.

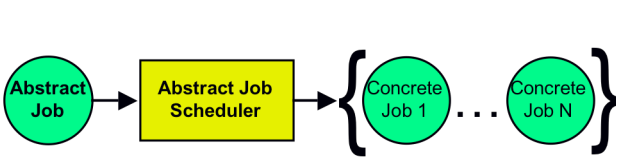

Figure 9: Compilation of abstract job into concrete jobs

Currently we have implemented three types of abstract jobs and their corresponding schedulers: one for complex workflows, one for MapReduce jobs, and one for running SQL queries. Each of these is described in more detail below.

3.6.2 Workflow Abstract Job Scheduler

A workflow is a DAG of operations such as the one shown in Figure 6. Workflows are common in many scientific applications. Workflows can be arbitrarily complex and are frequently used to take a relatively few base input files and pass them through a large number of discrete processing steps in order to produce relatively few final output files. It is not uncommon for the processing steps to use pre-existing executable files as building blocks. The true value of the workflow abstraction is that it provides end-users with a way to create new "programs" without writing a single line of executable code. Instead of writing code, users can create new dataflow programs simply by recombining the base processing steps in new and interesting ways. Workflows like this can often generate a large number of intermediate files that are of little or no use to the end-user: only the final output really matters. Workflows can also benefit from parallelism, but keeping track of the large number of processing steps and the intermediate files they depend on is a tedious, error-prone process, which is why users often use tools like Condor's DAGMan [23].

The Clustera Workflow Abstract Scheduler (WAS) accepts a workflow specification as input and translates it into a graph of concrete jobs that are submitted to the system. The information contained in the workflow specification is similar to that required for any workflow scheduler – the executables to run, the input files they require, the jobs that they depend on, etc. For the WAS, the base input files and the output files are all specified as logical files – the WAS will translate the logical file references into the concrete file references required by the system. Since the WAS input is a DAG of jobs, which the concrete job scheduler can deal with directly, little is required of the WAS beyond translating the file references and submitting the workflow to the scheduler.

3.6.3 MapReduce Abstract Jobs and Job Scheduler

A MapReduce abstract job consists of the name of the logical file to be used as input, the map and reduce functions, and the name to be assigned to the output logical file as shown in Figure 10. The Split, Sort (not shown), and Merge functions are provided by the system in the form of a permanently retained library of executables.

Given a MapReduce abstract job as input, the abstract scheduler will compile the abstract job into two sets of concrete jobs as shown in Figures 11 and 12. For the map phase (Figure 11) there will be one concrete job generated for each of the N concrete files of the input logical file. Each concrete job CF_i, $1 \le i \le N$, will read its concrete file as input and apply the user supplied executable map function to produce output records that the Split function partitions among M concrete output files $T_{i,j}$ $1 \le j \le N$.

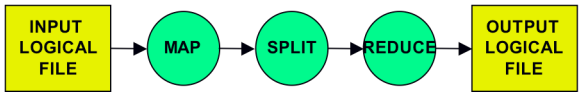

Figure 10: MapReduce Abstract Job Example

For the reduce phase (Figure 12), M concrete jobs will be generated during the compilation process. M is picked based on an estimate of how much data the map phase will produce. Once generated, the concrete jobs for the map and reduce phases can all be submitted to the scheduler. While the map jobs are immediately eligible for execution, the reduce jobs are not eligible until their input files have been produced.

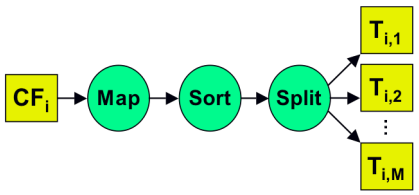

Figure 11: One of N concrete Map jobs generated by compiling the abstract job in Figure 10.

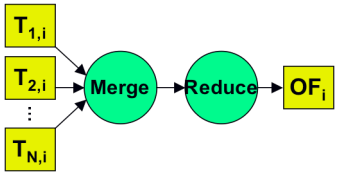

Figure 12: One of M concrete Reduce jobs generated by compiling the abstract job in Figure 10.

3.6.4 The SQL Abstract Job Scheduler

Currently, since we do not have a SQL parser and optimizer for Clustera, SQL plans must be constructed manually. The SQL plan is simply a standard tree of relational algebra operators. Each operator (i.e. selection, join, aggregate, etc.) was implemented in C++ and compiled into an executable file. Joins are implemented using the Grace hash join algorithm. Since we currently do not have indices, all selections are performed by sequentially scanning the input table. Aggregates are implemented by sorting on the group by attribute(s).

The SQL Abstract Job Scheduler is a two-phase compiler that combines the characteristics of the Workflow and MapReduce compilers. Like the Workflow compiler, the SQL compiler takes the query plan (which is itself a DAG of relational operators) and decomposes it into a DAG of operations that it orders according to the inter-operation data dependencies. For example, a query with two selections, two projections, a join and an aggregation on the join attribute will get turned into three linear sequences of operations. The first sequence will consist of the selection and

projection on the first base table, while the second sequence will be the selection and projection on the second base table. The third sequence will be the join (the outputs of the first two sequences are the inputs to the join) and subsequent aggregation. These three sequences are combined into one workflow graph. The compiler then checks the inputs to, and definitions of, all the join and aggregation operators to determine if any repartitioning is necessary. If so, it injects the necessary split and merge operators into the workflow prior to the join or aggregation operator in question. In the running example, the answer to whether or not the base tables require repartitioning prior to the join depends on whether or not they are already partitioned by the join attribute; since the aggregation is on the join attribute, however, we know that the aggregation operator will not require a repartitioning.

Once the SQL query has been translated into a workflow over relational tables, the workflow is processed in a fashion similar to the way MapReduce jobs are compiled. That is to say, each sequence will be compiled into a set of concrete jobs - one for each concrete file of the corresponding input table. As these concrete jobs are generated they are inserted into the database where they await scheduling by the concrete job scheduler.

3.6.5 Discussion

As described, pipelines of co-executing jobs are the fundamental units of scheduling in Clustera. This approach provides Clustera with two important opportunities for achieving run-time performance gains. First, the concrete job scheduler can take advantage of data-dependency information to make scheduling decisions that improve overall cluster efficiency. These decisions may, for example, help reduce network bandwidth consumption (by scheduling job executions on nodes where large input data files already reside) and disk I/O operations (by scheduling related executables to run on the same node so that intermediate files can be piped in memory rather than materialized in the file system) resulting in higher job throughput and reduced end-to-end execution times. Second, the concrete job scheduler can take advantage of multi-core nodes by dynamically segmenting the workload in order to match the number of executables in a pipeline to the number of available processing cores on the node.

3.7 Wrapup

Before moving on to the performance analysis in Section 4, it is interesting to consider one other architectural aspect in which Clustera significantly differs from comparable systems – the handling of inter-job dependencies. As was described above, Clustera is aware of, and explicitly and internally manages, inter-job data dependencies. Condor, on the other hand, employs an external utility, DAGMan [23], for managing the inter-job dependency graph. Users submit their workflow DAGs to the DAGMan meta-scheduler utility. DAGMan analyzes the workflow graph to determine which jobs are eligible for submission and submits those to an underlying Condor process that performs job queue management – the *schedd*. DAGMan then "sniffs" the relevant *schedd*'s log files in order to monitor job progress. As submitted jobs complete, DAGMan updates the dependency graph in order to determine when previously ineligible jobs become eligible for execution. As these jobs become eligible DAGMan submits them to the Condor *schedd*.

For Clustera, probably the biggest benefit of managing the dependencies internally is that it provides the concrete job scheduler with the ability to do more intelligent scheduling. It is clearly impossible to enable in-memory (or cross-node) piping of intermediate data files if the jobs that would consume the piped data are not even visible to the system until after the jobs that produce that data have completed. One other noteworthy advantage of handling the dependencies internally is that it simplifies examining and/or manipulating a workflow as a unit. More details of our scheduling mechanisms can be found in [24].

4. Performance Experiments and Results

As stated previously, a primary goal of the Clustera project is to enable efficient execution of a wide variety of job types on a potentially large cluster of compute nodes. To gauge our level of success in achieving this goal we ran a number of experiments designed to provide insight into the performance of the current prototype when executing three different types of jobs. Section 4.1 describes the experimental setup. The MapReduce results are presented in Section 4.2, the SQL results in Section 4.3, and arbitrary workflow results in Section 4.4. Section 4.5 explores the performance of the Clustera Server under load. A more complete set of experimental results along with more details about the system can be found in [27].

To give some additional context to the Clustera MapReduce and SQL performance numbers, we also ran comparable experiments using the latest stable release (version 0.16.0) of the open-source Hadoop system. There are four key reasons that we chose to compare Clustera numbers with Hadoop numbers for these workloads. The first reason, as laid out in the introduction, is to observe whether a general-purpose system like Clustera is capable of delivering MapReduce performance results that are even "in the same ballpark" as those delivered by a system, like Hadoop, that is specifically designed to perform MapReduce computations. The second reason is to understand how the relative performance of a moderately complex operation (e.g., a two-join SQL query) is affected by the rigidity of the underlying distributed execution paradigm; in substantially more concrete terms, this goal can be rephrased as trying to understand if it makes a difference whether a SQL query is translated to a set of MapReduce jobs or to an arbitrary tree of operators in preparation for parallel execution. The third reason is to understand if there are any structural differences between the two systems that lead to noticeably different performance or scalability properties. The fourth reason is to ascertain whether the benefit (if any) of enriching the data model for a parallel computing system (e.g., by making it partition-aware) is sufficient to justify the additional development and maintenance associated with the enriched model.

Note that providing a performance benchmark comparison between the two systems (or between Clustera and a parallel RDBMS) was not a goal of our experimentation. Parallel computation systems are very complex and, as will be discussed further in Section 4.3, can be sensitive to a variety of system parameters. While undoubtedly an interesting investigation in its own right, tuning either Hadoop or Clustera (much less both) to optimality for a particular workload is beyond the scope of this study and, along with a performance comparison including a parallel RDBMS, is deferred to future work. We are not attempting to make any claims about the optimally tuned performance of either system and the numbers should be interpreted with that in mind.

4.1 Experimental Setup

We used a 100-node test cluster for our experiments. Each node

has a single 2.40 GHz Intel Core 2 Duo processor running Red Hat Enterprise Linux 5 (kernel version 2.6.18) with 4GB RAM and two 250GB SATA-I hard disks. According to hdparm, the hard disks deliver ~7GB/sec for cached reads and ~74MB/sec for buffered reads. The 100 nodes are split across two racks of 50 nodes each. The nodes on each rack are connected via a Cisco C3560G-48TS switch. The switches have gigabit ethernet ports for each node and, according to the Cisco datasheet, have a forwarding bandwidth of 32Gbps. The two switches are connected via a single gigabit ethernet link. In all our experiments the participating nodes are evenly split across this link.

For the Hadoop experiments we used an additional desktop machine with a single 2.40 GHz Intel Core 2 Duo processor and 2GB RAM to act as the Namenode and to host the MapReduce Job Tracker. For the Clustera experiments we used two additional machines - one to run the application server and one to run the backend database. For the application server we used JBoss 4.2.1.GA running on the same hardware configuration used for the Hadoop Namenode/Job Tracker. For the database we used DB2 v8.1 running on a machine with two 3.00 GHz Intel Xeon processors and 4GB RAM. The database bufferpool was ~1GB which was sufficient to maintain all of the records in memory.

As mentioned in Section 3, Clustera accepts an abstract description of a MapReduce job or SQL query and compiles it into a DAG of concrete jobs. For MapReduce jobs, the *map* and *reduce* functions are specified as part of the MapReduce specification, with the *sort*, *split*, and *merge* functions taken from a system-provided library. These library functions are re-used in the compilation of SQL queries. In addition SQL workflows also use *select*, *project*, *hashJoin*, *aggregate* and *combine* library functions. All MapReduce and SQL library functions are Linux binaries written in C.

4.2 MapReduce Scaleup Test

This section presents the results of a series of scale-up experiments we ran to evaluate Clustera performance for MapReduce jobs. The base data for these experiments is the LINEITEM table of the TPC-H 100 scale dataset. The MapReduce job itself performs a group-by-aggregation on the first attribute (order-key) of the LINEITEM table. For each record in the input dataset the map function emits the first attribute as the key and the entire record as the value. After this intermediate data has been sorted and partitioned on the key (per the MapReduce contract), the reducer emits a row count for each unique key. If the goal were simply to calculate these row counts, then clearly this computation could be implemented more efficiently by having the map function emit, e.g., a "1" instead of the entire record as its output. For experimentation, however, the actual computation is immaterial; observing the performance impact of large intermediate data files is, however, quite interesting and important.

As explained earlier, we ran a set of MapReduce scale-up experiments on both Clustera and Hadoop in order to try to understand, among other things, whether Clustera could deliver MapReduce performance results in the same general range as those delivered by Hadoop. For the scale-up experiments we varied the cluster size from 25 nodes up to 100 nodes and scaled up the size of the dataset to process accordingly.

For the Clustera MapReduce experiments, the input logical file is composed of a set of concrete files that each contain ~6 million records and are approximately 759 MB in size. In each experiment the number of concrete files composing the logical input file and the number of Reduce jobs is set to be equal to the number of nodes participating in the experiment; since the MapReduce abstract compiler generates one Map job per concrete file, the number of Map jobs, then, was always also equal to the number of nodes participating in the experiment.

After getting up and running with Hadoop, we quickly realized that there were quite a few configuration parameters that we could adjust that would affect the observed MapReduce performance. We evaluated several different configurations of chunk size (from 64 MB to 780MB) and number of concurrent mappers per node. The configuration that gave us the best overall performance used a chunk size of 128MB and two map tasks per node. See Appendix I of [27] for the complete results of this study.

Finally, we ran one additional experiment designed to see how performance of this computation is affected if the input data is hash-partitioned and the scheduler is aware of the partitioning. Since the "vanilla" MapReduce computation model does not distinguish between hash-partitioned and randomly partitioned input files, we could not run this experiment on Hadoop or via the Clustera MapReduce abstract compiler. We could, however, perform the same computation by recasting it as a Clustera-SQL job over a relational table containing the same underlying dataset (partitioned on the aggregation key) in which *project* and *aggregate* take the place of *map* and *reduce*. To ensure comparability across results, the query's project operator outputs the same intermediate values as the map functions of the MapReduce jobs.

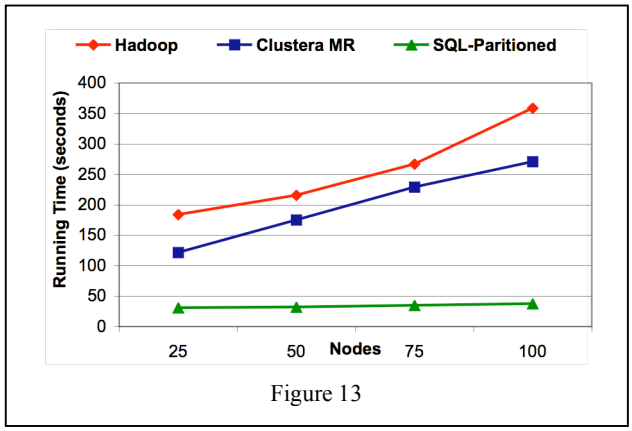

Figure 13

Figure 13 shows the scale-up results we obtained with Hadoop, the Clustera MapReduce computation, and the Clustera SQL computation over hash-partitioned data. The graph plots the number of nodes participating in the experiment (x-axis) against the end-to-end execution time (in seconds) of the experiment (y-axis). The top line shows the observed performance for Hadoop. The next line down shows the observed performance for the Clustera MapReduce experiment and the flat line at the bottom shows the observed performance for the equivalent SQL computation on hash-partitioned data.

As is evident from the graph, MapReduce performance on Clustera and Hadoop are in the same general range. In both systems performance scales roughly linearly as the cluster size

and amount of data to process are increased (more on this follows). The best observed performance was for the computation over hash-partitioned data. The end-to-end performance time is essentially flat across problem sizes implying very good scale-up. This makes sense since the partition-aware scheduler can simplify the computation to a set of independent three-step pipelines – one for each concrete file of the table – and avoid the repartitioning. A key learning from this set of experiments is that, at least for MapReduce workflows with large intermediate data files, it appears that a general-purpose system can deliver performance comparable to a special-purpose MapReduce processing system.

Before moving on to the SQL experiments, it is interesting to look a bit deeper into the Clustera MapReduce numbers to try to understand why execution time scales roughly linearly with problem size. To investigate this, we split the time taken for the Clustera MapReduce computation into 3 components – a Map phase, a Shuffle phase and a Reduce phase. We defined the Map phase to encompass the time from the start of the workflow to the end of the last *split* job. We defined the Shuffle phase to encompass the time from end of the last *split* to the start of the last *merge* job - i.e., the time taken for *all* the intermediate data to be transferred from the mappers to the reducers. We defined the Reduce phase to encompass the time from the start of the last *merge* to the end of the last *reduce* job. The end of the Reduce phase is the end of the MapReduce computation.

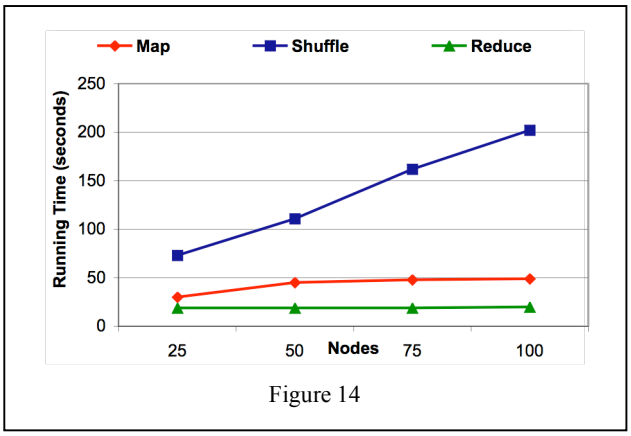

Figure 14

Figure 14 shows the breakdown of the Clustera MapReduce experiments into these three phases. As was the case with the previous graph, the x-axis corresponds to the number of nodes participating in the experiment and the y-axis corresponds to the time spent (in seconds) in a given phase. As this graph shows, the time spent in the Map phase and the Reduce phase remain basically constant across scale-factors. The time spent in the Shuffle phase (i.e., transferring files), however, increases linearly with the problem size and drives the linear growth of the entire computation. Considering that, for this computation, an increase in the input size implies a commensurate increase in the amount of intermediate data generated, this linear growth makes sense.

One important consequence of this linear growth is that intermediate data transfer can become a bottleneck in MapReduce computations - with network bandwidth serving as the limiting factor. In our experimental cluster the gigabit link between the racks has a strong influence on the length of the Shuffle phase of the computation. In the Hadoop MapReduce implementation, the reducers begin pulling intermediate data as soon as the first map task completes – the effect is that Hadoop is able to overlap some of the data transfer time with some of the map task execution time. Contrast this with the current Clustera prototype in which none of the file transfer begins until all of the map jobs have completed. This is an example of a structural difference between the two systems that has the potential to impact performance and scalability. In the near future we plan to extend Clustera to enable this overlapping of data transfer time with map execution time, though by pushing data through a socket rather than by pulling it, and hope to see some performance gains as a result. Of course an even more important structural issue is exposed by the hash-partitioned version of this computation in which the partition-aware scheduler is able to avoid all data transfer costs and, as a result, display the best performance and scale-up properties.

4.3 SQL Scaleup Test

This section presents the results of a series of scale-up experiments we ran to evaluate the performance of SQL jobs. The base data for these experiments consists of the CUSTOMER, ORDERS and LINEITEM tables of the TPC-H dataset. The SQL query we ran was:

```
SELECT l.okey, o.date, o.shipprio, SUM(l.eprice)
FROM lineitem l, orders o, customer c
WHERE c.mkstsegment = 'AUTOMOBILE' and
o.date < '1995-02-03' and l.sdate > '1995-02-03'
and o.ckey = c.ckey and l.okey = o.okey
GROUP BY l.okey, o.date, o.shipprio
```

We ran experiments on 25, 50, 75 and 100 nodes using appropriately scaled TPC-H datasets. For all the datasets, the size of the CUSTOMER, ORDERS and LINEITEM chunks on each node are 23MB, 169MB and 758 MB, respectively

For the Clustera experiments we used the SQL abstract scheduler. The query plan joined the CUSTOMER and ORDERS tables first (because they were the two smaller tables) and then joined that intermediate result with the LINEITEMS table. For the Clustera experiments we also varied the partitioning of the base tables. In one set of experiments the base tables were not hash-partitioned. For these experiments, each concrete file of a base table is first filtered through select and project operators, followed by a repartition according to the join key. Thus, there are four repartitions in the workflow – one for each of the base tables, and a fourth for the output of the first join (CUSTOMER-ORDERS). The final group-by-aggregation does not require a repartition since the final join key is *okey*. In the other set of experiments, the CUSTOMER and ORDERS tables are hash-partitioned by *ckey* and the LINEITEM table is hash-partitioned by *okey*. For these experiments, the base tables do not need to be repartitioned prior to the joins. The output of the CUSTOMER-ORDERS join, though, is partitioned on *ckey*, so it must be repartitioned on *okey* prior to the second join. The following table shows the total amount of intermediate data that is repartitioned in both cases.

Num Nodes	Total Data Shuffled (MB)	
	Partitioned	Unpartitioned
25	77	2122
50	154	4326
75	239	6537
100	316	8757

As was the case with the MapReduce experiments, we also ran the SQL experiments on Hadoop. While joining two tables may not fit "neatly" into the MapReduce paradigm, it can be done. In fact, it is precisely *because* a join does not fit neatly into the MapReduce paradigm that we thought that running the query on Hadoop would be interesting; we hoped that it would help us to understand if translating a SQL query into a series of MapReduce jobs (as opposed to a tree of operators) would have any noticeable impact on performance.

Rather than attempt to write our own MapReduce join code, we used the DataJoin *contrib* package that is included with Hadoop. In a DataJoin job in Hadoop, both join operands are treated as a single logical input. The intermediate data is tagged with the base table name in the Map phase, and the join is enumerated in the Reduce phase with the help of these tags. As with the Clustera-SQL queries, for the DataJoin jobs we joined the CUSTOMER and ORDERS tables first and then joined this intermediate result with the LINEITEMS table before performing the final aggregation. Note that, in addition to tagging the data with the base table name, we also performed the necessary selections and projections in the Map phase. The total amount of intermediate data to be repartitioned during the DataJoin jobs, then, is just slightly larger than what is shown in the un-partitioned column in the table above – the excess is due to the per-record tags required to support the DataJoin. Since Hadoop does not support hash-partitioned files, we only ran un-partitioned DataJoins. For all DataJoin experiments we used the same Hadoop base configuration as described in the Section 4.2 MapReduce experiments.

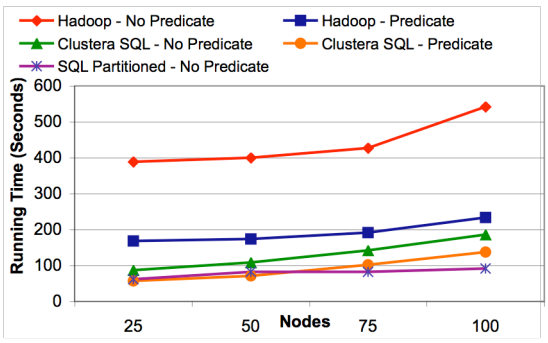

Figure 15

On our generated datasets, the predicate on this query has a selectivity of approximately 50%; this selectivity factor significantly reduces the size of the intermediate files that are generated. We also experimented with another version of the query with no selection predicates to see if altering the selectivity had any impact on performance.

Figure 15 plots the number of nodes in the experiment (x-axis) against the observed run-time (in seconds, y-axis) of the computation. Looking at the right side of the graph, the top line is the DataJoin with no selection predicate, the second line is the DataJoin with a selection predicate, the third line is the Clustera-SQL job on un-partitioned data with no selection predicate, the fourth line is the Clustera-SQL job on un-partitioned data with a selection predicate and the bottom line is the Clustera-SQL job on hash-partitioned data with no selection predicate. The Clustera-SQL job on partitioned data with a selection predicate performed on average about 30% better than the equivalent SQL job with no selection predicate, but we have not displayed it here in order to improve the readability of the figure.

As expected, observed performance and scalability on the hash-partitioned datasets is superior to all other approaches. Again, this would seem to suggest that, given the performance gains available, when implementing a parallel computation system it is probably worth the effort to enrich the data-model and scheduler to support and take advantage of partitioned data. For both of the Clustera SQL jobs over un-partitioned data, performance seems to scale roughly linearly with an increase in the problem size. Similar to the Clustera MapReduce results, the linear factor can be traced back to the data transfer when tables are repartitioned. Recall that in the current Clustera prototype none of the "merges" of a repartitioning are scheduled until all of the corresponding "splits" have completed. The result is that there is a spike in network activity when the merges are scheduled which causes the link between the racks to saturate (and become a bottleneck) leading to the linear growth in the data-transfer time and, by extension, the overall execution time. Compare this linear growth to that of the DataJoin job on un-partitioned data with a selection predicate (second line). This case still exhibits a linear component to the growth, but the linear growth here is damped relative to that of the Clustera SQL jobs just discussed; this dampening most likely occurs because, as described previously, the underlying Hadoop MapReduce jobs are able to overlap some of the data transfer time with some of the map task execution time. Again, in the near future we plan to apply this lesson by extending Clustera to support overlapping data-transfer time and execution time. Finally, the performance of the DataJoin with no selection predicate is interesting to consider. The run-time for the DataJoin with no selection predicate ranges from approximately 2.3 (25-nodes) to approximately 2.6 (100-nodes) times longer than the run-time for the DataJoin with a selection predicate. Compare this to the spread for the Clustera SQL jobs on un-partitioned data that ranges from approximately 1.4 (100-nodes) to approximately 1.5 (25-nodes) times longer. It is not entirely clear why the DataJoin spread is so much larger. One possible contributing factor is that perhaps less of the data-transfer time is overlapped with map task execution time. Another possible factor is that implementing a join in the MapReduce framework inherently incurs additional sorting overhead that can be avoided if a good plan using a hash-join is available. Unfortunately, despite some of their shared goals, the systems are sufficiently different that it appears that an in-depth investigation beyond the scope of this study is needed to draw firm conclusions about the performance impact that the underlying parallel execution model has on join implementation.

4.4 Blast Workflow Speedup test

Clustera also can run arbitrary workflows with user-defined data and programs. This section examines the execution of the BLAST scientific workflow. BLAST is a sequence alignment program that searches a file of well-known proteins for similarities with a new sequence of proteins. A BLAST DAG consists of two jobs, *blastall* and *javawrap*. *Blastall* takes a sequence of acids *seq* and performs sequence alignment using the *nr_db* files, which document known proteins. *Blastall* produces a *seq,blast* file which is then processed by the *javawrap* program to produce *csv* and *bin* files for later use. A typical BLAST workflow consists of several such BLAST DAGs. Each DAG may operate on a different set of sequences, but all of them use

the same *nr_db* and *all_java.tar* files. We ran a workflow of 1000 BLAST DAGs on 25, 50, 75 and 100 machines to test the speedup obtained. One copy of the input data (including the *nr_db* files and *all_java.tar*) is placed on either side of the switch. The mean execution times for *blastall* and *javawrap* are 207s and 6s.

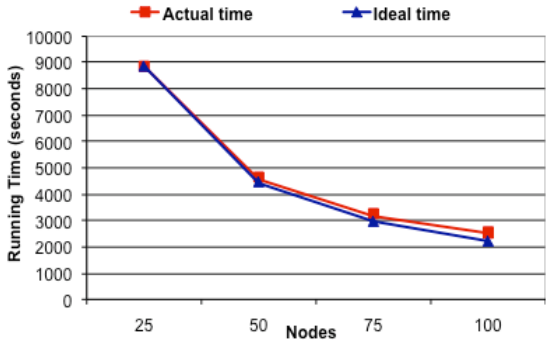

Figure 16

As we can see from Figure 16, as the number of nodes is increased, the actual speed-up obtained deviates slightly away from the ideal line. The reason is as follows. Before a *blastall* job can begin executing on a node, all of its required input data (the *nr_db* files) must be transferred to that node. As the number of nodes is increased, the amount of data that is transferred across the network increases, which delays the startup of the first *blastall* job on each node. Note that all of the file transfer time is spent before the execution of the first job. Also, once the first jobs on each node completes, the input data is essentially replicated throughout the cluster, and subsequent jobs do not incur any file transfer cost. This suggests that for workflows containing a large number of jobs with common input data, file transfer becomes a bottleneck as the number of nodes used to run the workflow is increased – for a given replication factor, when the number of nodes used is doubled, the data transfer required more than doubles. One way to alleviate this problem is to start with a high degree of replication for this input data.

4.5 Application Server Throughput Tests

One metric for evaluating the scalability of a workflow management system is the number of jobs that the system can process per second. Under full load, the average throughput demand is defined as the ratio of the number of nodes to the average length of a job. For example, a system with 1,200 nodes subject to a workload consisting solely of 20-minute jobs must be capable of a job throughput rate of at least one job per second. One important implication of this is that the overall system throughput is affected not only by the time it takes to make scheduling decisions and start up jobs, but also by the efficiency with which the system can perform any necessary post-execution processing. Post-execution tasks include recording historical information about the job, recording accounting information, and removing the job from the queue.

To evaluate the scalability of the Clustera server, using 100 nodes we configured each node to run up to two simultaneous single-job pipelines (since each node has two cores)[1]. We then pre-loaded the system with a number of identical, fixed-length jobs with lengths that varied from a maximum of 200 seconds down to a minimum of four seconds in order to cover a range from 1 job per second all the way up to 50 jobs per second.

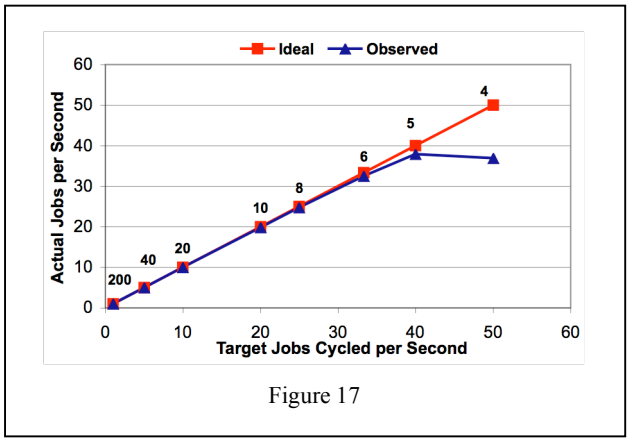

Figure 17

Figure 17 plots the number of jobs cycled per second against the targeted throughput rate for the experiment. The top line shows the ideal throughput rate while the bottom line shows the observed results. The labels above the top line show the job length for that experiment. Since our cluster was configured to run 200 jobs concurrently the targeted throughput rate is simply 200 (the number of concurrent jobs) divided by the job length; the 200 second jobs correspond to a target of one job per second whereas the four second jobs correspond to a target of 50 jobs per second. For the jobs that were six seconds or longer, we observed the server achieving throughput rates very close to the ideal. For the five- and four-second jobs the observed rate is below the ideal.

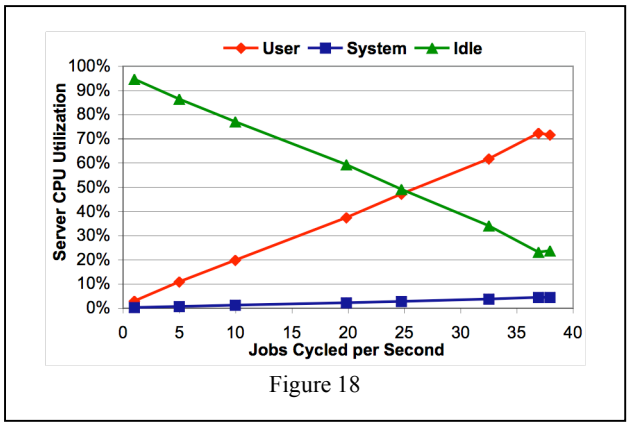

Figure 18

Figure 18 shows a plot of the server's CPU cycle consumption as a function of the number of jobs cycled per second. For each experiment we calculated the average throughput rate, excluding the ramp up and ramp down time. Looking at the left side of the graph, the bottom line (square data points) in Figure 20 is "System" usage (cycles spent executing in "kernel" mode), the middle line (diamond data points) is "User" usage (cycles spent doing actual computation) and the top line (triangular data points) is idle cycles (spare computational capacity). These three categories sum up to approximately 100%; the time spent handling interrupts or waiting on IO was negligible and is not plotted here. As Figure 18 shows, the server still had at least 20% of its cycles to spare during all experiments.

[1] We enforced the single-job pipeline limitation in order to make the results comparable with those published in [26].

One striking feature of Figure 18 is the apparent linear growth in cycle usage in response to increases in targeted throughput across experiments. This pattern changes at the right side of the chart as the observed throughput peaks just shy of 38 jobs cycled per second. Interestingly enough, the right-most data point actually corresponds to the five-second job experiment (~37.9 jobs cycled per second with ~23.8% idle cycles remaining) and the second-from-the-right data point corresponds to the four-second job experiment (~36.9 jobs cycled per second with ~23.2% idle cycles). The server apparently saturates at around 38 jobs per second after which additional demand causes interference on the server leading to reduced performance.

The observed throughput rates presented here are an improvement over previously published results for CondorJ2 (a precursor to Clustera) and Condor [26]. This improvement comes despite the fact that Clustera is required to manage not only the same job, machine and configuration information managed by CondorJ2 (and Condor), but also to manage the logical and physical file information required to support data-aware scheduling.

5. Conclusions and Future Directions

We are currently witnessing the early stages of the "cloud computing" revolution, in which large clusters of processors are exploited to perform various computing tasks on an "as needed" basis. To date the database community has played a minor role in this revolution in that (a) large-scale parallel database systems use a model of dedicated, single-use clusters very different from that adopted by cloud computing, and (b) cluster management systems for high-throughput and data intensive computing use database technology superficially if at all.

Our work on Clustera is an early step toward increasing the role of database systems in cloud computing in two ways. First, we have shown that cluster management is an ideal application for modern relational database system technology. Second, we have shown that a generic cluster management system like Clustera has potential as a platform upon which to execute massively parallel SQL queries. The potential in this second direction is huge – it opens the door to integrated systems that can run SQL queries on the same platform used to run other data intensive and compute-intensive applications. There is still a great deal of challenging research required to make such systems a reality. We hope the database community will participate in this research, enabling both parallel database systems and cloud computing to benefit from this opportunity.

6. Acknowledgements

This work was funded by National Science Foundation Award SCI-0515491. We would also like to thank the anonymous reviewers for their feedback and suggestions that guided us in improving the content and presentation of this paper.

7. References

[1] Litzkow, M., Livny, M., and M. Mutka, "Condor – A Hunter of Idle Workstations", Proc. of the 8th ICDCS Conf., 1988.

[2] Dean, J. and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Proc. of the 6th OSDI Conference, Dec. 2004

[3] Needham, R. M. and A. J. Herbert. 1982. The Cambridge Distributed Computing Systems, Addison-Wesley, MA.

[4] DeWitt, D., Finkel, R. and M. Solomon, "The Crystal Multicomputer: Design and Implementation Experience," IEEE TSE, Vol 13. No. 8, 1987.

[5] IBM, "Tivoli Workload Scheduler LoadLeveler v3.3.2 Using and Administering," April 2006.

[6] Platform Computing Corporation, "Administering Platform LSF", Platform Computing Corporation, Feb. 2006.

[7] Urban, A (Ed.), "Portable Batch System Administrator Guide," Altair Grid Technologies, April 2005.

[8] Sun Microsystems, Inc., "N1 Grid Engine 6 Administration Guide," June 2004.

[9] Microsoft Corporation, "Windows Compute Cluster Server 2003 Reviewers Guide," May 2006.

[10] United Devices, "Grid MP Platform Version 4.1 Application Developer's Guide," 2004.

[11] Schuster, S., Nguyen, H., Ozkarahan, and K. Smith, "RAP.2 – An Associative Processor for Databases," Proc. of the 5th ISCA Conference, 1978.

[12] Su, S. and G. Lipovski, "CASSM: A Cellular System for Very Large Databases," Proceedings of the 1st VLDB Conference, Framingham, MA, 1975.

[13] DeWitt, D., "DIRECT - A Multiprocessor Organization for Supporting Relational DBMS," IEEE Trans. on Computers, Vol. C-28, No. 6, June 1979.

[14] Stonebraker, M., "Muffin: A Distributed Database Machine," Proc. of the 1st ICDCS Conference, Oct. 1979.

[15] DeWitt, D., Gerber, B, Graefe, G., Heytens, M., Kumar, K. and M. Muralikrishna, "Gamma–A High Performance Dataflow Database Machine," Proc. of the 1986 VLDB Conf.

[16] Teradata, "DBC/1012 Data Base Computer Concepts & Facilities," Teradata Corp. Document No. C02-0001-00, 1983.

[17] http://hadoop.apache.org/

[18] Isard, M., Budiu, M, Yu, Y., Birrell, A., and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007.

[19] Kini, A., Shankar, S., Naughton, J., and D. DeWitt, "Database support for matching: limitations and opportunities," Proc. of the 2006 SIGMOD Conference, 2006.

[20]RedHat Enterprises, "JBoss Enterprise Application Platform," http://www.jboss.com/products/platforms/application

[21] Mutka, M. and M. Livny, "Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network," In Proc. 7-th Int. Conf. on Distr. Comp. Systems, 1987.

[22] DeWitt, D. and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," CACM Vol. 34, No. 6, 1992.

[23] http://www.cs.wisc.edu/condor/dagman/

[24] Shankar, S. and DeWitt, D. J. 2007. "Data driven workflow planning in cluster management systems," Proceedings of the 16th High Performance Distributed Computing Conf, 2007.

[25] Olson, C., Reed, B., Srivastava, U., Kumar, R. and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing, Proceedings of the 2008 SIGMOD Conference.

[26] Robinson, E., and D. DeWitt, "Turning Cluster Management into Data Management: A System Overview," Proceedings of the 2007 CIDR Conference, Asilomar, CA.

[27] Computer Sciences Tech. Report, Univ. of Wisconsin, #1637, http://www.cs.wisc.edu/techreports/2008/TR1637.pdf

8. Appendix

Data management is an important aspect of Clustera's overall system architecture. Reflecting this, Section 3 contained a general overview of the approach sufficient for understanding the content in the body of the paper. The purpose of this appendix is to provide some additional details on Clustera's data management infrastructure for interested readers. Section 8.1 has a brief overview of the basic concepts and terminology. Section 8.2 discusses issues related to file formats and schema management in greater detail. Section 8.3 discusses partitioning and our plans for incorporating indexing into the infrastructure.

8.1 Concrete Files, Logical Files and File Access Patterns

As explained previously, a **concrete file** is the atomic unit of storage in Clustera. The physical instantiation of a concrete file is a single operating system file. A **logical file**, in comparison, is a higher-level abstraction that does not have a direct physical instantiation. In its most basic form, a logical file is simply a collection of concrete files that the user can treat as a single logical unit (hence the name logical file) for manipulation and workflow specification. Note that, if necessary, a logical file can specify an ordering over the constituent concrete files. Note also that since the logical file itself is purely conceptual, it is possible for a single concrete file to "belong to" multiple different logical files simultaneously.

Given the notion of a logical file as a set of concrete files, it is natural to parallelize, for example, the execution of a MapReduce computation by running one map instance for each concrete file composing the logical input file. This works because MapReduce (like SQL) is a record-oriented paradigm so it never needs to view the entire input data set as a single unit. Recall, though, that Clustera also supports running arbitrary workflows that contain non-modifiable, user-provided executables. Since these executables often have file-oriented, rather than record-oriented, semantics they must be able to view the (logical) input file as a single unit. Setting the degree of partitioning for the logical file to one is a possible approach to resolving this issue. However, this approach will not always work. One common case in which this approach will fail is if the entire file will not fit on a single node. Another common case is if the user specifies a pre-existing logical file that consists of multiple concrete files as the input to one of these arbitrary executables.

To support logical files used in these ways, Clustera offers a Logical File Translator (LFT) interface. The LFT abstracts away the physical details (i.e., the mapping to, and optional ordering over, concrete files) of a logical file and provides users with a single file "view" they can read using the standard POSIX interface. To achieve this, the first thing the LFT does is contact the application server to obtain details on the relevant logical file - i.e., the names and locations of the constituent concrete files. Then, as the application needs them, the necessary concrete files are fetched. On Linux this is implemented through a device driver built on top of FUSE (Filesystem in USErspace).

For MapReduce and SQL computations, the LFT is generally unnecessary given that the operators are automatically parallelized across the relevant concrete files. For arbitrary workflows, however, it is impossible to know (without user input) whether or not a given "job" can be parallelized across the concrete files or not. For these workflows the default assumption is that the job **cannot** be parallelized in this way and the LFT is employed. We have, however, recently implemented some experimental extensions to the workflow specification model that permit the user to provide "hints" to the Workflow Abstract Scheduler when a computation can be parallelized across particular, user-specified, logical files. The scheduler can use these hints to optionally perform an automatic parallelization under certain conditions. This work is still in a very early stage, though it appears promising.

One final point to note with respect to concrete file management relates to the automatic piping of intermediate data between concrete jobs described in Section 3. Recall from that section that Clustera will attempt, when possible, to avoid disk I/O operations by transparently routing intermediate data in memory from one concrete job to another when those jobs are co-scheduled for execution on the same node. This type of piping, of course, assumes both sequentially generated output and sequentially accessed input. For MapReduce and SQL operators we know exactly which operators for which these assumptions are valid (e.g., map, select, project, etc…) or invalid (e.g., hash join, etc…). This means that, when the concrete jobs are created and inserted into the system, we know what intermediate data can be "tagged as pipeline-able" and what cannot. For arbitrary workflow executables, however, the assumption is that these assumptions are violated unless expressly indicated by the user as part of the workflow specification. Thus, for arbitrary workflows, data will in general not be pipelined unless the user explicitly specifies that the output is generated sequentially and the input file is consumed sequentially.

8.2 File Formats and Schemata

Another interesting topic to discuss is what, if anything, Clustera assumes about data file structure. The answer to this question depends on the level of the system in question. At the concrete level, Clustera assumes nothing about file structure. Both concrete files and concrete jobs are essentially black boxes. All that the concrete job scheduler knows about a given concrete job is what its executable file is, what input files it consumes, what output files it generates and whether its inputs and outputs are accessed/generated sequentially.

At the logical level, however, the amount of knowledge about the internal file structure depends on the specification of the logical file itself. For logical files specified as containing "binary" data, there must be an ordering provided over the concrete files and the assumption is that an LFT will be used (possibly in conjunction with a record reader) to access the file. For logical files specified as "record" data, a record delimiter is given when the file is stored in the system and is maintained as a property of the logical file. Because the record delimiter is known when the file enters Clustera, records need not span concrete files; this makes it possible to avoid using an LFT to access records. Note that "record" data need not specify a schema. Finally, "record" data that does specify a schema is managed as a relational table as described in Section 3.5 (i.e., a logical file plus the associated schema information). For all relational tables, the schema information includes the record attributes and the record format. For partitioned tables the schema information also includes the mapping between table partitions and concrete files. The SQL abstract scheduler uses all of this information to take an abstract workflow description specified in terms of tables and convert it into an executable workflow specified in terms of concrete files.

8.3 Partitioning and Indexing

Currently, Clustera only supports partitioning over relational tables. It should be possible to support partitioning over non-relational "record" data via a user-supplied partitioning function (e.g., a frequently used Map function), though we have yet to pursue any research in that direction. Binary data, by its nature, appears to be an unlikely candidate for partitioning.

The current Clustera prototype does not yet support indexing over relational tables. We do plan to implement support for indexing in the future. The indexes themselves would likely be physically instantiated as concrete files with all of the relevant meta-data stored in the central database alongside the rest of the schema information. As was the case with partitioning, it seems as if it should be possible to support indexing over non-relational "record" data via a user-supplied function. If we implement this functionality one interesting area of future research would be to explore how an abstract scheduler could take advantage of such indices.