

# GridBatch: Cloud Computing for Large-Scale Data-Intensive Batch Applications

Huan Liu, Dan Orban  
Accenture Technology Labs  
{huan.liu, dan.orban}@accenture.com

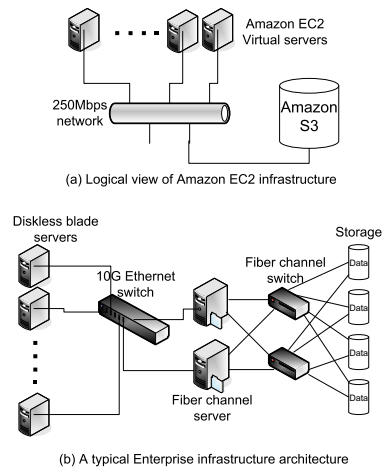
## Abstract

To be competitive, Enterprises are collecting and analyzing increasingly large amount of data in order to derive business insights. However, there are at least two challenges to meet the increasing demand. First, the growth in the amount of data far outpaces the computation power growth of a uniprocessor. The growing gap between the supply and demand of computation power forces Enterprises to parallelize their application code. Unfortunately, parallel programming is both time-consuming and error-prone. Second, the emerging Cloud Computing paradigm imposes constraints on the underlying infrastructure, which forces Enterprises to rethink their application architecture. We propose the GridBatch system, which aims at solving large-scale data-intensive batch problems under the Cloud infrastructure constraints. GridBatch is a programming model and associated library that hides the complexity of parallel programming, yet it gives the users complete control on how data are partitioned and how computation is distributed so that applications can have the highest performance possible. Through a real client example, we show that GridBatch achieves high performance in Amazon's EC2 computing Cloud.

## 1 Introduction

To stay ahead of competition, Enterprises are collecting and analyzing large amount of data to derive business insights [3], hopefully in real-time or near real-time. For example, Enterprises want to understand their customers better in order to develop a better marketing plan, or examine their supply chain to look for opportunities to improve efficiency, or analyze sensor data to predict machines failure and prevent revenue lost earlier.

The trends on data growth and processor speed improvement suggest that some form of parallelization is



**Figure 1. Comparison of infrastructure architectures.**

required to analyze the data. First, due to the need to derive better business insights and to improve the accuracy, the amount of data we are collecting is growing at an exponential rate [8] [1]. Second, uniprocessor speed has stopped exponential growth since roughly 2002 [9]. Many applications are already taking hours to days of computation time to process the data today, and it will take much longer over time as the data volume grows. Such long processing delays put Enterprises at competitive disadvantages since they cannot react fast enough.

There is a strong desire to analyze the data in the emerging Cloud Computing infrastructure (Fig. 1(a)) because of its strong value propositions. A Cloud Computing infrastructure, such as Amazon's Elastic Computing Cloud (EC2), provides computation capacity, typically in the form of virtual machines, to end users on-demand from remote locations in the Internet. Cloud presents many value propositions which include: provision on demand instead of waiting for the long

procurement lead time; pay as you go instead of paying large capital cost up front; pay only for what you need; and lower cost not only because of the elimination of support and maintenance cost, but also because a Cloud provider could pass on portions of its savings as a result of its economy of scale. In addition, because of its scale and expertise, Cloud could provide a much more reliable and secure infrastructure than most Enterprises could afford. For example, Amazon have multiple data centers and each data center also has multiple backup power generators. Amazon also has strong security measures to guard against not only physical security breach but also hacker attacks<sup>1</sup>.

There are challenges both in the programming model and in the underlying infrastructure to analyze the data in the Cloud.

From the programming model perspective, parallel programming is both time-consuming and error-prone. The Enterprise analytics applications, as well as a large class of batch applications, have obvious parallelism at the data level. It is straightforward to partition the job into many independent parts, and process them in parallel. However, it is in general not straightforward to implement a parallel application for several reasons. First, some forms of communication, coordination and synchronization are required between the machines, but they are not trivial to implement correctly. Second, the inherent asynchronous nature of parallel programs makes it hard for the programmers to reason about the interactions between all machines. Compared to sequential programs, there are many more scenarios that need to be examined, making it hard to guarantee program correctness in all cases. Last, there is still not an effective debugging tool. Because of the complex interactions, many bugs are transient in nature, which are hard to reproduce. In addition, it is hard to step through some code when there are many threads running on many machines. The difficulty in implementation translates into higher development cost and longer development cycle. Worse yet, the same programming effort often has to be repeated for each new project.

From the infrastructure perspective, Cloud Computing presents additional challenges, both in network bandwidth and hardware reliability. Fig. 1 shows a comparison between a traditional Enterprise infrastructure architecture and a Cloud infrastructure architecture. For ease of management, Enterprise infrastructure (Fig. 1(b)) typically has a large centralized storage, and multiple fiber channel links and switches to provide both high bandwidth and fault tolerance. Multiple high-performance high-reliability

blade servers are directly connected to a high bandwidth network, such as a 10G Ethernet, which is also connected directly to the Storage Array Network (SAN). The Enterprise infrastructure is carefully dimensioned to guarantee a sufficiently high bandwidth between the servers and the storage. In comparison, a Cloud infrastructure, such as Amazon's (Fig. 1(a)), is based on commodity server hardware and commodity network switches. For example, Amazon only promises 250Mbps network bandwidth, which is much smaller than what a typical Enterprise application requires (although in practice, the network bandwidth could be higher, especially on their larger virtual machine instances where the contention for the physical hardware is lower). Worse yet, Since no topology information is known, all servers in the Cloud could be sharing the same 250Mbps, e.g., when all servers are connected through a single 250Mbps link to the Amazon S3 storage. In addition, commodity servers are less reliable. Even though Cloud provider's data centers are unlikely to fail because of the various backup mechanisms, individual commodity hardware does fail often due to component failures. This is part of the reason why Amazon only have a 99.9% SLA on S3 data storage and none yet on the EC2 servers.

To overcome the lower network bandwidth in Cloud, Enterprise applications have to be re-architected to exploit the local bandwidth from the local hard disks. When we add a new server to the processing pool, we do not increase the network bandwidth, since Ethernet topology is always a tree and the bottleneck link determines the network capacity. However, we do increase the total local disk I/O bandwidth. To scale bandwidth, applications should be architected to make efficient use of the increased I/O bandwidth through data localization.

To overcome the hardware reliability problem, applications should be architected to tolerate hardware failures, i.e., treat hardware failures as a normal event and recover gracefully instead of treating it as a catastrophe. This means that not only data should be replicated but also means that the applications should be able to restart computations when the underlying hardware fails.

In this paper, we present a system which allows programmers to easily convert a high-level design into the actual parallel implementation in a Cloud-Computing-like infrastructure. Our goal is not to help the programmers find parallelism in their applications. Instead, we assume that the programmers understand their applications well and are fully aware of the parallelization potentials. Further, the programmers have thought through on how to break down the application

---

<sup>1</sup>private communication

into smaller tasks and how to partition the data in order to achieve the highest performance. But, instead of asking the programmers to implement the plan in detail, we provide a library of commonly used “operators” (a primitive for data set manipulation) as the building blocks. All the complexity associated with parallel programming is hidden within the library, and the programmers only need to think about how to apply the operators in sequence to correctly implement the application.

It is worth noting that even though GridBatch is specifically designed for the Cloud infrastructure, it would run equally well in a traditional Enterprise infrastructure, a Cluster environment or a Grid environment where bandwidth is more plentiful. In those environments, GridBatch is still an appealing solution compared to alternatives because of its simple parallel programming abstraction, which directly translate into higher programmer productivity.

To have large performance gains, the applications must have large amount of parallelism at the data level (e.g., large tables where each row is independent of each other). Among these, our framework is specifically targeted at analytics applications, whose unique characteristics require special operators. Analytics applications are often interested in collecting statistics from the large data set, such as how often a particular event happens. They often involve correlating data from two or more different data sets (i.e., table joins in database terms).

## 2 Related work

Traditional databases can be used to implement many analytics applications we are targeting. Unfortunately, they do not scale for large data sets for two reasons.

1. First, databases present a high level SQL language with the goal of hiding the execution details. Although easy to use, this high level language forces users to express computation in ways that are not performance efficient. Sometimes, the most efficient method would only scan the data set once, but when expressed in SQL, several passes are required (see the example in Yahoo Pig’s documentation [12]). Even though techniques have been developed to automatically optimize query processing, the performance is still far from what can be achieved by a programmer who understands the application well.
2. Second, databases run well in a traditional Enterprise infrastructure architecture where network

bandwidth is plentiful, but they will suffer badly in a Cloud Computing infrastructure because they are not able to exploit the local disk I/O bandwidth. Even though most database systems, including Oracle’s commercial products, employ sophisticated caching mechanisms, many data accesses still traverse the network, consuming precious bandwidth.

Furthermore, database is limited by the capabilities of SQL. In comparison, we allow any user-defined function to be used.

Google’s MapReduce [4] inspired our work. It shares many similarities with our system: it tries to utilize local disk bandwidth; it replicates data and restarts computation when hardware fails; it can handle large data sets; and it eases the pain of parallel programming. However, it is designed for web applications, such as counting word frequency, computing reverse web links; thus, it does not have the necessary optimization for analytics applications, in particular, the level of control we provide on data storage and movement. Also, it lacks many capabilities that are built-in in our system, such as the Join, Cartesian and Neighbor operators (will be described in Sec. 5), which are frequently used in Analytics applications.

Analytics applications differ from web applications in several regards:

- Analytics applications typically deal with structured data, whereas, web applications frequently deal with unstructured data.
- Analytics applications often require cross referencing information from different sources (e.g., different tables in database terms).
- Analytics applications are typically interested in much fewer statistics than web applications. For example, a word counting application would require statistics for all words in the vocabulary, whereas, an analytics application may be only interested in a few statistics, such as how many products were sold.

Both Yahoo Pig [12] and Google Sawzall [10] focus on the same analytics applications as we do, but both took an approach similar to database systems. They present to the users a higher level programming language, making it easy to write analytics applications. However, because the users are shielded away from the underlying system, they could not optimize the performance to the fullest extent. In fact, both Yahoo Pig and Google Sawzall build on top of Google’s MapReduce, which lacks a few fundamental operators for analytics applications.

Microsoft Dryad [6] is another system similar to Google’s MapReduce and ours. Compared to MapReduce, it is a much more flexible system. Users express their computation as a dependency graph, where the arcs are communications and the vertices are sequential programs. Their solution is very powerful because all parallel computations can be expressed as dependency graphs. Unfortunately, its generality also makes it harder to use, as evidenced by the many systems that are built on top of Dryad (SSIS, DryadLINQ, distributed shell).

Another recent system, Map-Reduce-Merge [2], extends the MapReduce framework to include the capability to merge two data sets, which is similar to the capability of the Join operator that we will describe. However, like MapReduce, it does not give programmers direct control on how data are distributedly stored and it does not have the full capabilities offered by our family of operators (such as the Cartesian and Neighbor operators).

Our system differs from these prior work in two regards.

1. We provide a family of operators, each for a common task often found in analytics applications. We describe several operators that we have already implemented, and more operators will be introduced as we gain further understanding of more analytics applications and be able to abstract out the fundamental operations they have to perform. Again, the goal is not to replace the thought process of a programmer, but rather to build a comprehensive family of operators, so that a programmer could arbitrarily compose them in any order to implement arbitrarily complex analytics applications.
2. We extend the Google File System [5] to allow the programmers to control where data are stored in order to optimize data locality. This is especially important in a Cloud Computing infrastructure where the network bandwidth is more limited compared to a dedicated infrastructure. For example, Amazon EC2 only promises 250Mbps bandwidth, whereas, 1Gbps bandwidth is common in Google’s internal infrastructure.

### 3 A sample application

The GridBatch system is currently under evaluation with one of our clients. We will use their applications to describe and demonstrate the capabilities of the GridBatch system. Due to confidentiality concerns, we only describe their applications at an abstract level.

The client has a large logistics operation. As part of the operation, a large number of items are shipped through the system, each item has an associated unique barcode. For efficiency of shipping, many items are typically packaged together in a container first. The barcode not only uniquely identifies the item, but also encodes meta data associated with it, such as the origin and destination, the container ID that it was shipped in, and the source which generated the barcode. The barcode is scanned once at the origin and once again at the destination.

The client is interested in collecting many analytics to improve their operation efficiency.

- The client wants to identify duplicates in barcodes. Since each item should have a unique barcode, a duplicate (two items having the same barcode) indicates potential problems in the operation.
- For auditing purposes, the client wants to identify all items with barcodes that are generated by a set of sources.
- Normally, all items within the same container should have been scanned consecutively. The client wants to find all occurrences where items from different containers are interlaced, since they indicate potential problems in operation.
- The client is interested in detecting business rule violations. A list of business rules are given, and every item should be checked against all possible rules to detect violations.

## 4 The GridBatch system

There are two fundamental data types in GridBatch: table or indexed table (borrowed from database terminology). A table contains a set of records (rows) that are independent of each other. All records in a table follow the same schema, and each record may contain several fields (columns). Indexed table is similar to table except that each record also has an associated index, where the index could simply be one of the fields or other data provided by the user.

Table is analogous to the vector concept in vector or SIMD (Single Instruction Multiple Data) machine or the stream concept in stream processors [7] in Computer Architecture. A table (or a vector/stream) implies that all records within it are independent of each other, and hence, they can be processed in parallel. Vector/Stream allows a computer architect to design specialized hardware which takes one instruction, but applies it to all records in a vector/stream. Similarly,

table allows us to design a software system which can process the records in parallel across many machines in the Cloud.

The GridBatch system consists of two pieces of related software components: the distributed file system (DFS) and the job scheduler.

#### 4.1 The distributed file system

DFS is an extension of GFS [5] that supports a new type of file (fixed-num-of-chunk files described below).

DFS is responsible for managing files and storing them across all nodes in the system. A large file is typically broken down into many smaller chunks, and each chunk may be stored on a separate node. Among all nodes in the system, one node serves as the name node, and all other nodes serve as data nodes.

The name node holds the name space for the file system. It maintains the mapping from a DFS file to the list of chunks, including which data node a chunk resides on and the location on the data node. It also responds to queries from DFS clients asking to create a new DFS file, as well as allocates new chunks for existing files or returns chunk locations when DFS clients ask to open an existing DFS file. A data node holds chunks of a large file. It responds to DFS client requests for reading from and writing to the chunks that it is responsible for. A DFS client first contacts the name node to obtain a list of chunk locations for a file, then it contacts the data nodes directly to read/write data.

DFS stores two types of files: fixed-chunk-size files or fixed-num-of-chunk files. Fixed-chunk-size files are the same as the files in GFS. They are broken down into chunks of 64MB each. When new records are written, they are appended to the end of the last chunk. When the last chunk reaches 64MB in size, a new chunk is allocated by the name node.

For indexed table, we introduce another type of files: fixed-num-of-chunk files, where each file has a fixed number of chunks (denoted as  $C$ , defined by the user) and each chunk could have arbitrarily large size. When a DFS client asks for a new file to be created, the name node allocates all  $C$  chunks at the same time and returns them all to the DFS client. Although the user can choose  $C$  to be any value, we recommend a  $C$  should be chosen such that the expected chunk size (expected file size divided by  $C$ ) is small enough for efficient processing, e.g., less than 64MB each.

Each fixed-num-of-chunk file has an associated partition function which defines how data should be partitioned across chunks. The DFS client submits the user-defined partition function (along with the param-

eter  $C$ ) when it creates the file, which is then stored by the name node. When another DFS client asks to open the file later, the partition function is returned to the DFS client, along with the chunk locations.

When a new data record needs to be written, the DFS client calls the partition function to determine the chunk number(s), then it appends the record to the end of the chunk(s).

The user-defined partition function takes the form:

```
int[] partitionFunc(Index  $x$ )
```

$x$  is the index for the record to be written. The partition function applies a hash function to convert the index into one or more integers in the range of 1 to  $C$ , which indicates which particular chunk(s) the record should be stored at. In most cases, one integer is returned. However, if the user desires to make the data locally available to more nodes, the partitionFunc could return an array of integers. For example, the user may desire to have a local copy of the data on all  $C$  chunks, then the user can design the partition function to return a list of all integers from 1 to  $C$ .

Typically,  $C$  is set to be much larger than  $N$ , the number of machines in the system. The mapping from  $C$  to  $N$  is fixed (i.e., data corresponding to a particular chunk number for all indexed tables are on the same machine) and it is prescribed by a system level lookup table, which is maintained at the name node. Such translation is necessary in order to support dynamic change of the cluster size. When old machines leave (possibly because of failures) and when new machines join, GridBatch can automatically re-balance the storage and workload.

We introduce the fixed-num-of-chunk file type because analytics applications that we are targeting are different from web applications. Web applications (word count, reverse web link etc.) have a large amount of unstructured data which work well with fixed-chunk-size files. In contrast, large analytics applications, such as data warehousing, have a large amount of structured data. For efficient processing, data partitioning is commonly used to segment data into smaller pieces (e.g., database partitioning in any modern database systems). If fixed-chunk-size files are used for analytics applications, constant data shuffling is required whenever a new analytics application starts.

Similar to GFS, all data chunks are replicated several times across the nodes in the system. When a machine fails, no data is lost and the system will adjust itself to re-balance the storage and workload. In GFS, the backup chunk is stored on a randomly chosen node, and the same backup chunk for the same chunk (e.g., the first backup chunk for chunk 1) for two different

files could be stored on two different nodes. For fixed-chunk-size files, we maintain the same backup scheme. However, for fixed-num-of-chunk files, we fix the mapping from backup chunks to nodes, e.g., the first backup chunks for chunk  $i$  for two different files are always stored on the same node. When a node fails, we can simply change the system-wide mapping table so that chunk  $i$  is pointing to the backup node and locality will be preserved.

## 4.2 The job scheduling system

The job scheduling system includes a master node and many slave nodes. The slave node is responsible for running a task assigned by the master node. The master node is responsible for breaking down a job into many smaller tasks as expressed in the user program. It distributes the tasks across all slave nodes in the system, and it monitors the tasks to make sure all of them complete successfully.

In general, a slave node is often a data node. Thus, when the master schedules a task, it could schedule the task on the node which holds the chunk of data to be processed. By processing data on the local node, we save on precious network bandwidth.

## 5 GridBatch operators

GridBatch does not attempt to help a programmer reason the best approach to program an application. Instead, it aims to provide a set of commonly used primitives, called operators, which the programmer can use to save on programming efforts. The operators handle the details of distributing the work to multiple machines, thus the user should not need to worry about parallel programming. Instead, the user just needs to apply a set of operators sequentially, just as if writing a traditional sequential program.

GridBatch extends the capabilities of Google's MapReduce system. MapReduce could be considered as two separate operators: Map and Reduce. The Map operator is applied to all records in a file independent of each other; hence, it can be easily parallelized. The operator produces a set of key-value pairs to be used in the Reduce operator. The Reduce operator takes all values associated with a particular key and applies a user-defined reduce function. Since all values associated with a particular key have to be moved to a single location where the Reduce operator is applied, the Reduce operator is inherently sequential.

GridBatch breaks down MapReduce into elementary operators, and in addition, introduces additional operators. GridBatch currently consists of the following

operators: Map, Distribute, Recurse, Join, Cartesian and Neighbor.

### 5.1 Map operator

The Map operator applies a user-defined function over all records of a table. A sample pseudo-code for the user-defined function is as follows:

```
mapFunc(Record  $x$ ):
  // Apply necessary processing on Record
  //  $x$  to generate Record  $y$ 
  .....
  EmitResult(Table  $Y$ , record  $y$ )
```

Record  $x$  is one of the records in Table  $X$  to which the Map operator is applied. Within the user-defined map function, the user can do any custom processing over record  $x$ . At the end, the user-defined function could generate one or more records for one or more tables. In the example, we generated one new record  $y$  for Table  $Y$ .

The user would invoke the Map operator as follows:

```
Map(Table  $X$ , Func mapFunc)
```

The first argument specifies to which table this Map operator is applied and the second argument specifies the user-defined function.

Many applications need to process records independently. Using MapReduce, even with an identity Reduce function, one would incur unnecessary sorting between the Map and Reduce stage. Instead, one can use the Map operator to process these records in parallel.

### 5.2 Distribute operator

The Distribute operator converts a table or an indexed table to another indexed table with a different index. The resulting indexed table is stored as a single fixed-num-of-chunk DFS file. This involves shuffling data from whichever chunk the data was on previously to a new chunk as indicated by the partition function for the new index.

The user invokes the Distribute operator as follows:

```
Table  $Y$  = Distribute(Table  $X$ , Field  $i$ , Func
newPartitionFunc)
```

$Y$  is the resulting table after applying the Distribute operator on  $X$ .  $i$  indicates which field of Table  $X$  should be used as the new index. newPartitionFunc is the new partition function for the newly generated table. It takes the following form:

```
int[] newPartitionFunc(Index x)
```

Function `newPartitionFunc` returns one or more integers to indicate which chunk(s) one record should be written to. If more than one integer is returned, the same record will be duplicated over all indicated chunks.

When the `Distribute` operator is invoked, the master node spawns  $C$  separate slave tasks on the slave nodes, one for each chunk. We refer to the task responsible for the  $i$ th chunk “task  $i$ ”. For efficient local processing, Task  $i$  is spawned on the same node that holds chunk  $i$  of Table  $X$ . The slave tasks run parallel to each other. Each slave task generates  $C$  output files locally, one for each chunk of Table  $Y$ . Task  $i$  goes through each record in chunk  $i$  of Table  $X$ , and for each record, it applies the `newPartitionFunc` to determine the chunk number  $j$  (or a list of chunk numbers) for Table  $Y$ , for which the record will be distributed to. It then writes the record to the output corresponding to chunk  $j$  (or to outputs corresponding to the list of chunks).

When a slave task completes, it notifies the master node about the task completion and the location of the  $C$  local output files. When the master node notes that all slave tasks have been completed, it will spawn another set of tasks, one for each chunk, again on the nodes that will hold the corresponding chunk for Table  $Y$ . Again, each slave task runs in parallel. Task  $j$  receives a list of file locations (including the host name), one for each slave task in step 2 indicating the location of Task  $i$ 's output for chunk  $j$  of Table  $Y$ . Task  $j$  remote copies all files to the local node and merges them into chunk  $j$  for Table  $Y$ . The `Distribute` operator finishes when the master node is notified that all slave tasks have finished.

The actions performed by `Map` and `Distribute` operators are similar to part of the actions performed by `MapReduce`. We extract them out as separate operators because we feel they are fundamental operations that are needed by many applications. Extracting them out as separate operators gives the users greater flexibility when they implement their applications through operator compositions.

Both the fixed-num-of-chunk file and the `Distribute` operator give users direct control on how data is placed on the nodes. This capability allows users to optimize local processing, thus saving precious network bandwidth. This is especially important in a Cloud or a Grid consisting of geographically distributed servers across Wide Area Networks (WAN) where the network bandwidth is much smaller than that in a traditional Enterprise infrastructure.

### 5.3 Join operator

The `Join` operator takes two indexed tables and merge the corresponding records if the index fields match. The `GridBatch` system finds the corresponding records that have a matching index, and then invokes a custom function defined by the user. The user-defined function can simply merge the two records, like in a traditional database join, or it can perform any special action as it desires.

The users invoke the `Join` operator as follows:

```
Join(Table X, Table Y, Func joinFunc)
```

where  $X$  and  $Y$  are the two input indexed tables, and `joinFunc` is the custom function provided by the user.

A sample pseudo-code for one implementation of the `joinFunc` is as follows:

```
joinFunc(Record x, Record y)
// Apply necessary processing on Record
// x and y to generate Record z
.....
EmitResult(Table Z, record z)
```

$x$  and  $y$  are a record of Table  $X$  and  $Y$  respectively. When `joinFunc` is invoked, it is guaranteed that the indices for record  $x$  and  $y$  match. `joinFunc` could emit zero or more records for zero or more tables. The example shown only emits one record for one table.

Before the `Join` operator is called, it is the user's responsibility to make sure that Table  $X$  and  $Y$  are partitioned already using the same partition function (e.g., by using the `Distribute` operator) on the index field that the join is based on. The `Join` operator simply performs the join locally without worrying about fetching data from other chunks. This is consistent with our philosophy that the user is the most knowledgeable about how to distribute data in order to achieve the best performance.

When the `Join` operator is invoked, the master node spawns  $C$  tasks, one for each chunk, on the slave node holding the corresponding chunks for Table  $X$  and  $Y$ . Task  $i$  first sorts chunk  $i$  of Table  $X$  and  $Y$  individually in increasing order of their indices. Then, task  $i$  walks through Table  $X$  and  $Y$  with the help of two pointers. Initially, one points at the beginning of  $X$  and the other points at the beginning of  $Y$ . Let  $i(x)$  and  $i(y)$  denote the index value of the records pointed to by the pointers for Table  $X$  and  $Y$  respectively. If  $i(x) = i(y)$ , `joinFunc` is invoked with  $x$  and  $y$  as parameters. Otherwise, if  $i(x) < i(y)$ , advance the pointer for Table  $X$ , and if  $i(x) > i(y)$ , advance the pointer for Table  $Y$ . This process continues until all records are scanned. The

Join operator finishes when the master node is notified that all slave tasks have finished.

In our client application of finding items generated by a set of sources, we first apply the Distribute operator on the barcode table based on the Source field, then simply perform a Join operator between the resulting barcode table and the table holding the list of sources.

## 5.4 Cartesian operator

Unlike the Join operator, which only matches records when their index fields match, the Cartesian operator will match every record of Table  $X$  with every record of Table  $Y$ , and apply a user-defined function.

In our client's business rule checking application, all barcode records are stored as one table and all business rules are stored as another table. The client wants to check all records against all rules to make sure there is no business rule violation. This can be accomplished by simply calling the Cartesian operator. The user-defined function only needs to check if a record violates a particular rule, and the GridBatch system takes care of the dirty plumbing work of matching corresponding records from the two tables.

Cartesian operator can be used to implement a join. The Join operator only works when both tables are indexed and when we desire an exact match on the index field. When a non-exact match is desired, we have to check every record  $x$  against every record  $y$ . The user-defined function can then determine whether  $x$  and  $y$  should be joined together.

The users invoke the Cartesian operator as follows:

```
Cartesian(Table X, Table Y, Func cartesianFunc)
```

where  $X$  and  $Y$  are the two input tables, and cartesianFunc is the custom function provided by the user.

A sample pseudo-code for one implementation of cartesianFunc is as follows:

```
cartesianFunc(Record x, Record y)
// Apply necessary processing on Record
// x and y to generate Record z
.....
EmitResult(Table Z, record z)
```

where  $x$  and  $y$  are records of Table  $X$  and  $Y$  respectively. cartesianFunc could emit zero or more records for zero or more tables. The example shown only emits one record for one table.

Like Join, it is the user's responsibility to first distribute the data. The Cartesian operator simply performs the operation locally without worrying about

fetching data from other chunks. The user should duplicate one of the tables over all chunks (e.g., using the Distribute operator) to guarantee that every record  $x$  is matched against every record  $y$ .

The implementation of the Cartesian operator is similar to that of the Join operator. The only difference is that no matching of indices is needed.

## 5.5 Recurse operator

The Reduce part of MapReduce is inherently not parallelizable. But, if there are many reduce operations, an application can still benefit from parallelization by spreading the reduce operations across many nodes. For web applications, it is generally true that there are many reduce operations (e.g., word count). However, this is not necessarily true for analytics applications, where the users are only using a few reduce operations. For example, the user may just want to sort the output for reporting or collect a few statistics. In this case, the Reduce operator becomes the bottleneck, limiting the scalability of the application.

Many reduce operations are commutative and associative, and hence, order independent. For example, counting the number of occurrences of an event involves addition, which is commutative and associative. The order of how addition happens does not affect the end result. Similarly, sorting is order independent.

For these order independent reduce operations, we introduce the Recurse operator. Users invoke Recurse as follows:

```
Recurse(Table X, Func recurseFunc)
```

where  $X$  is the input table, and recurseFunc is the custom function provided by the user. The Recurse operator merges the table into a single record.

A sample pseudo-code for one implementation of recurseFunc is as follows. For conciseness, this example shows the addition operation, but it is equally easy to implement the merge sort algorithm.

```
Record recurseFunc(Record x1, Record x2)
// Apply processing on x1 and x2
return x = x1 + x2
```

where  $x_1$  and  $x_2$  are partial results from merging two subparts of Table  $X$ . recurseFunc specifies how to merge the two partial results further, and GridBatch applies the function recursively over all records of Table  $X$  to eventually produce the overall sum.

Compared to the Reduce operation in MapReduce, Recurse is more efficient because it can parallelize the reduce operation over many nodes. In addition, Recurse can minimize network traffic by merging results

from close-by nodes. Since bandwidth is only consumed on local network segments, bandwidth on other links is preserved for other tasks. Network bandwidth consumption can be cut down further if only partial results are desired. For example, if we are only interested in the top 10 records, each node would only compute the local top 10, and send it to the neighboring node, who in turn will merge it with the local result to produce the top 10. Since only 10 records are passed from node to node, the traffic is much smaller than that used by MapReduce, which would require every node sending every record to a single node where the reduce operation is carried out.

When the Recurse operator is invoked, the master node spawns many tasks, one for each chunk and it is spawned on the slave node that holds that chunk for Table  $X$ . Task  $i$  first merges all records in chunk  $i$  using `recurseFunc`. First, it takes the first two records  $x_1$  and  $x_2$ , and apply `recurseFunc`. The result is saved in record  $s$ . Task  $i$  then takes the third record  $x_3$  and apply `recurseFunc` on  $s$  and  $x_3$ . This process continues for all of the rest of records.

We now need to merge the results from each chunk together. Half of the tasks will send their results  $s$  to another task in the other half, where  $s$  is merged with the local result. At the end, only half of the tasks have partial results. This process repeats, i.e., one quarter of the tasks will send their partial results to another task in the other quarter tasks, where results are merged. The process ends when the final result is derived. The master node is responsible for coordinating the merging sequence (who sends results to who else) and it will take the network topology into account so that, most of the time, a task only sends its result to a nearby task.

## 5.6 Neighbor operator

Unlike database tables, tables in GridBatch could have an implicit order semantic. For example, the barcode table in our client application preserves the scanning order. Some analytics functions, such as our client’s interlacing detection problem, need to analyze the sequence to derive meaningful results.

The Neighbor operator groups neighboring records and invokes a user-defined function to analyze the subsequence. The users invoke the Neighbor operator as follows:

```
Neighbor(int k, Table X, Func neighborFunc)
```

where  $k$  is a small constant that indicates how many neighboring records to group together,  $X$  is the input table, and `neighborFunc` is the custom function provided by the user.

`neighborFunc` takes  $k$  records as arguments. The  $k$  arguments follow the order in the table, i.e., the record in argument  $j$  immediately follows the record in argument  $j - 1$  in the table. A sample `neighborFunc` pseudo-code for our client’s interlacing detection is as follows:

```
neighborFunc(Record x1, Record x2)
// report discontinuity
if ( x1.containerID != x2.containerID )
    EmitResult(Table Z, record x1)
```

where  $x_1$  and  $x_2$  are neighboring records of Table  $X$ .

This function adds the first record to a new Table  $Z$  if the two records belong to different containers. To detect whether there is any interlacing, it is sufficient to count the number of occurrences of each container ID in Table  $Z$ . If any container appears more than once in Table  $Z$ , then some items from that container have been misplaced (note that the container ID is globally unique). Counting the number of appearances can be accomplished by the Recurse operator.

Interlacing detection using SQL is very hard to do since databases do not preserve the sequence semantic. Furthermore, it is not possible to perform interlacing detection with MapReduce either for the same reason. Until now, the only alternative is to write a sequential program to scan the whole barcode table and detect any discontinuity. However, this naive solution is very time consuming since the barcode table is many terabytes long. By using the Neighbor and Recurse operators, we implemented the same logic with only a few lines of code, yet we are able to achieve very high performance. This demonstrates the power and capabilities of the GridBatch system.

## 6 Experiment results

The flexibility advantage of GridBatch is apparent from the last section. Therefore, we focus on demonstrating the performance advantage of GridBatch in this section. Even though extensive experiments have been carried out to validate our solution, due to space limitations, we only report a couple of the results that highlight where we gain the most performance advantage.

Our implementation of GridBatch is based on Hadoop[11], an open-source implementation of MapReduce. Using the same code base allows us to compare the performance with MapReduce.

In addition to comparing to MapReduce, we also compare against another competing proposal under evaluation by our client. The competing proposal

is based on the traditional single server architecture which employs a Sun Fire E25K server with 72 CPUs and 1 Terabytes of memory. To achieve the desired performance target, an in-memory database is used to run the analytics. The Sun server costs roughly 4 million dollars, which is much higher than the cost of the GridBatch approach since the cost of a commodity PC server is expected to be less than one thousand dollars.

Our experiments are carried out in Amazon EC2 Cloud. Amazon promises each server is equivalent to a system with a 1.7Ghz x86 processor, 1.75GB of RAM, 160GB of local disk, and 250Mb/s of network bandwidth.

### 6.1 Duplicate detection

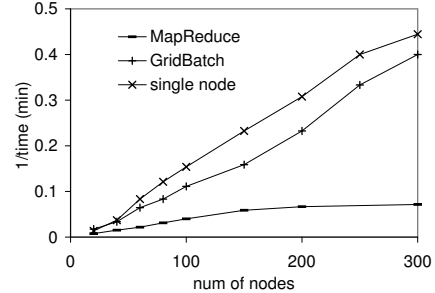
For the duplicate detection problem, we used a real data set that is 250GB in size and each record is roughly 170 bytes long. The client has set a performance goal of reporting results in 5 minutes.

With GridBatch, we first load the data set as an indexed table with the index being the barcode and store it as a fixed-num-of-chunk file (it takes the same amount of time to load it as a fixed-chunk-size file). Since the index field is the barcode, we are guaranteed that no duplicate will reside on two different chunks. The duplicate detection application is run frequently, so storing the table based on the barcode is the most efficient as it will eliminate data movement.

The duplicate detection application first performs a Join between the table and itself (Join is needed to sort the data set) and then it performs a Neighbor operator to report any duplicate barcodes. Since the barcodes are sorted already after Join, a Neighbor operator with  $k = 2$  (checking on immediate neighbors) is sufficient. The user-defined function for the Neighbor operator simply outputs the record if the two neighbors have identical barcodes. Because we join the barcode table with itself, if a barcode is duplicated  $n$  times in the barcode table, it will appear  $n^2$  times in the resulting table. This is not a problem not only because the number of duplicates is expected to be small, but also because there is a direct relationship between the number of occurrences in the resulting table and that in the original table; hence, a simple post-processing step is sufficient.

As a comparison, the MapReduce version uses the fixed-chunk-size file as in GFS. The Map function extracts the barcode as the key and 1 as the value. The Reduce function simply counts the number of occurrences and if it is more than once, reports the barcode.

As a base line, the single server in-memory database analyzed the same data set. It is able to produce the



**Figure 2. Duplicate detection. Y axis is inverse of time in minutes. X axis is the number of nodes. “Single node” shows what a single node would take to process local data as if it is a single-node cluster.**

result in 5 minutes.

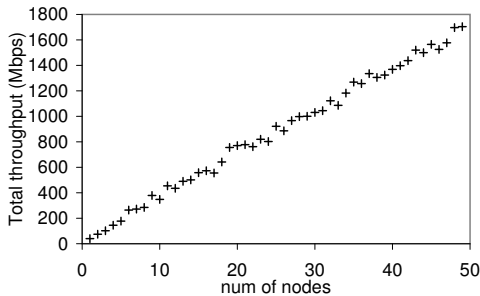
The results for GridBatch and MapReduce is shown in Fig. 2. We plot the inverse of time in minutes (which shows the amount of work completed per unit time) v.s. the number of nodes used. MapReduce quickly saturates at 150 nodes mainly due to the limited network bandwidth. In comparison, GridBatch scales very well because little network bandwidth is consumed. In fact, we have noticed superlinear scaling (i.e., the amount of work more than doubles when the number of nodes doubles). We attribute this to the inefficiency of processing large amount of data on one node (memory page misses, swapping etc.). As a verification, the curve marked “single node” shows how much time a single node takes to process only the local data. The gap between “single node” and “GridBatch” indicates the overhead incurred by the job scheduling system for coordination.

Based on the trend, MapReduce would not be able to meet the performance target with even 1000 nodes. In comparison, using GridBatch, we processed all data in 4.3 minutes using only 150 nodes. More importantly, the trend shows that GridBatch will scale beyond 300 nodes even if the performance target is more stringent.

### 6.2 Data write throughput

Because of the large number of items shipped, the client requires a very high data write throughput to capture all barcodes. The throughput from a single server architecture is inherently limited by the server capability or the directly connected network link bandwidth. In contrast, a distributed architecture like ours could achieve very high throughput.

Fig. 3 shows the total data write throughput as a function of the number of DFS clients. Each DFS



**Figure 3. Data write throughput as a function of the number of DFS clients.**

client represents a origin or destination which will upload barcodes to a unique fixed-num-of-chunk DFS file at the highest rate that can be handled by the DFS client. The GridBatch system has 100 data nodes. It is clear from the figure that we scale linearly. More importantly, with 50 DFS clients, we achieved 1.7Gbps throughput, much higher than the 250Mbps promised bandwidth. This is because DFS clients are writing to different data nodes at any given time; thus, it is unlikely to saturate any given network link.

## 7 Conclusion

Due to competitive pressure, Enterprise wants to analyze a large amount of data in a short amount of time, under a conflicting goal of using the least cost possible. There are a couple of challenges in meeting these goals. First, the amount of data we collect is so large that it is already beyond the capability of a uniprocessor to process in a reasonable amount of time. The growth trend of data volume and uniprocessor capacity suggests that this problem cannot be solved by technology scaling. Instead, some form of parallel processing is necessary. Unfortunately, writing parallel programs is inherently difficult due to the distributed nature.

Second, even though Cloud Computing promises to dramatically lower the cost, especially for time-varying computation demands, its infrastructure is very different from the traditional Enterprise infrastructure. Cloud Computing, at least at its current incarnation, is synonymous with Commodity Computing, i.e., a Cloud is composed of many commodity servers interconnected by commodity network switches. The lower network bandwidth and the inherent lower hardware reliability force Enterprises to rethink their application architecture.

Inspired by Google's MapReduce framework, we propose GridBatch which not only performs well in a Cloud infrastructure, but also makes it easy to write

parallel programs for data-intensive batch applications. We extended the file system used by MapReduce to directly support partitioned data set, as well as introduced several new operators, each implementing a specific data manipulation pattern. Using a real client application example, we showed how these operators could be used to solve real problems. Through experiments, we demonstrated that the operators give the users not only direct control on data movement and storage but also flexibility in composition, so that the application can achieve higher performance, not only compared to the traditional approach in a traditional Enterprise infrastructure, but also compared to MapReduce in a Cloud infrastructure.

## References

- [1] M. Atkinson. Uk e-science grid infrastructure meets biological research challenges. <http://www.nesc.ac.uk/talks/mpa/BioGridCambridge2Oct02.ppt>, Oct. 2002.
- [2] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proc. SIGMOD*, 2007.
- [3] T. H. Davenport and J. G. Harris. *Competing on Analytics: The New Science of Winning*. Harvard Business School Press, 2007.
- [4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.
- [5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles*, October 2003.
- [6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, March 2007.
- [7] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B. Khailany. The imagine stream processor. In *Proceedings of Int'l Conference on Computer Design*, Sep. 2002.
- [8] C. S. Mullins. Database trends. <http://www.bwdb2ug.org/Presentations/DatabaseTrends.pdf>.
- [9] D. Patterson and J. Hennessy. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition, 2006.
- [10] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, 13(4):227-298, 2005.
- [11] Hadoop. <http://lucene.apache.org/hadoop>.
- [12] Pig. <http://research.yahoo.com/project/pig>.