UCS15E08 – CLOUD COMPUTING

UNIT 3

DATA IN THE CLOUD

In the process of creating a planetary scale web search service, Google in particular has developed a massively parallel and fault tolerant distributed file system (GFS) along with a data organization (BigTable) and programming paradigm (MapReduce) that is markedly different from the traditional relational model. Such 'cloud data strategies' are particularly well suited for large-volume massively parallel text processing, as well as possibly other tasks, such as enterprise analytics. The public cloud computing offerings from Google (i.e. App Engine) as well as those from other vendors have made similar data models (Google's Datastore, Amazon's SimpleDB) and programming paradigms (Hadoop on Amazon's EC2) available to users as part of their cloud platforms.

At the same time there have been new advances in building specialized database organizations optimized for analytical data processing, in particular column-oriented databases such as Vertica. It is instructive to note that the BigTable-based data organization underlying cloud databases exhibits some similarities to column-oriented databases. These concurrent trends along with the ease of access to cloud platforms are witnessing a resurgence of interest in non-relational data organizations and an exploration of how these can best be leveraged for enterprise applications.

RELATIONAL DATABASES

Before we delve into cloud data structures we first review traditional relational database systems and how they store data. Users (including application programs) interact with an RDBMS via SQL; the database 'front-end' or parser transforms queries into memory and disk level operations to optimize execution time. Data records are stored on pages of contiguous disk blocks, which are managed by the disk-space-management layer.

Pages are fetched from disk into memory buffers as they are requested, in many ways similar to the file and buffer management functions of the operating system, using pre-fetching and page replacement policies. However, database systems usually do not rely on the file system layer of the OS and instead manage disk space themselves. This is primarily so that the database can have full control of when to retain a page in memory and when to release it. The database needs be able to adjust page replacement policy when needed and pre-fetch pages from disk based on expected access patterns that can be very different from file operations. Finally, the operating system files used by databases need to span multiple disks so as to handle the large storage requirements of a database, by efficiently exploiting parallel I/O systems such as RAID disk arrays or multi-processor clusters.

Traditional relational databases are designed to support high-volume transaction processing involving many, possibly concurrent, record level insertions and updates. Supporting concurrent access while ensuring that conflicting actions by simultaneous users do not result in inconsistency is the responsibility of the transaction management layer of the database system that ensures ‘isolation’ between different transactions through a variety of locking strategies.

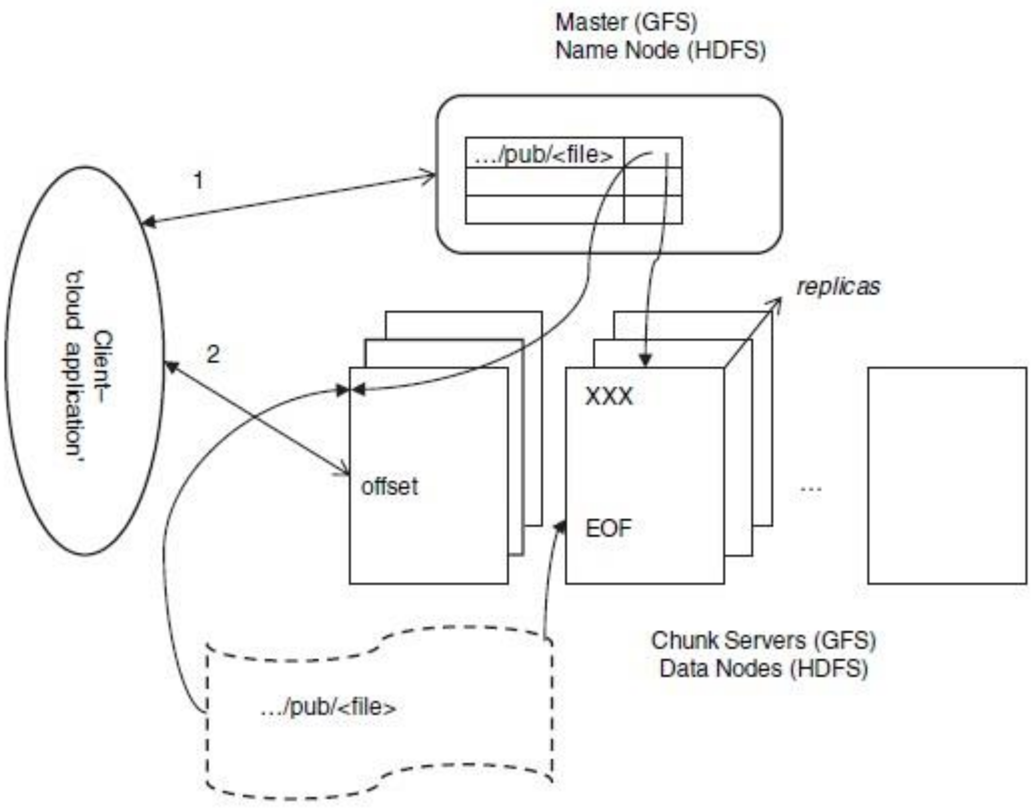
In the case of parallel and distributed architectures, locking strategies are further complicated since they involve communication between processors via the well-known ‘two-phase’ commit protocol. It is important to note that the parallel database systems developed as extensions to traditional relational databases were designed either for specially constructed parallel architectures, such as Netezza, or for closely coupled clusters of at most a few dozen processors.

At this scale, the chances of system failure due to faults in any of the components could be sufficiently compensated for by transferring control to a ‘hot-standby’ system in the case of transaction processing or by restarting the computations in the case of data warehousing applications. As we shall see below, a different approach is required to exploit a parallelism at a significantly larger scale.

CLOUD FILE SYSTEMS: GFS AND HDFS

The Google File System (GFS) is designed to manage relatively large files using a very large distributed cluster of commodity servers connected by a high-speed network. It is therefore designed to (a) expect and tolerate hardware failures, even during the reading or writing of an individual file (since files are expected to be very large) and (b) support parallel reads, writes and appends by multiple client programs. A common use case that is efficiently supported is that of many ‘producers’ appending to the same file in parallel, which is also being simultaneously read by many parallel ‘consumers.

The architecture of cloud file systems is illustrated in Figure below. Large files are broken up into 'chunks' (GFS) or 'blocks' (HDFS), which are themselves large (64MB being typical). These chunks are stored on commodity (Linux) servers called Chunk Servers (GFS) or Data Nodes (HDFS); further each chunk is replicated at least three times, both on a different physical rack as well as a different network segment in anticipation of possible failures of these components apart from server failures.



When a client program ('cloud application') needs to read/write a file, it sends the full path and offset to the Master (GFS) which sends back meta-data for one (in the case of read) or all (in the case of write) of the replicas of the chunk where this data is to be found. The client caches such meta-data so that it need not contact the Master each time. Thereafter the client directly reads data from the designated chunk server; this data is not cached since most reads are large and caching would only complicate writes.

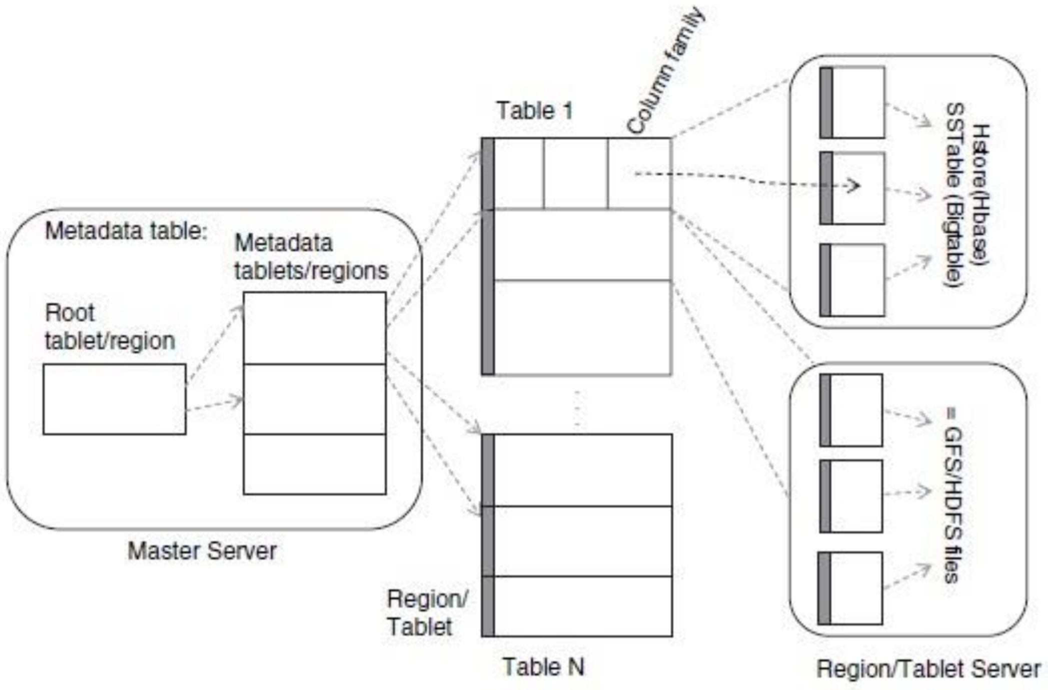
In case of a write, in particular an append, the client sends only the data to be appended to all the chunk servers; when they all acknowledge receiving this data it informs a designated ‘primary’ chunk server, whose identity it receives (and also caches) from the Master. The primary chunk server appends its copy of data into the chunk at an offset of its choice; note that this may be beyond the EOF to account for multiple writers who may be appending to this file simultaneously. The primary then forwards the request to all other replicas which in turn write the data at the same offset if possible or return a failure. In case of a failure the primary rewrites the data at possibly another offset and retries the process.

The Master maintains regular contact with each chunk server through heart beat messages and in case it detects a failure its meta-data is updated to reflect this, and if required assigns a new primary for the chunks being served by a failed chunk server. Since clients cache meta-data, occasionally they will try to connect to failed chunk servers, in which case they update their meta-data from the master and retry.

BIGTABLE, HBASE AND DYNAMO

BigTable is a distributed structured storage system built on GFS; Hadoop’s HBase is a similar open source system that uses HDFS. A BigTable is essentially a sparse, distributed, persistent, multidimensional sorted map. Data in a BigTable is accessed by a row key, column key and a timestamp. Each column can store arbitrary name–value pairs of the form column-family:label, string. The set of possible column-families for a table is fixed when it is created whereas columns, i.e. labels within the column family, can be created dynamically at any time. Column families are stored close together in the distributed file system; thus the BigTable model shares elements of column oriented databases. Further, each Bigtable cell (row, column) can contain multiple versions of the data that are stored in decreasing timestamp order.

Since data in each column family is stored together, using this data organization results in efficient data access patterns depending on the nature of analysis: For example, only the location column family may be read for traditional data-cube based analysis of sales, whereas only the product column family is needed for say, market-basket analysis. Thus, the BigTable structure can be used in a manner similar to a column-oriented database.



BigTable and HBase rely on the underlying distributed file systems GFS and HDFS respectively and therefore also inherit some of the properties of these systems. In particular large parallel reads and inserts are efficiently supported, even simultaneously on the same table, unlike a traditional relational database. In particular, reading all rows for a small number of column families from a large table, such as in aggregation queries, is efficient in a manner similar to column-oriented databases. Random writes translate to data inserts since multiple versions of each cell are maintained, but are less efficient since cell versions are stored in descending order and such inserts require more work than simple file appends.

Dynamo's data model is that of simple key-value pairs, and it is expected that applications read and write such data objects fairly randomly. This model is well suited for many web-based e-commerce applications that all need to support constructs such as a 'shopping cart.'

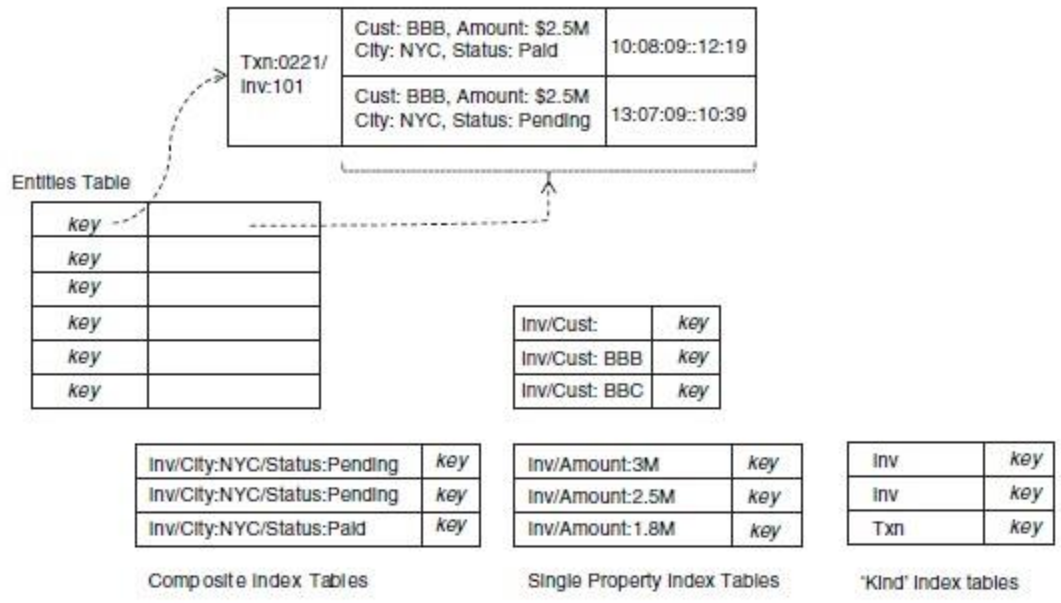
Dynamo also replicates data for fault tolerance, but uses distributed object versioning and quorum-consistency to enable writes to succeed without waiting for all replicas to be successfully updated, unlike in the case of GFS. Managing conflicts if they arise is relegated to reads which are provided enough information to enable application dependent resolution.

Because of these features, Dynamo does not rely on any underlying distributed file system and instead directly manages data storage across distributed nodes.

Dynamo is able to handle transient failures by passing writes intended for a failed node to another node temporarily. Such replicas are kept separately and scanned periodically with replicas being sent back to their intended node as soon as it is found to have revived. Finally, Dynamo can be implemented using different storage engines at the node level, such as Berkeley DB or even MySQL.

CLOUD DATA STORES: DATASTORE AND SIMPLEDB

The Google and Amazon cloud services do not directly offer BigTable and Dynamo to cloud users. Using Hadoop's HDFS and HBase, which are available as Amazon AMIs, users can set up their own BigTable-like stores on Amazon's EC2. Google and Amazon both offer simple key-value pair database stores, viz. Google App Engine's Datastore and Amazon's SimpleDB.



It is useful to think of a BigTable as an array that has been horizontally partitioned (also called 'sharded') across disks, and sorted lexicographically by key values. In addition to single key lookup, this structure also enables highly efficient execution of prefix and range queries on key values, e.g. all keys having prefix 'Txn,' or in the range 'Amount:1M' to 'Amount:3M.' From this feature derive the key structures of entity and index tables that implement Datastore.

The entities table stores multiple versions of each entity, primarily in order to support transactions spanning updates of different entities in the same group. Only one of the versions of each entity is tagged as ‘committed,’ and this is updated only when a transaction succeeds on all the entities in the group; journal entries consisting of previous entity versions are used to rollback if needed.

Notice also that this mapping of the Datastore API onto BigTable does not exploit the column-oriented storage of BigTable, since a single column family is used. Thus while BigTable (and HBase) are potentially useful in large analytical tasks for the same reason that column-oriented databases are, Datastore as an application of BigTable does not share this property. Datastore is much more suited to transactional key-value pair updates, in much the same manner as Amazon’s SimpleDB is, with the difference that its consistency properties are stronger (as compared to the ‘eventual’ consistency of SimpleDB), at the cost of a fixed overhead even for small writes.

MAPREDUCE AND EXTENSIONS

The MapReduce programming model was developed at Google in the process of implementing large-scale search and text processing tasks on massive collections of web data stored using BigTable and the GFS distributed file system. The MapReduce programming model is designed for processing and generating large volumes of data via massively parallel computations utilizing tens of thousands of processors at a time. The underlying infrastructure to support this model needs to assume that processors and networks will fail, even during a particular computation, and build in support for handling such failures while ensuring progress of the computations being performed**.**

Hadoop is an open source implementation of the MapReduce model developed at Yahoo, and presumably also used internally. Hadoop is also available on pre-packaged AMIs in the Amazon EC2 cloud platform, which has sparked interest in applying the MapReduce model for large-scale, fault-tolerant computations in other domains, including such applications in the enterprise context.

1. PARALLEL COMPUTING

Parallel computing has a long history with its origins in scientific computing in the late 60s and early 70s. Different models of parallel computing have been used based on the nature and evolution of multiprocessor computer architectures. The shared-memory model assumes that any processor can access any memory location, albeit not equally fast In the distributed memory model each processor can address only its own memory and communicates with other processors using message passing over the network. In scientific computing applications for which these models were developed, it was assumed that data would be loaded from disk at the start of a parallel job and then written back once the computations had been completed, as scientific tasks were largely compute bound.

A scalable parallel implementation is one where:

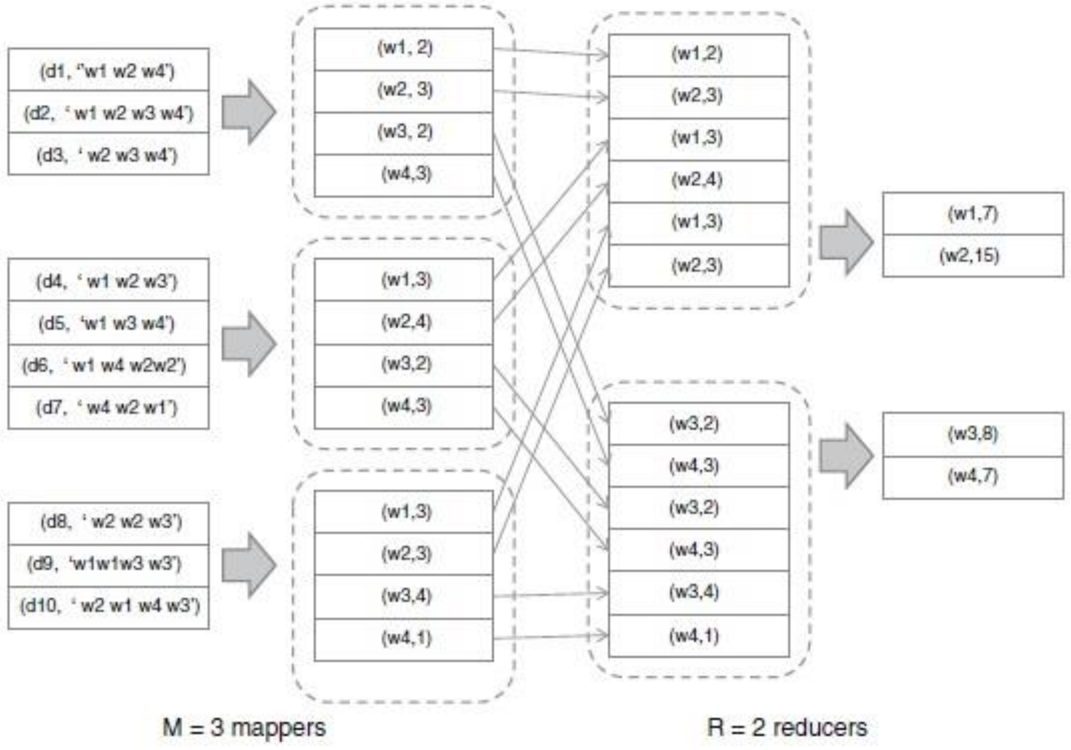
A. The parallel efficiency remains constant as the size of data is increased along with a corresponding increase in processors.

B. The parallel efficiency increases with the size of data for a fixed number of processors.

We assume a distributed-memory model with a shared disk, so that each processor is able to access any document from disk in parallel with no contention. Further we assume that the time spent c for reading each word in the document is the same as that of sending it to another processor via inter processor communication. On the other hand, the time to add to a running total of frequencies is negligible as compared to the time spent on a disk read or inter processor communication, so we ignore the time taken for arithmetic additions in our analysis. Finally, assume that each word occurs f times in a document, on average. With these assumptions, the time for computing all the m frequencies with a single processor is $n \times m \times f \times c$, i.e. since each word needs to be read approximately f times in each document.

2. THE MAPREDUCE MODEL

Traditional parallel computing algorithms were developed for systems with a small number of processors, dozens rather than thousands. So it was safe to assume that processors would not fail during a computation. At significantly larger scales this assumption breaks down, as was experienced at Google in the course of having to carry out many large-scale computations similar to the one in our word counting example. The MapReduce parallel programming abstraction was developed in response to these needs, so that it could be used by many different parallel applications while leveraging a common underlying fault-tolerant implementation that was transparent to application developers.



The MapReduce programming model generalizes the computational structure of the above example. Each map operation consists of transforming one set of key-value pairs to another:

Map: (k1, v1) → [(k2, v2)].

In our example each map operation takes a document indexed by its id and emits a list if word-count pairs indexed by word-id:

(dk, [w1 . . .wn]) → [(wi, ci)].

The reduce operation groups the results of the map step using the same key k2 and performs a function f on the list of values that correspond to each key value:

Reduce: (k2, [v2]) → (k2, f ([v2])).

In our example each reduce operation sums the frequency counts for each word:

$$(w_i, [c_i]) \rightarrow \left(w_i, \sum_i c_i\right).$$

The implementation also generalizes. Each mapper is assigned an input-key range (set of values for k1) on which map operations need to be performed. The mapper writes results of its map operations to its local disk in R partitions, each corresponding to the output-key range (values of k2) assigned to a particular reducer, and informs the master of these locations. Next each reducer fetches these pairs from the respective mappers and performs reduce operations for each key k2 assigned to it.

Such a fault-tolerant implementation of the MapReduce model has been implemented and is widely used within Google; more importantly from an enterprise perspective, it is also available as an open source implementation through the Hadoop project along with the HDFS distributed file system.

The MapReduce model is widely applicable to a number of parallel computations, including database-oriented tasks which we cover later. Finally we describe one more example, that of indexing a large collection of documents or, for that matter any data including database records: The map task consists of emitting a word-document/record id pair for each word:

$$(dk, [w1 \ldots wn]) \rightarrow [(wi, dk)].$$

The reduce step groups the pairs by word and creates an index entry for each word:

$$[(wi, dk)] \rightarrow (wi, [di1 \ldots dim]).$$

Indexing large collections is not only important in web search, but also a critical aspect of handling structured data; so it is important to know that it can be executed efficiently in parallel using MapReduce. Traditional parallel databases focus on rapid query execution against data warehouses that are updated infrequently; as a result these systems often do not parallelize index creation sufficiently well.

PARALLEL EFFICIENCY OF MAPREDUCE

Parallel efficiency is impacted by overheads such as synchronization and communication costs, or load imbalance. The MapReduce master process is able to balance load efficiently if the number of map and reduce operations are significantly larger than the number of processors. For large data sets this is usually the case (since an individual map or reduce operation usually deals with a single document or record). However, communication costs in the distributed file system can be significant, especially when the volume of data being read, written and transferred between processors is large.

Now consider running the decomposed computation on P processors that serve as both mappers and reducers in respective phases of a MapReduce based parallel implementation. As compared to the single processor case, the additional overhead in a parallel MapReduce implementation is between the map and reduce phases where each mapper writes to its local disk followed by each reducer remotely reading from the local disk of each mapper. For the purposes of our analysis we shall assume that the time spent reading a word from a remote disk is also c, i.e. the same as for a local read.

A strict implementation of MapReduce as per the definitions and does not allow for partial reduction across all input values seen by a particular reducer, which is what enabled the parallel implementation of Section to be highly efficient and scalable. Therefore, in practice the map phase usually includes a combine operation in addition to the map, defined as follows:

Combine: (k2, [v2]) → (k2, fc([v2])).

Finally, recall our definition of a scalable parallel implementation: A MapReduce implementation is scalable if we are able to achieve an efficiency that approaches one as data volume D grows, and remains constant as D and P both increase. Using combiners is crucial to achieving scalability in practical MapReduce implementations by achieving a high degree of data ‘compression’ in the map phase.

RELATIONAL OPERATIONS USING MAPREDUCE

Enterprise applications rely on structured data processing, which over the years has become virtually synonymous with the relational data model and SQL. Traditional parallel databases have become fairly sophisticated in automatically generating parallel execution plans for SQL statements. At the same time these systems lack the scale and fault-tolerance properties of MapReduce implementations, naturally motivating the quest to execute SQL statements on large data sets using the MapReduce model.

The MapReduce implementation works as follows: In the map step, each mapper reads a (random) subset of records from each input table Sales and Cities, and segregates each of these by address, i.e. the reduce key k2 is ‘address.’ Next each reducer fetches Sales and Cities data for its assigned range of address values from each mapper, and then performs a local join operation including the aggregation of sale value and grouping by city. Note that since addresses are randomly assigned to reducers, sales aggregates for any particular city will still be distributed across reducers. A second mapreduce step is needed to group the results by city and compute the final sales aggregates.

Note that while the parallel MapReduce implementation looks very similar to a traditional parallel sort-merge join, as can be found in most database textbooks, parallel SQL implementations usually distribute the smaller table, Cities in this case, to all processors. As a result, local joins and aggregations can be performed in the first map phase itself, followed by a reduce phase using city as the key, thus obviating the need for two phases of data exchange.

Naturally there have been efforts at automatically translating SQL-like statements to a map-reduce framework. Two notable examples are Pig Latin developed at Yahoo!, and Hive [62] developed and used at Facebook. Both of these are open source tools available as part of the Hadoop project, and both leverage the Hadoop distributed file system HDFS.

There has been considerable interest in comparing the performance of MapReduce-based implementations of SQL queries with that of traditional parallel databases, especially specialized column-oriented databases tuned for analytical queries. In general, as of this writing, parallel databases are still faster than available open source implementations of MapReduce (such as Hadoop), for smaller data sizes using fewer processes where fault tolerance is less critical. MapReduce-based implementations, on the other hand, are able to handle orders of magnitude larger data using massively parallel clusters in a fault-tolerant manner. Thus, MapReduce is better suited to ‘extract-transformload’ tasks, where large volumes of data need to be processed (especially using complex operations not easily expressed in SQL) and the results loaded into a database or other form of permanent structured storage. MapReduce is also preferable over traditional databases if data needs to be processed only once and then discarded: As an example, the time required to load some large data sets into a database is 50 times greater than the time to both read and perform the required analysis using MapReduce.

HadoopDB is an attempt at combining the advantages of MapReduce and relational databases by using databases locally within nodes while using MapReduce to coordinate parallel execution. Another example is SQL/MR from Aster Data that enhances a set of distributed SQL-compliant databases with MapReduce programming constructs. Needless to say, relational processing using MapReduce is an active research area and many improvements to the available state of the art are to be expected in the near future.

ENTERPRISE BATCH PROCESSING USING MAPREDUCE

In the enterprise context there is considerable interest in leveraging the MapReduce model for high-throughput batch processing, analysis on data warehouses as well as predictive analytics. We have already illustrated how analytical SQL queries can be handled using MapReduce. High-throughput batch processing operations on transactional data, usually performed as 'end-of-day' processing, often need to access and compute using large data sets. These operations are also naturally time bound, having to complete before transactional operations can resume fully. The time window required for daily batch processing often constrains the online availability of a transaction processing system. Exploiting parallel computing leveraging cloud technology presents an opportunity to accelerate batch processing.

The challenge in deploying public cloud-based batch processing is the cost of data transfer: Thus until transactional data is itself stored in the cloud MapReduce-based parallel batch processing can best be leveraged within enterprises using open source tools such as Hadoop