A Unified Engine for RDBMS and MapReduce

Program, query, and analyze petabytes of data in a new way

Inside this Whitepaper

Introduction

In 2004, Google published a research paper on the MapReduce framework they developed for their internal data processing needs. In simple, approachable terms, the paper describes how Google developers harness massively parallel clusters of computers to analyze some of the largest datasets ever collected. Since that paper was published, there has been ongoing discussion about the role of this technology outside the walls of Google. Excitement about MapReduce has spread quickly in the computing industry, particularly in young and forward-looking firms. But there is confusion and controversy about how the technology fits into the larger ecosystem of information technology, especially with respect to other "big data" solutions like massively parallel SQL database engines.

In this whitepaper, we provide a technical context for that discussion. In a nutshell, we present SQL and MapReduce as two different programming paradigms that are implemented via a common engine architecture: parallel dataflow. Seen in these terms, MapReduce can be viewed as a new programming interface to traditional data-parallel computing.

After presenting this context, we introduce Greenplum MapReduce: a seamless integration of MapReduce and relational database functionality unified in one massively parallel dataflow engine. We describe how Greenplum allows MapReduce programs and SQL to interoperate, efficiently and flexibly processing data in both standard files and database tables.

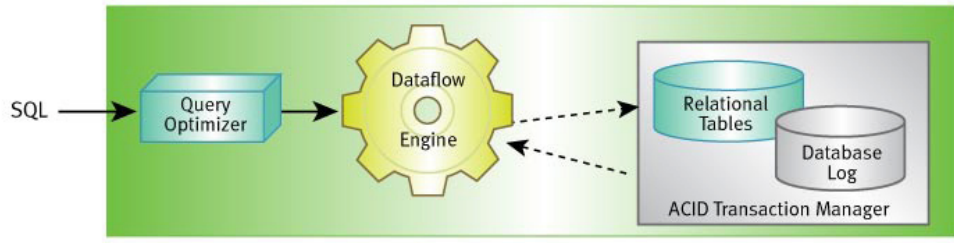
History: Three Revolutions

To understand Greenplum's MapReduce implementation, it is helpful to see it in the context of three historical shifts in the technology behind large-scale data management: the Relational Database revolution, the rise of shared-nothing parallelism, and the popularization of the MapReduce parallel programming paradigm.

Greenplum

Relational Revolution (1970s-present)

Relational Database Systems (RDBMSs) were a radical idea when they were introduced, and they revolutionized the way that enterprises manage their records. The birth of relational technology in research is well documented. In 1970, IBM researcher Ted Codd published his first paper on the relational model of data [Codd70], which proposed representing data in tables of rows and columns, and querying the data using a high-level declarative language that formed the foundation for what we now know as SQL. Some 5 years later, Jim Gray and colleagues at IBM research proposed ACID transactions as a model for correctness of concurrent updates in a database [Gray78]. Codd and Gray both received the Turing award (the "Nobel prize of computer science") for this work. By the mid-1970's, researchers at IBM and UC Berkeley were hard at work on the two key prototype systems – System R and Ingres – that gave birth to the modern relational database industry. Both of these systems developed query optimizer technology that compiles declarative queries and passes the result to a dataflow processing engine, which direct streams of data through operations like filters, index lookups, joins, grouping and aggregation.



Relational databases remain the workhorses of modern record keeping 30 years later, and for good reason. Modern implementations of ACID transactions ensure dependable, consistent management of data storage. The declarative nature of SQL enables data analysis via ad hoc queries, and ensures that data-centric applications continue to work correctly even as data layouts and database hardware evolves. Beneath all this, the simple elegance of the relational model helps provide discipline needed for sound, long-term database design.

"Shared Nothing" Parallelism (1980's-present)

As relational database systems were becoming a commercial reality in the 1980's, efforts were afoot to accelerate database performance via custom hardware known then as "database machines". However, it quickly became clear that economies of scale favored commodity hardware over custom solutions: the latest version of a commodity computer invariably provides a better price/performance point than last year's custom-designed

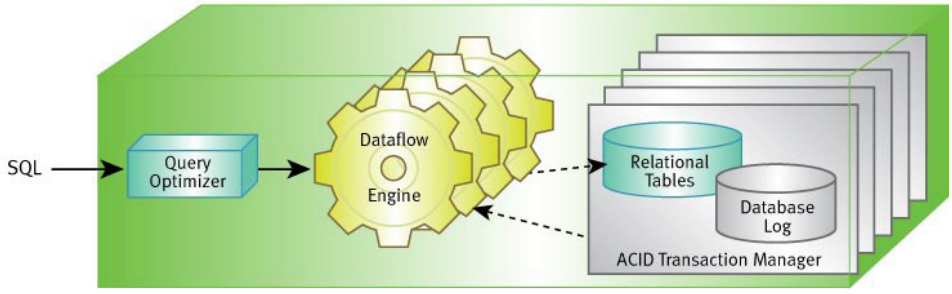
The Relational Database Revolution

- Relational Data Model
  - Simple, general table abstraction
  - Encourages disciplined database design for reuse over time
- ACID Transactions
  - All updates of a transaction either written together to stable storage, or rolled back
  - Database and query results guaranteed consistent even in the face of concurrent updates
- Declarative Language (SQL)
  - High-level queries specify "What" data to get, not "How" to get it
  - Hardware and disk layouts can change without modification to apps ("data independence")
  - Query optimizer translates from declarative SQL to a dataflow execution graph
- Dataflow Processing Engine
  - Efficient processing of big data sets
  - Data streamed off disk, piped through a graph of dataflow operators

machine, negating the performance benefits of customization. As a result, the main early efforts toward database machines were deemed a failure by the researchers and entrepreneurs who pioneered the area [BoralDeWitt83].

Out of the ashes of the work on database machines, a new idea rose: database software could be parallelized to leverage multiple commodity processors in a network to deliver increased scalability and performance. The failed history of custom database machines led to a particular interest in building parallel databases using commodity computers. The term "shared-nothing" parallelism was coined for these computing clusters, to distinguish them from the shared-memory multiprocessors that were being designed at the time for scientific applications.

In order to harness the power of a cluster, query processing software had to evolve to take advantage of multiple disks, processors, and network links operating in parallel. To achieve this, the rows of a table were partitioned across multiple machines with separate disks, enabling parallel I/O scans of big tables. Basic relational query operators like selection, join, grouping and aggregation were reinvented to run in parallel via similar partitioning schemes: the operations undertaken by each node in the cluster are the same, but the data being pumped through the fabric is automatically partitioned to allow each node to work on its piece of the operator independently. Finally, these architectures allowed multiple relational operators to operate at the same time, allowing pipeline parallelism in which an operator producing a data stream runs in parallel with the operator consuming it. The resulting shared-nothing parallel RDBMSs were explored in research prototypes like Gamma and Bubba, and commercialized early on by Teradata and Tandem.



Shared-nothing architectures enabled relational databases to scale to unprecedented levels. This changed the way that many businesses approached the value of data: in addition to keeping the current books correct, analytic applications could be built over historical records to provide new business efficiencies. In the 1990's, WalMart famously utilized parallel databases to gain radical efficiencies in supply chain management via item-

The Basics of Shared-Nothing Parallelism

- Hardware Economics
  - Commodity machines and network switches
  - Scale out systems incrementally;
- Parallel Dataflow
  - Tables partitioned by row across disks
  - Work partitioned by running replicas of each dataflow operator on multiple machines
  - Partitioned parallelism: dataflows partitioned across parallel workers, with each row in a dataflow delivered to one worker
  - Pipelined parallelism: while one operator is producing a dataflow, another operator can be consuming and processing it in parallel

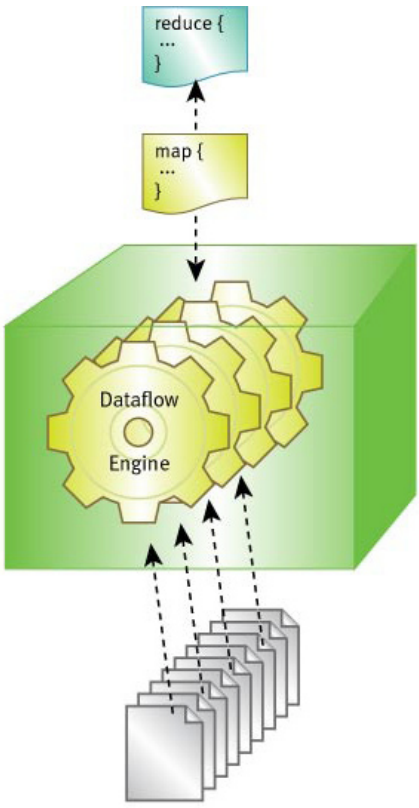
Greenplum

level inventory and historical sales information. In recent years, virtually every sizable enterprise has realized the importance of scalable solutions for data warehousing and analytics.

Parallel Programming with MapReduce (2000-present)

In the last decade, the importance of shared-nothing clusters was rediscovered in the design of web services like search engine infrastructure and messaging [Brewer01]. However, the implementation of those early web services was done by small teams of expert developers, much as the early parallel database systems were built. In this context, Google was differentiating itself as a company by developing a data-driven culture, in which employees are explicitly encouraged to (a) develop innovative solutions by analyzing the company's data assets, and (b) gain project approval from colleagues by using data analysis to overcome "conventional wisdom" and other institutional arguments against innovation [Kaushik06]. The growth of Google's data-driven culture was facilitated by getting the right analytic tools into the hands of employees: tools that could allow software developers to conveniently explore and analyze some of the largest data sets ever assembled.

The key tools that Google built for their developers were the MapReduce programming paradigm, and a proprietary runtime engine for internal use at Google [DeanGhemawat08]. At heart, MapReduce is a very simple dataflow programming model that passes data items through simple user-written code fragments. Google's MapReduce programs start with a large datafile that is broken into contiguous pieces called "splits". Each split is converted via user-defined parsing code into (key, value) pairs that are sent to a Map module, which invokes a usersupplied Map function on each pair, producing a new key and list of output values. Each (key, output_list) pair is passed to a Reduce module (possibly on another machine) that gathers them together, assembles them into groups by key, and then calls a user-supplied Reduce function



The MapReduce Programming Model

- Open data access
  - Any data source can be used for input
  - Input data can be in any format (the developer provides code to parse out simple records)
- Traditional code
  - Developers use familiar programming languages
  - Focus on simple single-node logic: dataflow operators to process individual records (Map) and groups of records (Reduce)
  - Programmer specifies an explicit pipeline of Map and Reduce operators.
- Partitioned parallel dataflow engine
  - Runs copies of each Mapper and Reducer on multiple nodes
  - Dataflows partitioned across parallel workers, with each record (Map) or group (Reduce) routed to a single node

to produce one reduced output list per group of mapped (key, output_list) pairs. Both the Map and Reduce modules utilize partition parallelism to enable many Map tasks (and many Reduce tasks) to run in parallel.

MapReduce has become very popular within Google for everything from the construction of their core web index, to simple programs written by a single developer in a half hour [DeanGhemawat08][1]. The MapReduce programming model has become available to programmers outside of Google as well, via the Hadoop open-source runtime. MapReduce is particularly attractive to developers for two main reasons:

- Data accessibility: Data is accessed from standard files, with no need for a priori definition of schemas or file formats, and no need to configure and load a database before getting answers. This allows developers to "wrangle" any file format that they have at hand; at a company like Google this includes web crawls (HTML), term occurrence data, clickstream logs, and advertising revenue history. The focus on standard files also means that developers can typically get work done without requesting permission from the "Keepers of the Data" that guard traditional IT shops.

- Language Familiarity: Most of the MapReduce programmer's work is done in familiar programming languages used by developers: Google's MapReduce uses C++, and Hadoop uses Java. This exposes massive data parallelism to developers within the context of their familiar development environment: editors, debuggers, and so on. By contrast, relatively few developers work with data-centric languages like SQL on a daily basis, and SQL experts tend to inhabit a different ecosystem (training, job title) than typical software developers.

Technical common ground: Parallel Dataflow

The MapReduce revolution is so recent that the dust has yet to settle on the new regime – there is still plenty of debate about how MapReduce and parallel RDBMSs fit together in a data-centric organization. Some database leaders have argued publicly that the MapReduce phenomenon is not a technical revolution at all – they characterize it as a reinvention of well-known parallel database techniques that is missing key database functionality (ACID storage, indexes, query optimization, etc.) [DeWittStonebraker08]. The MapReduce proponents argue that they neither need nor want a heavyweight database for

[1] In September 2007 alone, Google's MapReduce installation processed over 2,000 jobs that churned through 403,000 Terabytes of data. Perhaps more interesting than the data volume is the diversity of code that was written: that workload included over 4,000 unique Map functions, and over 2,400 unique Reduce functions [DeanGhemawat08].

many tasks, and they have no interest in coding in SQL. From their standpoint, MapReduce has revolutionized the developer ecosystem, providing them with easy access to parallelism over their own data, in their own language framework.

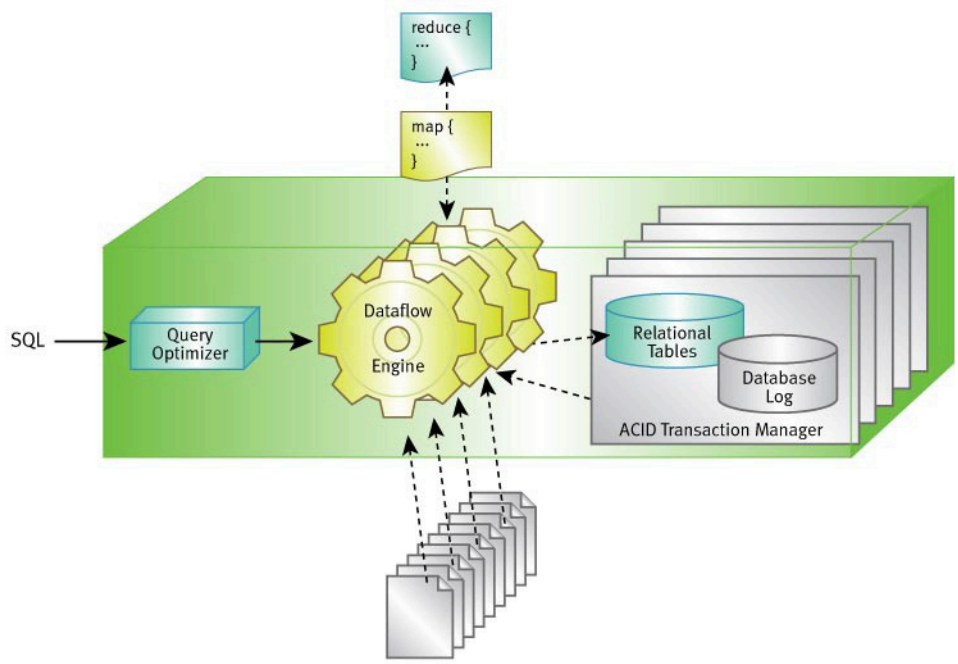
Both these arguments have merit. But the disconnect between these viewpoints can lead to inefficiencies and confusion in an organization trying to instill a broad data-driven culture. Consider what happens if the traditionally cautious IT department requires the use of a full RDBMS feature stack, and the maverick developers focus on the lightweight and programmer-friendly MapReduce framework. Data assets get partitioned across teams, as do in-house program for data analysis. Worse, two separate data cultures evolve within the organization, leading to destructive "data ownership" politics, and arguments over tools rather than solutions.

Despite the differences in programming interfaces and software philosophy, RDBMSs and MapReduce engines both are brought alive by the same "beating heart": a massively parallel dataflow engine, pumping data across a parallel network mesh, through high-performance bulk operations (join, map, reduce, etc.). Is it possible to take that core dataflow component, and provide interfaces for both ecosystems? In principle, this should be entirely natural. The main barriers come from software engineering realities. The tried-and-true parallel RDBMS engines were built in the 1980's and 90's with the dataflow engine embedded deep into the relational codebase. It is a tall order to extract the "beating heart" from those systems for reuse elsewhere. By contrast, MapReduce implementations like Hadoop provide none of the key features required of a DBMS. They have a lot of "heart", but the body-building required to replicate a full-featured RDBMS would take years.

Greenplum enters this arena from a unique direction. Greenplum began in the "heart transplant" business: its core technology effort was to take PostgreSQL, the best-of-breed open-source RDBMS, and insert a massively parallel dataflow engine into its core. Based on that success, Greenplum is now able to offer the first commercial implementation of MapReduce, built on that same core parallel technology. Because Greenplum's RDBMS and MapReduce share the same core engine, they are uniquely interoperable.

Introducing Greenplum MapReduce

Greenplum MapReduce provides a convenient, easy-to-program platform for massive data-parallelism. It implements a harness for parallel Map and Reduce functions, along with flexible data access to files, database records, and system services.



Greenplum allows developers to write Map and Reduce functions in a variety of popular scripting languages: the list currently includes Python and Perl. Support for these popular languages includes access to entire ecosystems of open-source packages via the Python Package Index (PyPi) and the Comprehensive Perl Archive Network (CPAN). This includes a host of features not usually found in an RDBMS: free-text analysis, statistical toolkits, graph algorithms, HTML and XML parsing, web connectivity (SOAP, REST, HTTP), and many more.

In terms of data access, Greenplum MapReduce provides developers with the familiar flexibility to access their data "where it lives": in files, websites, or even via arbitrary operating system commands. Greenplum provides this data without any of the overheads that developers often associate with traditional RDBMSs: no locking, logging or distributed "commit" protocols. On the other hand, for data that does need to be protected by a full-featured RDBMS, Greenplum MapReduce offers efficient native access to database records: it pushes MapReduce programs down into Greenplum's parallel database engine, without the cost of going "out-of-box" to get data from a separate DBMS over narrow client interfaces.

The Power of Synergy

Greenplum is unique in offering a commercial-grade implementation of MapReduce, providing a robust implementation of the open interfaces that enable and encourage developers to work with massive data sets. But the biggest advantage of Greenplum's implementation comes from its shared technology core, which unifies MapReduce and RDBMS functionality within a single parallel dataflow engine.

This unique architecture allows developers to mix and match data sources and programming styles. Greenplum's solution is also able to make MapReduce programs visible to SQL queries and vice-versa. This set of features enables a number of natural design patterns that are unique to Greenplum:

- MapReduce programs over high-performance database tables. Access to database data is trivial in Greenplum MapReduce: the MapReduce program simply specifies the database table name as its input. Because Greenplum database tables are partitioned across multiple machines, the initial Map phase is executed in the database engine directly on the local partition, providing fully parallel I/O with computation "pushed" to the data. By contrast, a standalone MapReduce engine would require the programmer to write data access routines into their MapReduce script. That extra programmer code would then access a remote database server via a connectivity protocol like JDBC, and pull the database records over to Map workers.

- SQL over external data sources. Greenplum's "External Table" facility allows files and data-producing programs to be registered as read-only tables in the database, and queried in SQL alongside database tables. External data is accessed and converted to records on the fly during query processing. Because these external tables can be stored or generated on an arbitrary number of nodes in the cluster, data access and conversion is a massively parallel process.

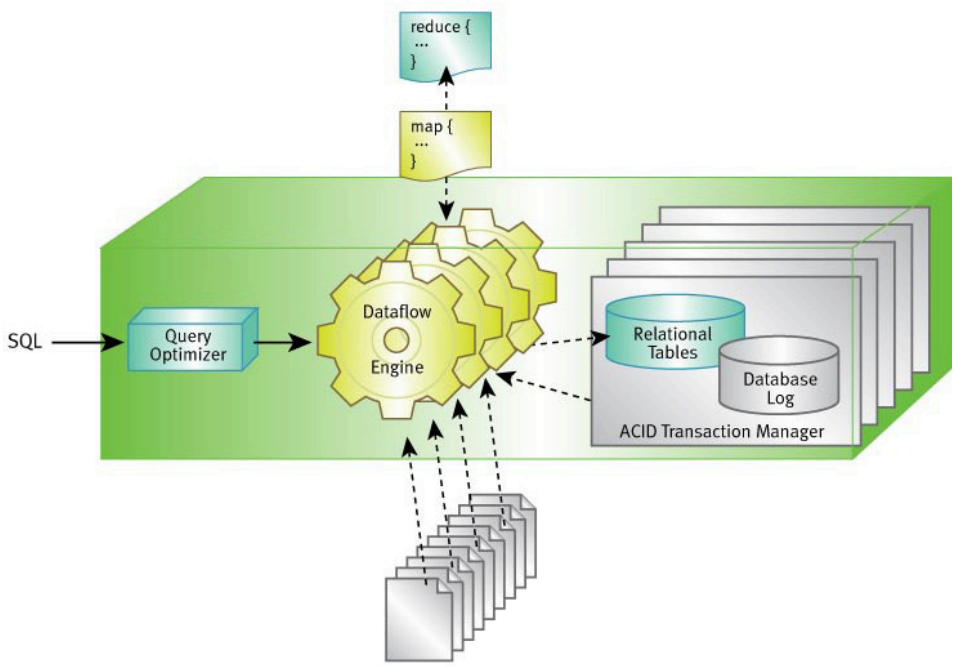
- Durable storage of MapReduce outputs. Many MapReduce programs run for hours, and provide important analytic results. Like standalone MapReduce implementations, Greenplum can store these results in a filesystem. But it is equally easy to store the results of Greenplum MapReduce in a Greenplum database, with full ACID durability guarantees, and the option to subsequently analyze those outputs via Business Intelligence tools, SQL queries, and other enterprise analytic software designed for databases. Again, because the MapReduce code runs in the same engine as the database, writing of output tables is fully parallelized and requires no remote connectivity overheads.

- Rich integration of MapReduce and SQL code. Greenplum's unique architecture removes barriers between code written in the MapReduce framework, and code written in SQL. Because Greenplum MapReduce scripts can be configured to flexibly access the database, they can use arbitrary SQL queries as input. In the other direction, Greenplum MapReduce scripts can be registered as "views" in the database, and used as virtual tables within SQL statements: the MapReduce job is run on the fly as part of the SQL query processing, and its

outputs are pipelined directly into the relational query plan. Greenplum's engine executes all code – SQL, Map functions, Reduce functions – on the same cluster of machines where the database is stored. This integration allows SQL and MapReduce developers to share code freely without performance penalties or the need to work with "adapter" software. This flexibility removes the overhead of cultural and political debates about the "right" programming framework within an organization.

MapReduce in Use

MapReduce supports simple, easy-to-program dataflows: a single data source piped into a chain of customizable Map and Reduce operators. As a result, MapReduce is especially well suited for parallelizing custom tasks over a single dataset.



Data extraction and transformation tasks fit this model well. Consider the example of an e-commerce website with free-text descriptions of products. As a first phase in categorizing products, we would like to automatically extract keywords from the HTML description text for each product. That is, we want to convert each free text description into a set of pairs of the form (productID, keyword).

MapReduce makes this easy. We configure it to route product pages (which may be in files, a database, or even on the Web) to a cluster of Greenplum servers, each running a Python Map operator. The Python Map code on each node repeatedly gets a product page, splits the product description text into a list of potential keywords, and then loops through the resulting list and

outputs (productID, keyword) pairs2. These can be routed to Python Reduce operators running on the cluster, which can gather up and count these pairs to produce outputs of the form (productID, keyword, occurrences), where the last field captures the number of times each keyword occurs in each product description. This output can be stored in a database table for use in subsequent tasks. For example, using this table, products can be "auto-categorized" by a simple SQL query that joins the MapReduce output with a table of keywords and product categories. As a very different example, the New York Times used a simple MapReduce program to convert years of scanned newspaper articles into digital text. The approach is to use parallel dataflow to "kick off" parallel computation. To do this in Greenplum, a list of image filenames can be piped into a cluster of Greenplum Map operators written in Perl. Each Map operator uses Perl's system command to execute an Optical Character Recognition program (e.g. the open source Tesseract tool) to convert the image file into text. No Reduce phase is required; the results of the OCR program can be written to text files, or loaded as text fields into a database table.

Both of these examples do information extraction, transformation and loading, often called ETL in the data warehousing business. Some database engines advertise the ability to do ELT: loading the data into the database before transforming it, to allow subsequent transformations to run in SQL. Greenplum's flexibility makes the reordering of the "L" phase completely fluid: data can be stored inside or outside the database, and accessed in either case by massively parallel code written in either MapReduce or SQL. So Greenplum easily enables either ETL or ELT, along with options like ET (in which the data is always stored outside the database) and LET (in which the raw form of the information is stored in the database.) This is the kind of flexibility that comes from decoupling the parallel dataflow engine, allowing it to interoperate with various storage and language interfaces.

The previous examples focused on data extraction and transformation, but MapReduce is also useful for deeper data mining and analytics. Many companies employ experts in statistics and finance, who increasingly want to run complex mathematical models over large volumes of data. Recently, there have been a number of tutorials and papers on easily implementing popular data mining techniques in parallel using MapReduce [KimballMichelsBisciglia07, ChuEtAl06]. A variety of sophisticated

[2] An important detail in handling free text is to canonicalize multiple forms of the same word: e.g. "driver", "drivers", and "driving" should all be converted to "drive" so they will match. Because Greenplum MapReduce provides access to Perl and Python's open-source libraries, we can use Python's nltk toolkit for Natural Language Processing to do this task – a two-line addition to the basic Map program sketched above.

data mining and machine learning algorithms have been expressed in this framework, including popular techniques for classification, clustering, regression, and dimensionality reduction. And in the Greenplum context, these algorithms can be flexibly combined with SQL and run over both database tables and files.

Conclusion

MapReduce and SQL are two useful interfaces that enable software developers to take advantage of parallel processing over big data sets. Until recently, SQL was targeted at enterprise application programmers accessing transactional records, and MapReduce was targeted at more general software developers manipulating files. This distinction was mostly an artifact of the limitations of systems in the marketplace, but has led to significant confusion and slowed the adoption of MapReduce as a programming model in traditional data-rich settings in the business world.

Greenplum's technical core competency is parallel data technology. By applying that expertise to both MapReduce and SQL programs, Greenplum has changed the landscape for parallel data processing, removing arbitrary barriers between programming styles and usage scenarios. The resulting Greenplum engine is a uniquely flexible and scalable data processing system, allowing flexible combinations of SQL and MapReduce, database tables and files.

Contact
Greenplum
1900 S. Norfolk Drive,
Suite 224
San Mateo, CA 94403
United States
+1 650 286 8012 (ph)
+1 650 286 8010 (fax)

Greenplum

References

[Codd70] E. F. Codd: A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13(6): 377-387 (1970)

[Gray78] Jim Gray: Notes on Data Base Operating Systems. In Michael J. Flynn, et al. (Eds.): Operating Systems, An Advanced Course. Lecture Notes in Computer Science 60. Springer, 1978: 393-481.

[BoralDeWitt83] Haran Boral, David J. DeWitt: Database Machines: An Idea Whose Time Passed? A Critique of the Future of Database Machines. International Workshop on Database Machines (IWDM) 1983: 166-187

[Brewer01] Eric A. Brewer: Lessons from Giant-Scale Services. IEEE Internet Computing 5(4): 46-55 (2001)

[Kaushik06] Avinash Kaushik. Web Analytics: An Hour a Day. Sybex Publishers, 2007.

[DeanGhemawat08] Jeffrey Dean, Sanjay Ghemawat: MapReduce:simplified data processing on large clusters. Commun. ACM 51(1): 107-113 (2008)

[DeWittStonebraker08] David J. DeWitt and Michael Stonebraker. MapReduce: A Major Step Backwards. The Database Column (weblog). January 17, 2008. http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html

[KimballMichelsBisciglia07] Aaron Kimball, Sierra Michels-Slettvet, and Christophe Bisciglia. Cluster Computing and MapReduce. Google Code University (website). Summer, 2007.

[ChuEtAl06] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Ng, and Kunle Olukotun. MapReduce for Machine Learning on Multicore. Advances in Neural Information Processing Systems (NIPS), December, 2006. Revision 1. August 2008.