Chapter 5
Data-Intensive Technologies for Cloud Computing

Anthony M. Middleton

5.1 Introduction

As a result of the continuing information explosion, many organizations are drowning in data and the resulting "data gap" or inability to process this information and use it effectively is increasing at an alarming rate. Data-intensive computing represents a new computing paradigm (Kouzes, Anderson, Elbert, Gorton, & Gracio, 2009) which can address the data gap using scalable parallel processing to allow government, commercial organizations, and research environments to process massive amounts of data and implement applications previously thought to be impractical or infeasible. Cloud computing provides the opportunity for organizations with limited internal resources to implement large-scale data-intensive computing applications in a cost-effective manner.

The fundamental challenges of data-intensive computing are managing and processing exponentially growing data volumes, significantly reducing associated data analysis cycles to support practical, timely applications, and developing new algorithms which can scale to search and process massive amounts of data. Researchers at LexisNexis believe that the answer to these challenges is a scalable, integrated computer systems hardware and software architecture designed for parallel processing of data-intensive computing applications. This chapter explores the challenges of data-intensive computing and offers an in-depth comparison of commercially available system architectures including the LexisNexis Data Analytics Supercomputer (DAS) also referred to as the LexisNexis High-Performance Computing Cluster (HPCC), and Hadoop, an open source implementation based on Google's MapReduce architecture.

Cloud computing emphasizes the ability to scale computing resources as needed without a large upfront investment in infrastructure and associated ongoing operational costs (Napper & Bientinesi, 2009; Reese, 2009; Velte, Velte, & Elsenpeter, 2009). Cloud computing services are typically categorized in three models:

A.M. Middleton (✉)
LexisNexis Risk Solutions, Boca Raton, FL, USA
e-mail: tony.middleton@lexisnexis.com

B. Furht, A. Escalante (eds.), *Handbook of Cloud Computing*,
DOI 10.1007/978-1-4419-6524-0_5, © Springer Science+Business Media, LLC 2010

(1) *Infrastructure as a Service (IaaS)*. Service includes provision of hardware and software for processing, data storage, networks and any required infrastructure for deployment of operating systems and applications which would normally be needed in a data center managed by the user; (2) *Platform as a Service (PaaS)*. Service includes programming languages and tools and an application delivery platform hosted by the service provider to support development and delivery of end-user applications; and (3) *Software as a Service (SaaS)*. Hosted software applications are provided and managed by the service provider for the end-user replacing locally-run applications with Web-based applications (Lenk, Klems, Nimis, Tai, & Sandholm, 2009; Levitt, 2009; Mell & Grance, 2009; Vaquero, Rodero-Merino, Caceres, & Lindner, 2009; Viega, 2009).

Data-intensive computing applications are implemented using either the IaaS model which allows the provisioning of scalable clusters of processors for data-parallel computing using various software architectures, or the PaaS model which provides a complete processing and application development environment including both infrastructure and platform components such as programming languages and applications development tools. Data-intensive computing can be implemented in a *public cloud* (cloud infrastructure and platform is publicly available from a cloud services provider) such as Amazon's Elastic Compute Cloud (EC2) and Elastic MapReduce or as a *private cloud* (cloud infrastructure and platform is operated solely for a specific organization and may exist internally or externally to the organization) (Mell & Grance, 2009). IaaS and PaaS implementations for data-intensive computing can be either dynamically provisioned in virtualized processing environments based on application scheduling and data processing requirements, or can be implemented as a persistent high-availability configuration. A persistent configuration has a performance advantage since it uses dedicated infrastructure instead of virtualized servers shared with other users.

5.1.1 Data-Intensive Computing Applications

Parallel processing approaches can be generally classified as either *compute-intensive*, or *data-intensive* (Skillicorn & Talia, 1998; Gorton, Greenfield, Szalay, & Williams, 2008; Johnston, 1998). Compute-intensive is used to describe application programs that are compute bound. Such applications devote most of their execution time to computational requirements as opposed to I/O, and typically require small volumes of data. Parallel processing of compute-intensive applications typically involves parallelizing individual algorithms within an application process, and decomposing the overall application process into separate tasks, which can then be executed in parallel on an appropriate computing platform to achieve overall higher performance than serial processing. In compute-intensive applications, multiple operations are performed simultaneously, with each operation addressing a particular part of the problem. This is often referred to as functional parallelism or control parallelism (Abbas, 2004).

Data-intensive is used to describe applications that are I/O bound or with a need to process large volumes of data (Gorton et al., 2008; Johnston, 1998; Gokhale, Cohen, Yoo, & Miller, 2008). Such applications devote most of their processing time to I/O and movement of data. Parallel processing of data-intensive applications typically involves partitioning or subdividing the data into multiple segments which can be processed independently using the same executable application program in parallel on an appropriate computing platform, then reassembling the results to produce the completed output data (Nyland, Prins, Goldberg, & Mills, 2000). The greater the aggregate distribution of the data, the more benefit there is in parallel processing of the data. Gorton et al. (2008) state that data-intensive processing requirements normally scale linearly according to the size of the data and are very amenable to straightforward parallelization. The fundamental challenges for data-intensive computing according to Gorton et al. (2008) are managing and processing exponentially growing data volumes, significantly reducing associated data analysis cycles to support practical, timely applications, and developing new algorithms which can scale to search and process massive amounts of data. Cloud computing can address these challenges with the capability to provision new computing resources or extend existing resources to provide parallel computing capabilities which scale to match growing data volumes (Grossman, 2009).

5.1.2 Data-Parallelism

Computer system architectures which can support data-parallel applications are a potential solution to terabyte and petabyte scale data processing requirements (Nyland et al., 2000; Ravichandran, Pantel, & Hovy, 2004). According to Agichtein and Ganti (2004), parallelization is considered to be an attractive alternative for processing extremely large collections of data such as the billions of documents on the Web (Agichtein, 2004). Nyland et al. (2000) define data-parallelism as a computation applied independently to each data item of a set of data which allows the degree of parallelism to be scaled with the volume of data. According to Nyland et al. (2000), the most important reason for developing data-parallel applications is the potential for scalable performance, and may result in several orders of magnitude performance improvement. The key issues with developing applications using data-parallelism are the choice of the algorithm, the strategy for data decomposition, load balancing on processing nodes, message passing communications between nodes, and the overall accuracy of the results (Nyland et al., 2000; Rencuzogullari & Dwarkadas, 2001). Nyland et al. (2000) also note that the development of a data-parallel application can involve substantial programming complexity to define the problem in the context of available programming tools, and to address limitations of the target architecture. Information extraction from and indexing of Web documents is typical of data-intensive processing which can derive significant performance benefits from data-parallel implementations since Web and other types of document collections can typically then be processed in parallel (Agichtein, 2004).

5.1.3 The "Data Gap"

The rapid growth of the Internet and World Wide Web has led to vast amounts of information available online. In addition, business and government organizations create large amounts of both structured and unstructured information which needs to be processed, analyzed, and linked. Vinton Cerf of Google has described this as an "Information Avalanche" and has stated "we must harness the Internet's energy before the information it has unleashed buries us" (Cerf, 2007). An IDC white paper sponsored by EMC estimated the amount of information currently stored in a digital form in 2007 at 281 exabytes and the overall compound growth rate at 57% with information in organizations growing at even a faster rate (Gantz et al., 2007). In another study of the so-called information explosion it was estimated that 95% of all current information exists in unstructured form with increased data processing requirements compared to structured information (Lyman & Varian, 2003). The storing, managing, accessing, and processing of this vast amount of data represents a fundamental need and an immense challenge in order to satisfy needs to search, analyze, mine, and visualize this data as information (Berman, 2008). In 2003, LexisNexis defined this issue as the "Data Gap": the ability to gather information is far outpacing organizational capacity to use it effectively.

Organizations build the applications to fill the storage they have available, and build the storage to fit the applications and data they have. But will organizations be able to do useful things with the information they have to gain full and innovative use of their untapped data resources? As organizational data grows, how will the "Data Gap" be addressed and bridged? Researchers at LexisNexis believe that the answer is a scalable computer systems hardware and software architecture designed for data-intensive computing applications which can scale to processing billions of records per second (BORPS) (Note: the term BORPS was introduced by Seisint, Inc. in 2002. Seisint was acquired by LexisNexis in 2004). What are the characteristics of data-intensive computing systems and what system architectures are available to organizations to implement data-intensive computing applications? Can these capabilities be implemented using cloud computing to reduce risk and upfront investment in infrastructure and to allow a pay-as-you-go model? This chapter will explore those issues and offer a comparison of commercially available system architectures.

5.2 Characteristics of Data-Intensive Computing Systems

The National Science Foundation believes that data-intensive computing requires a "fundamentally different set of principles" than current computing approaches (NSF, 2009). Through a funding program within the Computer and Information Science and Engineering area, the NSF is seeking to "increase understanding of the capabilities and limitations of data-intensive computing." The key areas of focus are:

- Approaches to parallel programming to address the parallel processing of data on data-intensive systems
- Programming abstractions including models, languages, and algorithms which allow a natural expression of parallel processing of data
- Design of data-intensive computing platforms to provide high levels of reliability, efficiency, availability, and scalability.
- Identifying applications that can exploit this computing paradigm and determining how it should evolve to support emerging data-intensive applications.

Pacific Northwest National Labs has defined data-intensive computing as "capturing, managing, analyzing, and understanding data at volumes and rates that push the frontiers of current technologies" (Kouzes et al., 2009; PNNL, 2008). They believe that to address the rapidly growing data volumes and complexity requires "epochal advances in software, hardware, and algorithm development" which can scale readily with size of the data and provide effective and timely analysis and processing results. The HPCC architecture developed by LexisNexis represents such an advance in capabilities.

5.2.1 Processing Approach

Current data-intensive computing platforms use a "divide and conquer" parallel processing approach combining multiple processors and disks in large computing clusters connected using high-speed communications switches and networks which allows the data to be partitioned among the available computing resources and processed independently to achieve performance and scalability based on the amount of data (Fig. 5.1). Buyya, Yeo, Venugopal, Broberg, and Brandic (2009) define a cluster as "a type of parallel and distributed system, which consists of a collection

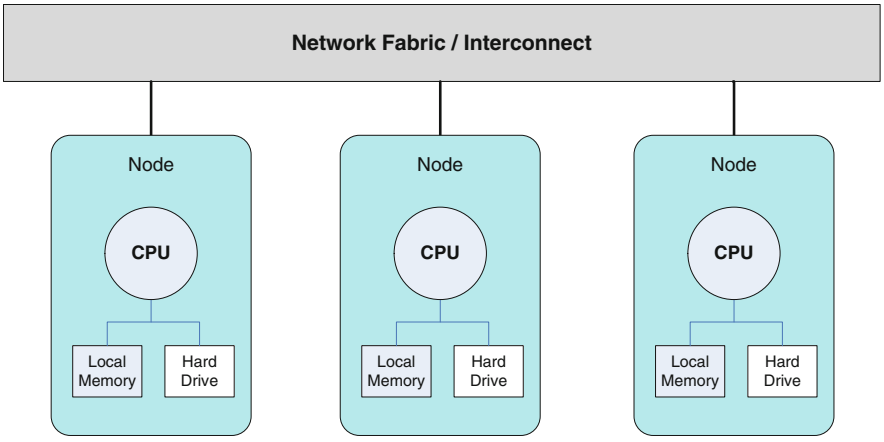


Fig. 5.1 Shared nothing computing cluster

of inter-connected stand-alone computers working together as a single integrated computing resource." This approach to parallel processing is often referred to as a "shared nothing" approach since each node consisting of processor, local memory, and disk resources shares nothing with other nodes in the cluster. In parallel computing this approach is considered suitable for data processing problems which are "embarrassingly parallel" , i.e. where it is relatively easy to separate the problem into a number of parallel tasks and there is no dependency or communication required between the tasks other than overall management of the tasks. These types of data processing problems are inherently adaptable to various forms of distributed computing including clusters and data grids and cloud computing.

5.2.2 Common Characteristics

There are several important common characteristics of data-intensive computing systems that distinguish them from other forms of computing. First is the principle of collocation of the data and programs or algorithms to perform the computation. To achieve high performance in data-intensive computing, it is important to minimize the movement of data (Gray, 2008). In direct contrast to other types of computing and supercomputing which utilize data stored in a separate repository or servers and transfer the data to the processing system for computation, data-intensive computing uses distributed data and distributed file systems in which data is located across a cluster of processing nodes, and instead of moving the data, the program or algorithm is transferred to the nodes with the data that needs to be processed. This principle – "Move the code to the data" – which was designed into the data-parallel processing architecture implemented by Seisint in 2003, is extremely effective since program size is usually small in comparison to the large datasets processed by data-intensive systems and results in much less network traffic since data can be read locally instead of across the network. This characteristic allows processing algorithms to execute on the nodes where the data resides reducing system overhead and increasing performance (Gorton et al., 2008).

A second important characteristic of data-intensive computing systems is the programming model utilized. Data-intensive computing systems utilize a machine-independent approach in which applications are expressed in terms of high-level operations on data, and the runtime system transparently controls the scheduling, execution, load balancing, communications, and movement of programs and data across the distributed computing cluster (Bryant, 2008). The programming abstraction and language tools allow the processing to be expressed in terms of data flows and transformations incorporating new dataflow programming languages and shared libraries of common data manipulation algorithms such as sorting. Conventional supercomputing and distributed computing systems typically utilize machine dependent programming models which can require low-level programmer control of processing and node communications using conventional imperative programming languages and specialized software packages which adds complexity to the parallel

programming task and reduces programmer productivity. A machine dependent programming model also requires significant tuning and is more susceptible to single points of failure.

A third important characteristic of data-intensive computing systems is the focus on reliability and availability. Large-scale systems with hundreds or thousands of processing nodes are inherently more susceptible to hardware failures, communications errors, and software bugs. Data-intensive computing systems are designed to be fault resilient. This includes redundant copies of all data files on disk, storage of intermediate processing results on disk, automatic detection of node or processing failures, and selective re-computation of results. A processing cluster configured for data-intensive computing is typically able to continue operation with a reduced number of nodes following a node failure with automatic and transparent recovery of incomplete processing.

A final important characteristic of data-intensive computing systems is the inherent scalability of the underlying hardware and software architecture. Data-intensive computing systems can typically be scaled in a linear fashion to accommodate virtually any amount of data, or to meet time-critical performance requirements by simply adding additional processing nodes to a system configuration in order to achieve billions of records per second processing rates (BORPS). The number of nodes and processing tasks assigned for a specific application can be variable or fixed depending on the hardware, software, communications, and distributed file system architecture. This scalability allows computing problems once considered to be intractable due to the amount of data required or amount of processing time required to now be feasible and affords opportunities for new breakthroughs in data analysis and information processing.

5.2.3 Grid Computing

A similar computing paradigm known as grid computing has gained popularity primarily in research environments (Abbas, 2004). A computing grid is typically heterogeneous in nature (nodes can have different processor, memory, and disk resources), and consists of multiple disparate computers distributed across organizations and often geographically using wide-area networking communications usually with relatively low-bandwidth. Grids are typically used to solve complex computational problems which are compute-intensive requiring only small amounts of data for each processing node. A variation known as data grids allow shared repositories of data to be accessed by a grid and utilized in application processing, however the low-bandwidth of data grids limit their effectiveness for large-scale data-intensive applications.

In contrast, data-intensive computing systems are typically homogeneous in nature (nodes in the computing cluster have identical processor, memory, and disk resources), use high-bandwidth communications between nodes such as gigabit Ethernet switches, and are located in close proximity in a data center using

high-density hardware such as rack-mounted blade servers. The logical file system typically includes all the disks available on the nodes in the cluster and data files are distributed across the nodes as opposed to a separate shared data repository such as a storage area network which would require data to be moved to nodes for processing. Geographically dispersed grid systems are more difficult to manage, less reliable, and less secure than data-intensive computing systems which are usually located in secure data center environments.

5.2.4 Applicability to Cloud Computing

Cloud computing can take many shapes. Most visualize the cloud as the Internet or Web which is often depicted in this manner, but a more general definition is that cloud computing shifts the location of the computing resources and infrastructure providing computing applications to the network (Vaquero et al., 2009). Software accessible through the cloud becomes a service, application platforms accessible through the cloud to develop and deliver new applications become a service, and hardware and software to create infrastructure and virtual data center environments accessible through the cloud becomes a service (Weiss, 2007). Other characteristics usually associated with cloud computing include a reduction in the costs associated with management of hardware and software resources (Hayes, 2008), pay-per-use or pay-as-you-go access to software applications and on-demand computing resources (Vaquero et al., 2009), dynamic provisioning of infrastructure and scalability of resources to match the size of the data and computing requirements which is directly applicable to the characteristics of data-intensive computing (Grossman & Gu, 2009). Buyya et al. (2009) provide the following comprehensive definition of a cloud: "A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumer."

The cloud computing models directly applicable to data-intensive computing characteristics are Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). IaaS typically includes a large pool of configurable virtualized resources which can include hardware, operating systems, middleware, and development platforms or other software services which can be scaled to accommodate varying processing loads (Vaquero et al., 2009). The computing clusters typically used for data-intensive processing can be provided in this model. Processing environments such as Hadoop MapReduce and LexisNexis HPCC which include application development platform capabilities in addition to basic infrastructure implement the Platform as a Service (PaaS) model. Applications with a high degree of data-parallelism and a requirement to process very large datasets can take advantage of cloud computing and IaaS or PaaS using hundreds of computers provisioned for a short time instead of one or a small number of computers for a long time (Armbrust

et al., 2009). According to Armbrust et al. in a University of California Berkeley research report (Armbrust et al., 2009), this processing model is particularly well-suited to data analysis and other applications that can benefit from parallel batch processing. However, the user cost/benefit analysis should also include the cost of moving large datasets into the cloud in addition the speedup and lower processing cost offered by the IaaS and PaaS models.

5.3 Data-Intensive System Architectures

A variety of system architectures have been implemented for data-intensive and large-scale data analysis applications including parallel and distributed relational database management systems which have been available to run on shared nothing clusters of processing nodes for more than two decades (Pavlo et al., 2009). These include database systems from Teradata, Netezza, Vertica, and Exadata/Oracle and others which provide high-performance parallel database platforms. Although these systems have the ability to run parallel applications and queries expressed in the SQL language, they are typically not general-purpose processing platforms and usually run as a back-end to a separate front-end application processing system. Although this approach offers benefits when the data utilized is primarily structured in nature and fits easily into the constraints of a relational database, and often excels for transaction processing applications, most data growth is with data in unstructured form (Gantz et al., 2007) and new processing paradigms with more flexible data models were needed. Internet companies such as Google, Yahoo, Microsoft, Facebook, and others required a new processing approach to effectively deal with the enormous amount of Web data for applications such as search engines and social networking. In addition, many government and business organizations were overwhelmed with data that could not be effectively processed, linked, and analyzed with traditional computing approaches.

Several solutions have emerged including the MapReduce architecture pioneered by Google and now available in an open-source implementation called Hadoop used by Yahoo, Facebook, and others. LexisNexis, an acknowledged industry leader in information services, also developed and implemented a scalable platform for data-intensive computing which is used by LexisNexis and other commercial and government organizations to process large volumes of structured and unstructured data. These approaches will be explained and contrasted in terms of their overall structure, programming model, file systems, and applicability to cloud computing in the following sections. Similar approaches using commodity computing clusters including Sector/Sphere (Grossman & Gu, 2008; Grossman, Gu, Sabala, & Zhang, 2009; Gu & Grossman, 2009), SCOPE/Cosmos (Chaiken et al., 2008), DryadLINQ (Yu, Gunda, & Isard, 2009), Meandre (Llor et al., 2008), and GridBatch (Liu & Orban, 2008) recently described in the literature are also suitable for data-intensive cloud computing applications and represent additional alternatives.

5.3.1 Google MapReduce

The MapReduce architecture and programming model pioneered by Google is an example of a modern systems architecture designed for processing and analyzing large datasets and is being used successfully by Google in many applications to process massive amounts of raw Web data (Dean & Ghemawat, 2004). The MapReduce architecture allows programmers to use a functional programming style to create a map function that processes a key-value pair associated with the input data to generate a set of intermediate key-value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key (Dean & Ghemawat, 2004). According to Dean and Ghemawat (2004), the MapReduce programs can be used to compute derived data from documents such as inverted indexes and the processing is automatically parallelized by the system which executes on large clusters of commodity type machines, highly scalable to thousands of machines. Since the system automatically takes care of details like partitioning the input data, scheduling and executing tasks across a processing cluster, and managing the communications between nodes, programmers with no experience in parallel programming can easily use a large distributed processing environment.

The programming model for MapReduce architecture is a simple abstraction where the computation takes a set of input key-value pairs associated with the input data and produces a set of output key-value pairs. The overall model for this process is shown in Fig. 5.2. In the Map phase, the input data is partitioned into input splits

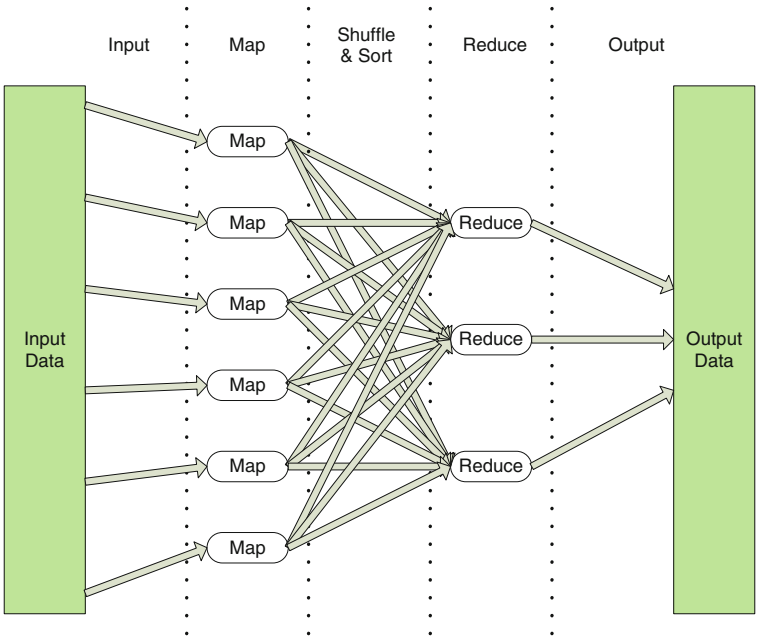


Fig. 5.2 MapReduce processing architecture (O'Malley, 2008)

and assigned to Map tasks associated with processing nodes in the cluster. The Map task typically executes on the same node containing its assigned partition of data in the cluster. These Map tasks perform user-specified computations on each input key-value pair from the partition of input data assigned to the task, and generates a set of intermediate results for each key. The shuffle and sort phase then takes the intermediate data generated by each Map task, sorts this data with intermediate data from other nodes, divides this data into regions to be processed by the reduce tasks, and distributes this data as needed to nodes where the Reduce tasks will execute. All Map tasks must complete prior to the shuffle and sort and reduce phases. The number of Reduce tasks does not need to be the same as the number of Map tasks. The Reduce tasks perform additional user-specified operations on the intermediate data possibly merging values associated with a key to a smaller set of values to produce the output data. For more complex data processing procedures, multiple MapReduce calls may be linked together in sequence.

Figure 5.3 shows the MapReduce architecture and key-value processing in more detail. The input data can consist of multiple input files. Each Map task will produce an intermediate output file for each key region assigned based on the number of Reduce tasks R assigned to the process (hash(key) modulus R). The reduce function then "pulls" the intermediate files, sorting and merging the files for a specific region from all the Map tasks. To minimize the amount of data transferred across the network, an optional Combiner function can be specified which is executed on the same node that performs a Map task. The combiner code is usually the same as

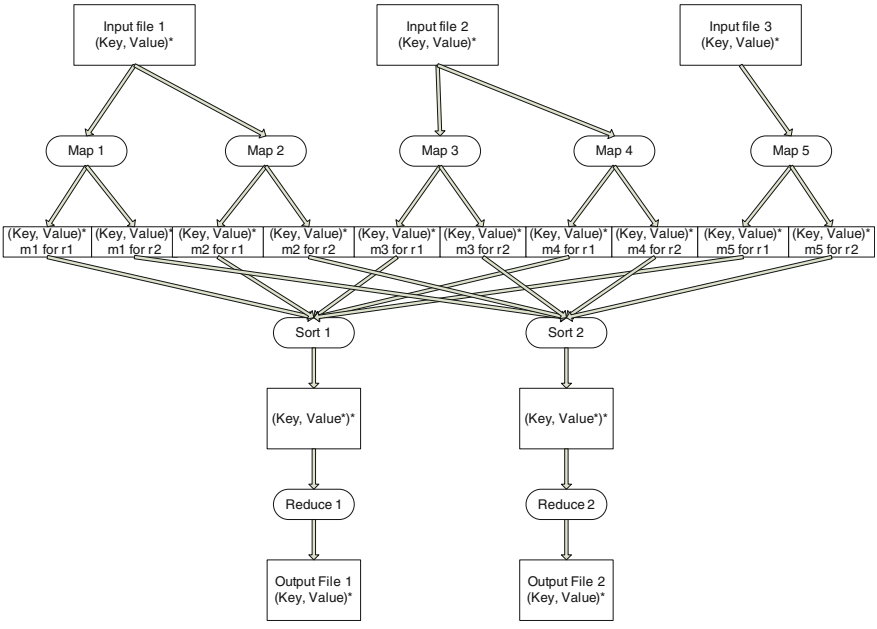


Fig. 5.3 MapReduce key-value processing (Nicosia, 2009)

the reducer function code which does partial merging and reducing of data for the local partition, then writes the intermediate files to be distributed to the Reduce tasks. The output of the Reduce function is written as the final output file. In the Google implementation of MapReduce, functions are coded in the C++ programming language.

Underlying and overlayed with the MapReduce architecture is the Google File System (GFS). GFS was designed to be a high-performance, scalable distributed file system for very large data files and data-intensive applications providing fault tolerance and running on clusters of commodity hardware (Ghemawat, Gobioff, & Leung, 2003). GFS is oriented to very large files dividing and storing them in fixed-size chunks of 64 Mb by default which are managed by nodes in the cluster called chunkservers. Each GFS consists of a single master node acting as a nameserver and multiple nodes in the cluster acting as chunkservers using a commodity Linux-based machine (node in a cluster) running a user-level server process. Chunks are stored in plain Linux files which are extended only as needed and replicated on multiple nodes to provide high-availability and improve performance. Secondary nameservers provide backup for the master node. The large chunk size reduces the need for MapReduce clients programs to interact with the master node, allows filesystem metadata to be kept in memory in the master node improving performance, and allows many operations to be performed with a single read on a chunk of data by the MapReduce client. Ideally, input splits for MapReduce operations are the size of a GFS chunk. GFS has proven to be highly effective for data-intensive computing on very large files, but is less effective for small files which can cause hot spots if many MapReduce tasks are accessing the same file.

Google has implemented additional tools using the MapReduce and GFS architecture to improve programmer productivity and to enhance data analysis and processing of structured and unstructured data. Since the GFS filesystem is primarily oriented to sequential processing of large files, Google has also implemented a scalable, high-availability distributed storage system for structured data with dynamic control over data format with keyed random access capabilities (Chang et al., 2006). Data is stored in Bigtable as a sparse, distributed, persistent multi-dimensional sorted map structured which is indexed by a row key, column key, and a timestamp. Rows in a Bigtable are maintained in order by row key, and row ranges become the unit of distribution and load balancing called a tablet. Each cell of data in a Bigtable can contain multiple instances indexed by the timestamp. Bigtable uses GFS to store both data and log files. The API for Bigtable is flexible providing data management functions like creating and deleting tables, and data manipulation functions by row key including operations to read, write, and modify data. Index information for Bigtables utilize tablet information stored in structures similar to a B+Tree. MapReduce applications can be used with Bigtable to process and transform data, and Google has implemented many large-scale applications which utilize Bigtable for storage including Google Earth.

Google has also implemented a high-level language for performing parallel data analysis and data mining using the MapReduce and GFS architecture called Sawzall and a workflow management and scheduling infrastructure for Sawzall jobs called

Workqueue (Pike, Dorward, Griesemer, & Quinlan, 2004). According to Pike et al. (2004), although C++ in standard MapReduce jobs is capable of handling data analysis tasks, it is more difficult to use and requires considerable effort by programmers. For most applications implemented using Sawzall, the code is much simpler and smaller than the equivalent C++ by a factor of 10 or more. A Sawzall program defines operations on a single record of the data, the language does not allow examining multiple input records simultaneously and one input record cannot influence the processing of another. An emit statement allows processed data to be output to an external aggregator which provides the capability for entire files of records and data to be processed using a Sawzall program. The system operates in a batch mode in which a user submits a job which executes a Sawzall program on a fixed set of files and data and collects the output at the end of a run. Sawzall jobs can be chained to support more complex procedures. Sawzall programs are compiled into an intermediate code which is interpreted during runtime execution. Several reasons are cited by Pike et al. (2004) why a new language is beneficial for data analysis and data mining applications: (1) a programming language customized for a specific problem domain makes resulting programs "clearer, more compact, and more expressive"; (2) aggregations are specified in the Sawzall language so that the programmer does not have to provide one in the Reduce task of a standard MapReduce program; (3) a programming language oriented to data analysis provides a more natural way to think about data processing problems for large distributed datasets; and (4) Sawzall programs are significantly smaller that equivalent C++ MapReduce programs and significantly easier to program.

Google does not currently make available its MapReduce architecture in a public cloud computing IaaS or PaaS environment. Google however does provide the Google Apps Engine as a public cloud computing PaaS environment (Lenk et al., 2009; Vaquero et al., 2009).

5.3.2 Hadoop

Hadoop is an open source software project sponsored by The Apache Software Foundation (http://www.apache.org). Following the publication in 2004 of the research paper describing Google MapReduce (Dean & Ghemawat, 2004), an effort was begun in conjunction with the existing Nutch project to create an open source implementation of the MapReduce architecture (White, 2009). It later became an independent subproject of Lucene, was embraced by Yahoo! after the lead developer for Hadoop became an employee, and became an official Apache top-level project in February of 2006. Hadoop now encompasses multiple subprojects in addition to the base core, MapReduce, and HDFS distributed filesystem. These additional subprojects provide enhanced application processing capabilities to the base Hadoop implementation and currently include Avro, Pig, HBase, ZooKeeper, Hive, and Chukwa. More information can be found at the Apache Web site.

The Hadoop MapReduce architecture is functionally similar to the Google implementation except that the base programming language for Hadoop is Java instead of C++. The implementation is intended to execute on clusters of commodity processors (Fig. 5.4) utilizing Linux as the operating system environment, but can also be run on a single system as a learning environment. Hadoop clusters also utilize the "shared nothing" distributed processing paradigm linking individual systems with local processor, memory, and disk resources using high-speed communications switching capabilities typically in rack-mounted configurations. The flexibility of Hadoop configurations allows small clusters to be created for testing and development using desktop systems or any system running Unix/Linux providing a JVM environment, however production clusters typically use homogeneous rack-mounted processors in a data center environment.

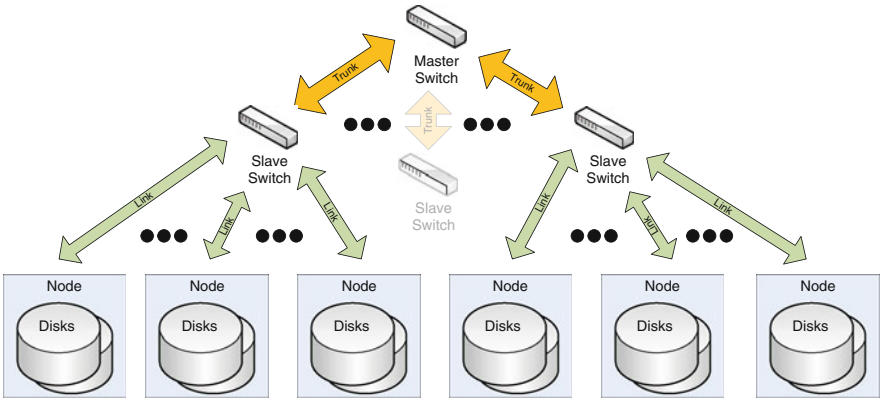


Fig. 5.4 Commodity hardware cluster (O'Malley, 2008)

The Hadoop MapReduce architecture is similar to the Google implementation creating fixed-size input splits from the input data and assigning the splits to Map tasks. The local output from the Map tasks is copied to Reduce nodes where it is sorted and merged for processing by Reduce tasks which produce the final output as shown in Fig. 5.5.

Hadoop implements a distributed data processing scheduling and execution environment and framework for MapReduce jobs. A MapReduce job is a unit of work that consists of the input data, the associated Map and Reduce programs, and user-specified configuration information (White, 2009). The Hadoop framework utilizes a master/slave architecture with a single master server called a jobtracker and slave servers called tasktrackers, one per node in the cluster. The jobtracker is the communications interface between users and the framework and coordinates the execution of MapReduce jobs. Users submit jobs to the jobtracker, which puts them in a job queue and executes them on a first-come/first-served basis. The jobtracker manages the assignment of Map and Reduce tasks to the tasktracker nodes which then execute these tasks. The tasktrackers also handle data movement between the Map and Reduce phases of job execution. The Hadoop framework assigns the Map tasks to

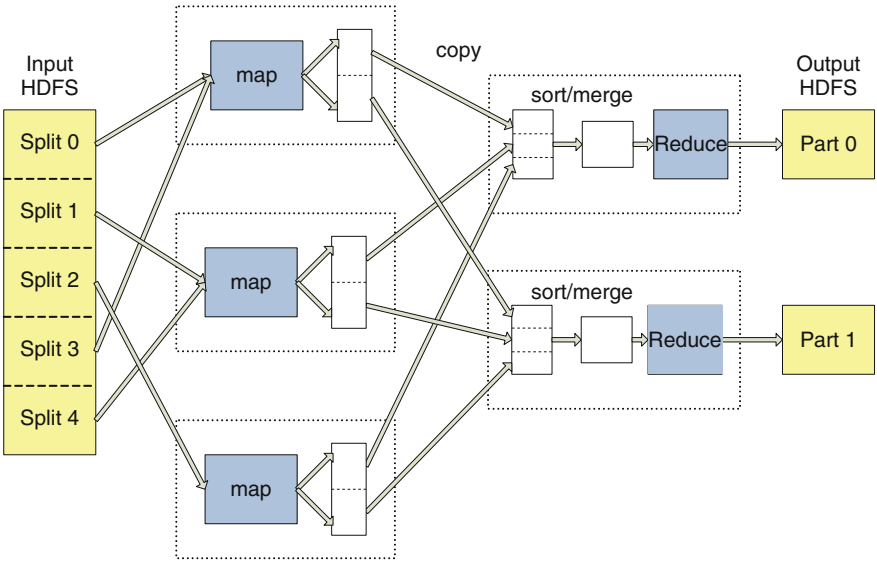


Fig. 5.5 Hadoop MapReduce (White, 2008)

every node where the input data splits are located through a process called data locality optimization. The number of Reduce tasks is determined independently and can be user-specified and can be zero if all of the work can be accomplished by the Map tasks. As with the Google MapReduce implementation, all Map tasks must complete before the shuffle and sort phase can occur and Reduce tasks initiated. The Hadoop framework also supports Combiner functions which can reduce the amount of data movement in a job.

The Hadoop framework also provides an API called Streaming to allow Map and Reduce functions to be written in languages other than Java such as Ruby and Python and provides an interface called Pipes for C++.

Hadoop includes a distributed file system called HDFS which is analogous to GFS in the Google MapReduce implementation. A block in HDFS is equivalent to a chunk in GFS and is also very large, 64 Mb by default but 128 Mb is used in some installations. The large block size is intended to reduce the number of seeks and improve data transfer times. Each block is an independent unit stored as a dynamically allocated file in the Linux local filesystem in a datanode directory. If the node has multiple disk drives, multiple datanode directories can be specified. An additional local file per block stores metadata for the block. HDFS also follows a master/slave architecture which consists of a single master server that manages the distributed filesystem namespace and regulates access to files by clients called the Namenode. In addition, there are multiple Datanodes, one per node in the cluster, which manage the disk storage attached to the nodes and assigned to Hadoop. The Namenode determines the mapping of blocks to Datanodes. The Datanodes are responsible for serving read and write requests from filesystem clients such as

MapReduce tasks, and they also perform block creation, deletion, and replication based on commands from the Namenode. An HDFS system can include additional secondary Namenodes which replicate the filesystem metadata, however there are no hot failover services. Each datanode block also has replicas on other nodes based on system configuration parameters (by default there are 3 replicas for each datanode block). In the Hadoop MapReduce execution environment it is common for a node in a physical cluster to function as both a Tasktracker and a datanode (Venner, 2009). The HDFS system architecture is shown in Fig. 5.6.

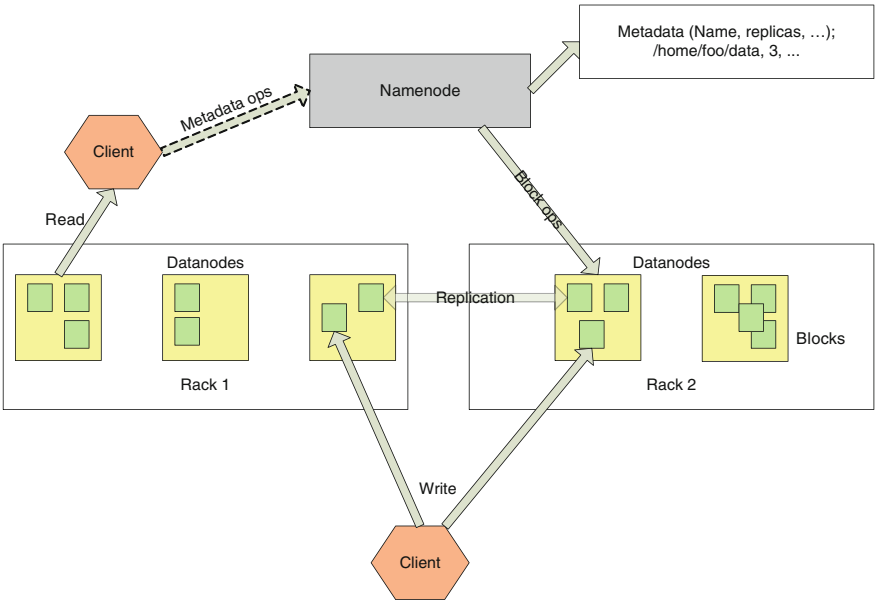


Fig. 5.6 HDFS architecture (Borthakur, 2008)

The Hadoop execution environment supports additional distributed data processing capabilities which are designed to run using the Hadoop MapReduce architecture. Several of these have become official Hadoop subprojects within the Apache Software Foundation. These include HBase, a distributed column-oriented database which provides similar random access read/write capabilities as and is modeled after Bigtable implemented by Google. HBase is not relational, and does not support SQL, but provides a Java API and a command-line shell for table management. Hive is a data warehouse system built on top of Hadoop that provides SQL-like query capabilities for data summarization, ad-hoc queries, and analysis of large datasets. Other Apache sanctioned projects for Hadoop include Avro – A data serialization system that provides dynamic integration with scripting languages, Chukwa – a data collection system for managing large distributed systems, ZooKeeper – a high-performance coordination service for distributed applications,

and Pig – a high-level data-flow language and execution framework for parallel computation.

Pig is high-level dataflow-oriented language and execution environment originally developed at Yahoo! ostensibly for the same reasons that Google developed the Sawzall language for its MapReduce implementation – to provide a specific language notation for data analysis applications and to improve programmer productivity and reduce development cycles when using the Hadoop MapReduce environment. Working out how to fit many data analysis and processing applications into the MapReduce paradigm can be a challenge, and often requires multiple MapReduce jobs (White, 2009). Pig programs are automatically translated into sequences of MapReduce programs if needed in the execution environment. In addition Pig supports a much richer data model which supports multi-valued, nested data structures with tuples, bags, and maps. Pig supports a high-level of user customization including user-defined special purpose functions and provides capabilities in the language for loading, storing, filtering, grouping, de-duplication, ordering, sorting, aggregation, and joining operations on the data (Olston, Reed, Srivastava, Kumar, & Tomkins, 2008a). Pig is an imperative dataflow-oriented language (language statements define a dataflow for processing). An example program is shown in Fig. 5.7. Pig runs as a client-side application which translates Pig programs into MapReduce jobs and then runs them on an Hadoop cluster. Figure 5.8 shows how the program listed in Fig. 5.7 is translated into a sequence of MapReduce jobs. Pig compilation and execution stages include a parser, logical optimizer, MapReduce compiler, MapReduce optimizer, and the Hadoop Job Manager (Gates et al., 2009).

According to Yahoo! where more than 40% of Hadoop production jobs and 60% of ad-hoc queries are now implemented using Pig, Pig programs are 1/20th the size of the equivalent MapReduce program and take 1/16th the time to develop (Olston, 2009). Yahoo! uses 12 standard benchmarks (called the PigMix) to test Pig performance versus equivalent MapReduce performance from release to release. With the

```
visits      = load '/data/visits' as (user, url, time);
gVisits     = group visits by url;
visitCounts = foreach gVisits generate url, count(urlVisits);

urlInfo     = load '/data/urlInfo' as (url, category, pRank);

visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';
```

Fig. 5.7 Sample pig latin program (Olston et al., 2008a)

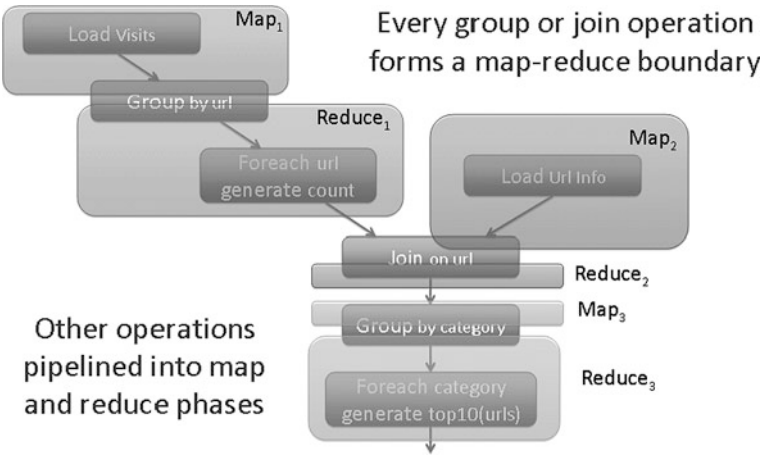


Fig. 5.8 Pig program translation to MapReduce (Olston et al., 2008a)

current release, Pig programs take approximately 1.5 times longer than the equivalent MapReduce (http://wiki.apache.org/pig/PigMix). Additional optimizations are being implemented that should reduce this performance gap further.

Hadoop is available in both public and private cloud computing environments. Amazon's EC2 cloud computing platform now includes Amazon Elastic MapReduce (http://aws.amazon.com/elasticmapreduce/) which allows users to provision as much capacity as needed for data-intensive computing applications. Data for MapReduce applications can be loaded to the HDFS directly from Amazon's S3 (Simple Storage Service).

5.3.3 LexisNexis HPCC

LexisNexis, an industry leader in data content, data aggregation, and information services independently developed and implemented a solution for data-intensive computing called the HPCC (High-Performance Computing Cluster) which is also referred to as the Data Analytics Supercomputer (DAS). The LexisNexis vision for this computing platform is depicted in Fig. 5.9. The development of this computing platform by the Seisint subsidiary of LexisNexis began in 1999 and applications were in production by late 2000. The LexisNexis approach also utilizes commodity clusters of hardware running the Linux operating system as shown in Figs. 5.1 and 5.4. Custom system software and middleware components were developed and layered on the base Linux operating system to provide the execution environment and distributed filesystem support required for data-intensive computing. Because LexisNexis recognized the need for a new computing paradigm to address its growing volumes of data, the design approach included the definition of a new high-level language for parallel data processing called ECL (Enterprise Data Control

Language). The power, flexibility, advanced capabilities, speed of development, and ease of use of the ECL programming language is the primary distinguishing factor between the LexisNexis HPCC and other data-intensive computing solutions. The following provides an overview of the HPCC systems architecture and the ECL language and a general comparison to the Hadoop MapReduce architecture and platform.

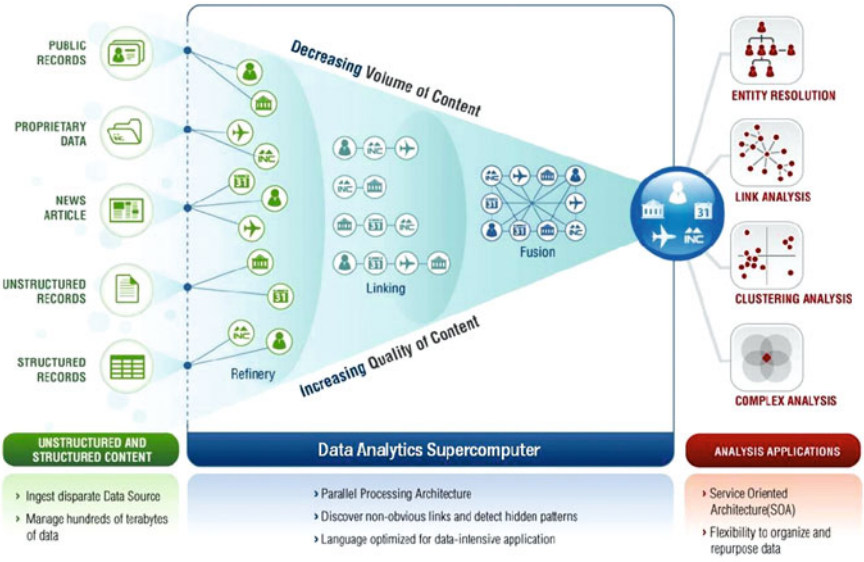


Fig. 5.9 LexisNexis vision for a data analytics supercomputer

LexisNexis developers recognized that to meet all the requirements of data-intensive computing applications in an optimum manner required the design and implementation of two distinct processing environments, each of which could be optimized independently for its parallel data processing purpose. The first of these platforms is called a Data Refinery whose overall purpose is the general processing of massive volumes of raw data of any type for any purpose but typically used for data cleansing and hygiene, ETL processing of the raw data (extract, transform, load), record linking and entity resolution, large-scale ad-hoc analysis of data, and creation of keyed data and indexes to support high-performance structured queries and data warehouse applications. The Data Refinery is also referred to as Thor, a reference to the mythical Norse god of thunder with the large hammer symbolic of crushing large amounts of raw data into useful information. A Thor system is similar in its hardware configuration, function, execution environment, filesystem, and capabilities to the Hadoop MapReduce platform, but offers significantly higher performance in equivalent configurations.

The second of the parallel data processing platforms designed and implemented by LexisNexis is called the Data Delivery Engine. This platform is designed as an

online high-performance structured query and analysis platform or data warehouse delivering the parallel data access processing requirements of online applications through Web services interfaces supporting thousands of simultaneous queries and users with sub-second response times. High-profile online applications developed by LexisNexis such as Accurint utilize this platform. The Data Delivery Engine is also referred to as Roxie, which is an acronym for Rapid Online XML Inquiry Engine. Roxie uses a special distributed indexed filesystem to provide parallel processing of queries. A Roxie system is similar in its function and capabilities to Hadoop with HBase and Hive capabilities added, but provides significantly higher throughput since it uses a more optimized execution environment and filesystem for high-performance online processing. Most importantly, both Thor and Roxie systems utilize the same ECL programming language for implementing applications, increasing continuity and programmer productivity.

The Thor system cluster is implemented using a master/slave approach with a single master node and multiple slave nodes for data parallel processing. Each of the slave nodes is also a data node within the distributed file system for the cluster. This is similar to the Jobtracker, Tasktracker, and Datanode concepts in an Hadoop configuration. Multiple Thor clusters can exist in an HPCC environment, and job queues can span multiple clusters in an environment if needed. Jobs executing on a Thor cluster in a multi-cluster environment can also read files from the distributed file system on foreign clusters if needed. The middleware layer provides additional server processes to support the execution environment including ECL Agents and ECL Servers. A client process submits an ECL job to the ECL Agent which coordinates the overall job execution on behalf of the client process. An ECL Job is compiled by the ECL server which interacts with an additional server called the ECL Repository which is a source code repository and contains shared ECL code. ECL programs are compiled into optimized C++ source code, which is subsequently compiled into executable code and distributed to the slave nodes of a Thor cluster by the Thor master node. The Thor master monitors and coordinates the processing activities of the slave nodes and communicates status information monitored by the ECL Agent processes. When the job completes, the ECL Agent and client process are notified, and the output of the process is available for viewing or subsequent processing. Output can be stored in the distributed filesystem for the cluster or returned to the client process. ECL is analogous to the Pig language which can be used in the Hadoop environment.

The distributed filesystem used in a Thor cluster is record-oriented which is different from the block format used by Hadoop clusters. Records can be fixed or variable length, and support a variety of standard (fixed record size, CSV, XML) and custom formats including nested child datasets. Record I/O is buffered in large blocks to reduce latency and improve data transfer rates to and from disk Files to be loaded to a Thor cluster are typically first transferred to a landing zone from some external location, then a process called “spraying” is used to partition the file and load it to the nodes of a Thor cluster. The initial spraying process divides the file on user-specified record boundaries and distributes the data as evenly as possible in order across the available nodes in the cluster. Files can also be “desprayed” when

needed to transfer output files to another system or can be directly copied between Thor clusters in the same environment.

Nameservices and storage of metadata about files including record format information in the Thor DFS are maintained in a special server called the Dali server (named for the developer's pet Chinchilla), which is analogous to the Namenode in HDFS. Thor users have complete control over distribution of data in a Thor cluster, and can re-distribute the data as needed in an ECL job by specific keys, fields, or combinations of fields to facilitate the locality characteristics of parallel processing. The Dali nameserver uses a dynamic datastore for filesystem metadata organized in a hierarchical structure corresponding to the scope of files in the system. The Thor DFS utilizes the local Linux filesystem for physical file storage, and file scopes are created using file directory structures of the local file system. Parts of a distributed file are named according to the node number in a cluster, such that a file in a 400-node cluster will always have 400 parts regardless of the file size. The Hadoop fixed block size can end up splitting logical records between nodes which means a node may need to read some data from another node during Map task processing. With the Thor DFS, logical record integrity is maintained, and processing I/O is completely localized to the processing node for local processing operations. In addition, if the file size in Hadoop is less than some multiple of the block size times the number of nodes in the cluster, Hadoop processing will be less evenly distributed and node to node disk accesses will be needed. If input splits assigned to Map tasks in Hadoop are not allocated in whole block sizes, additional node to node I/O will result. The ability to easily redistribute the data evenly to nodes based on processing requirements and the characteristics of the data during a Thor job can provide a significant performance improvement over the Hadoop approach. The Thor DFS also supports the concept of "superfiles" which are processed as a single logical file when accessed, but consist of multiple Thor DFS files. Each file which makes up a superfile must have the same record structure. New files can be added and old files deleted from a superfile dynamically facilitating update processes without the need to rewrite a new file. Thor clusters are fault resilient and a minimum of one replica of each file part in a Thor DFS file is stored on a different node within the cluster.

Roxie clusters consist of a configurable number of peer-coupled nodes functioning as a high-performance, high availability parallel processing query platform. ECL source code for structured queries is pre-compiled and deployed to the cluster. The Roxie distributed filesystem is a distributed indexed-based filesystem which uses a custom B+Tree structure for data storage. Indexes and data supporting queries are pre-built on Thor clusters and deployed to the Roxie DFS with portions of the index and data stored on each node. Typically the data associated with index logical keys is embedded in the index structure as a payload. Index keys can be multi-field and multivariate, and payloads can contain any type of structured or unstructured data supported by the ECL language. Queries can use as many indexes as required for a query and contain joins and other complex transformations on the data with the full expression and processing capabilities of the ECL language. For example, the LexisNexis Accurint comprehensive person report which produces many pages of output is generated by a single Roxie query.

A Roxie cluster uses the concept of Servers and Agents. Each node in a Roxie cluster runs Server and Agent processes which are configurable by a System Administrator depending on the processing requirements for the cluster. A Server process waits for a query request from a Web services interface then determines the nodes and associated Agent processes that have the data locally that is needed for a query, or portion of the query. Roxie query requests can be submitted from a client application as a SOAP call, HTTP or HTTPS protocol request from a Web application, or through a direct socket connection. Each Roxie query request is associated with a specific deployed ECL query program. Roxie queries can also be executed from programs running on Thor clusters. The Roxie Server process that receives the request owns the processing of the ECL program for the query until it is completed. The Server sends portions of the query job to the nodes in the cluster and Agent processes which have data needed for the query stored locally as needed, and waits for results. When a Server receives all the results needed from all nodes, it collates them, performs any additional processing, and then returns the result set to the client requestor.

The performance of query processing varies depending on factors such as machine speed, data complexity, number of nodes, and the nature of the query, but production results have shown throughput of a thousand results a second or more. Roxie clusters have flexible data storage options with indexes and data stored locally on the cluster, as well as being able to use indexes stored remotely in the same environment on a Thor cluster. Nameservices for Roxie clusters are also provided by the Dali server. Roxie clusters are fault-resilient and data redundancy is built-in using a peer system where replicas of data are stored on two or more nodes, all data including replicas are available to be used in the processing of queries by Agent processes. The Roxie cluster provides automatic failover in case of node failure, and the cluster will continue to perform even if one or more nodes are down. Additional redundancy can be provided by including multiple Roxie clusters in an environment.

Load balancing of query requests across Roxie clusters is typically implemented using external load balancing communications devices. Roxie clusters can be sized as needed to meet query processing throughput and response time requirements, but are typically smaller that Thor clusters.

The implementation of two types of parallel data processing platforms (Thor and Roxie) in the HPCC processing environment serving different data processing needs allows these platforms to be optimized and tuned for their specific purposes to provide the highest level of system performance possible to users. This is a distinct advantage when compared to the Hadoop MapReduce platform and architecture which must be overlayed with different systems such as HBase, Hive, and Pig which have different processing goals and requirements, and don't always map readily into the MapReduce paradigm. In addition, the LexisNexis HPCC approach incorporates the notion of a processing environment which can integrate Thor and Roxie clusters as needed to meet the complete processing needs of an organization. As a result, scalability can be defined not only in terms of the number of nodes in a cluster, but in terms of how many clusters and of what type are needed to meet system

performance goals and user requirements. This provides a distinct advantage when compared to Hadoop clusters which tend to be independent islands of processing.

LexisNexis HPCC is commercially available to implement private cloud computing environments (http://risk.lexisnexis.com/Article.aspx?id=51). In addition, LexisNexis provides hosted persistent HPCC environments to external customers. Public cloud computing PaaS utilizing the HPCC platform is planned as a future offering.

5.3.4 ECL

The ECL programming language is a key factor in the flexibility and capabilities of the HPCC processing environment. ECL was designed to be a transparent and implicitly parallel programming language for data-intensive applications. It is a high-level, declarative, non-procedural dataflow-oriented language that allows the programmer to define what the data processing result should be and the dataflows and transformations that are necessary to achieve the result. Execution is not determined by the order of the language statements, but from the sequence of dataflows and transformations represented by the language statements. It combines data representation with algorithm implementation, and is the fusion of both a query language and a parallel data processing language. ECL uses an intuitive syntax which has taken cues from other familiar languages, supports modular code organization with a high degree of reusability and extensibility, and supports high-productivity for programmers in terms of the amount of code required for typical applications compared to traditional languages like Java and C++. Similar to the benefits Sawzall provides in the Google environment, and Pig provides to Hadoop users, a 20 times increase in programmer productivity is typical significantly reducing development cycles. ECL is compiled into optimized C++ code for execution on the HPCC system platforms, and can be used for complex data processing and analysis jobs on a Thor cluster or for comprehensive query and report processing on a Roxie cluster. ECL allows inline C++ functions to be incorporated into ECL programs, and external programs in other languages can be incorporated and parallelized through a PIPE facility. External services written in C++ and other languages which generate DLLs can also be incorporated in the ECL system library, and ECL programs can access external Web services through a standard SOAPCALL interface.

The basic unit of code for ECL is called an attribute. An attribute can contain a complete executable query or program, or a shareable and reusable code fragment such as a function, record definition, dataset definition, macro, filter definition, etc. Attributes can reference other attributes which in turn can reference other attributes so that ECL code can be nested and combined as needed in a reusable manner. Attributes are stored in ECL code repository which is subdivided into modules typically associated with a project or process. Each ECL attribute added to the repository effectively extends the ECL language like adding a new word to a dictionary, and

attributes can be reused as part of multiple ECL queries and programs. With ECL a rich set of programming tools is provided including an interactive IDE similar to Visual C++, Eclipse and other code development environments.

The ECL language includes extensive capabilities for data definition, filtering, data management, and data transformation, and provides an extensive set of built-in functions to operate on records in datasets which can include user-defined transformation functions. Transform functions operate on a single record or a pair of records at a time depending on the operation. Built-in transform operations in the ECL language which process through entire datasets include PROJECT, ITERATE, ROLLUP, JOIN, COMBINE, FETCH, NORMALIZE, DENORMALIZE, and PROCESS. The transform function defined for a JOIN operation for example receives two records, one from each dataset being joined, and can perform any operations on the fields in the pair of records, and returns an output record which can be completely different from either of the input records. Example syntax for the JOIN operation from the ECL Language Reference Manual is shown in Fig. 5.10. Other important data operations included in ECL which operate across datasets and indexes include TABLE, SORT, MERGE, MERGEJOIN, DEDUP, GROUP, APPLY, ASSERT, AVE, BUILD, BUILDINDEX, CHOOSESETS, CORRELATION, COUNT, COVARIANCE, DISTRIBUTE, DISTRIBUTION, ENTH, EXISTS, GRAPH, HAVING, KEYDIFF, KEYPATCH, LIMIT, LOOP, MAX, MIN, NONEMPTY, OUTPUT, PARSE, PIPE, PRELOAD, PULL, RANGE, REGROUP, SAMPLE, SET, SOAPCALL, STEPPED, SUM, TOPN, UNGROUP, and VARIANCE.

The Thor system allows data transformation operations to be performed either locally on each node independently in the cluster, or globally across all the nodes in a cluster, which can be user-specified in the ECL language. Some operations such as PROJECT for example are inherently local operations on the part of a distributed file stored locally on a node. Others such as SORT can be performed either locally or globally if needed. This is a significant difference from the MapReduce architecture in which Map and Reduce operations are only performed locally on the input split assigned to the task. A local SORT operation in an HPCC cluster would sort the records by the specified key in the file part on the local node, resulting in the records being in sorted order on the local node, but not in full file order spanning all nodes. In contrast, a global SORT operation would result in the full distributed file being in sorted order by the specified key spanning all nodes. This requires node to node data movement during the SORT operation. Figure 5.11 shows a sample ECL program using the LOCAL mode of operation which is the equivalent of the sample PIG program for Hadoop shown in Fig. 5.7. Note the explicit programmer control over distribution of data across nodes. The colon-equals ":="operator in an ECL program is read as "is defined as". The only action in this program is the OUTPUT statement, the other statements are definitions.

An additional important capability provided in the ECL programming language is support for natural language processing (NLP) with PATTERN statements and the built-in PARSE function. The PARSE function cam accept an unambiguous grammar defined by PATTERN, TOKEN, and RULE statements with penalties

JOIN

JOIN(*leftrecset, rightrecset, joincondition* [, *transform*] [, *jointype*] [, *joinflags*])

JOIN(*setofdatasets, joincondition, transform,* **SORTED**(*fields*) [, *jointype*])

leftrecset	The left set of records to process.
rightrecset	The right set of records to process. This may be an INDEX.
joincondition	An expression specifying how to match records in the *leftrecset* and *rightrecset* or *setofdatasets* (see **Matching Logic** discussions below). In the expression, the keyword LEFT is the dataset qualifier for fields in the *leftrecset* and the keyword RIGHT is the dataset qualifier for fields in the *rightrecset*.
transform	Optional. The TRANSFORM function to call for each pair of records to process. If omitted, JOIN returns all fields from both the *leftrecset* and *rightrecset*, with the second of any duplicate named fields removed.
jointype	Optional. An inner join if omitted, else one of the listed types in the **JOIN Types** section below.
joinflags	Optional. Any option (see the **JOIN Options** section below) to specify exactly how the JOIN operation executes.
setofdatasets	The SET of recordsets to process ([idx1,idx2,idx3]), typically INDEXes, which all must have the same format.
SORTED	Specifies the sort order of records in the input *setofdatasets* and also the output sort order of the result set.
fields	A comma-delimited list of fields in the *setofdatasets*, which must be a subset of the input sort order. These *fields* must all be used in the *joincondition* as they define the order in which the fields are STEPPED.
Return:	JOIN returns a record set.

The **JOIN** function produces a result set based on the intersection of two or more datasets or indexes (as determined by the *joincondition*).

Fig. 5.10 ECL Sample syntax for JOIN operation

Go Queue: dev_eclserver_ Cluster: thor400_88_der More

```
1  // Sample ECL Code
2  layout_visits := RECORD string user; string url; string time; END;
3  visits := DATASET('~thor_data400::data::visits',layout_visits,FLAT);
4
5  layout_urlInfo := RECORD string url; string category; string pRank; END;
6  urlInfo := DATASET('~thor_data400::data::urlinfo',layout_urlInfo,FLAT);
7
8  // Distribute Visits by URL, Count vists by URL
9  layout_visitCounts := RECORD visits.url; visits_cnt := COUNT(GROUP); END;
10 visitCounts := TABLE(DISTRIBUTE(visits,HASH(url)),layout_visitCounts,url,LOCAL);
11
12 // Distribute Category by URL, Join category to URLs
13 visitCountsCat := JOIN(visitCounts,DISTRIBUTE(urlInfo,HASH(url)),LEFT.URL=RIGHT.URL,LOCAL);
14
15 // Distribute and Group by Category, Output top 10 URLs for each category
16 topUrls := TOPN(GROUP(DISTRIBUTE(visitCountsCat,HASH(category)),category,ALL,LOCAL),10, -visits_cnt);
17 OUTPUT(topUrls,,'~thor_data400::data::topurls',OVERWRITE);
```

Fig. 5.11 ECL code example

or preferences to provide deterministic path selection, a capability which can significantly reduce the difficulty of NLP applications. PATTERN statements allow matching patterns including regular expressions to be defined and used to parse information from unstructured data such as raw text. PATTERN statements can be combined to implement complex parsing operations or complete grammars from BNF definitions. The PARSE operation function across a dataset of records on a specific field within a record, this field could be an entire line in a text file for example. Using this capability of the ECL language it is possible to implement parallel processing for information extraction applications across document files including XML-based documents or Web pages. The key benefits of ECL can be summarized as follows:

- ECL incorporates transparent and implicit data parallelism regardless of the size of the computing cluster and reduces the complexity of parallel programming increasing the productivity of application developers.
- ECL enables implementation of data-intensive applications with huge volumes of data previously thought to be intractable or infeasible. ECL was specifically

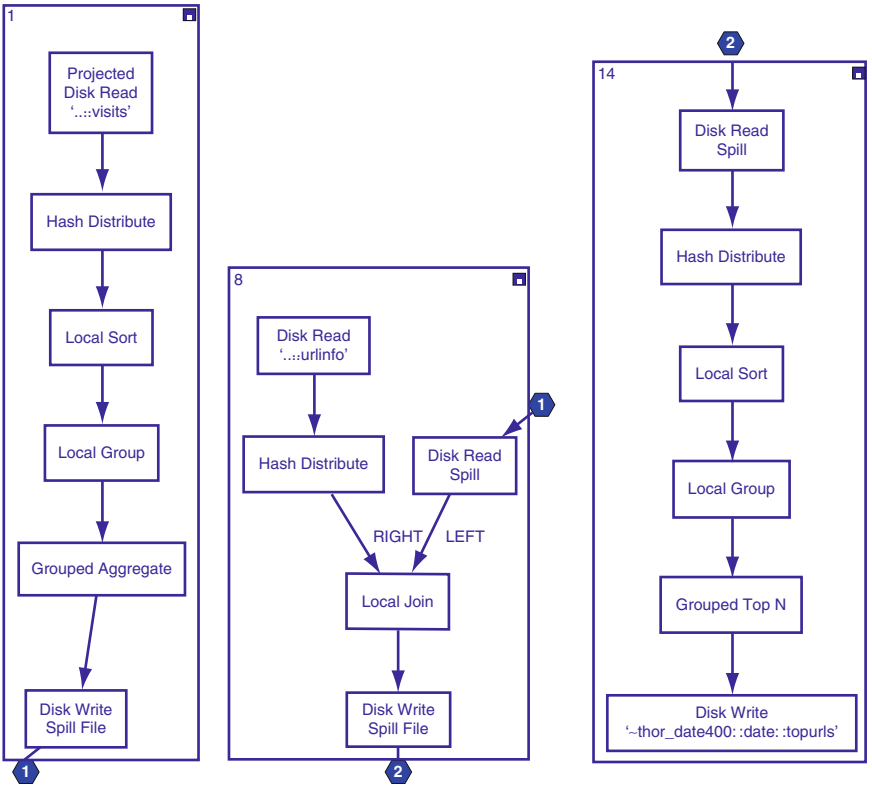


Fig. 5.12 ECL code example execution graph

designed for manipulation of data and query processing. Order of magnitude performance increases over other approaches are possible.
- ECL provides a comprehensive IDE and programming tools that provide a highly-interactive environment for rapid development and implementation of ECL applications.
- ECL is a powerful, high-level, parallel programming language ideal for implementation of ETL, Information Retrieval, Information Extraction, and other data-intensive applications.
- ECL is a mature and proven language but still evolving as new advancements in parallel processing and data-intensive computing occur.

5.4 Hadoop vs. HPCC Comparison

Hadoop and HPCC can be compared directly since it is possible for both systems to be executed on identical cluster hardware configurations. This permits head-to-head system performance benchmarking using a standard workload or set of application programs designed to test the parallel data processing capabilities of each system. A standard benchmark available for data-intensive computing platforms is the Terasort benchmark managed by an industry group led by Microsoft and HP. The Terabyte sort has evolved to be the GraySort which measures the number of terabytes per minute that can be sorted on a platform which allows clusters with any number of nodes to be utilized. However, in comparing the effectiveness and equivalent cost/performance of systems, it is useful to run benchmarks on identical system hardware configurations. A head-to-head comparison of the original Terabyte sort on a 400-node cluster will be presented here. An additional method of comparing system platforms is a feature and functionality comparison, which is a subjective evaluation based on factors determined by the evaluator. Although such a comparison contains inherent bias, it is useful in determining strengths and weaknesses of systems.

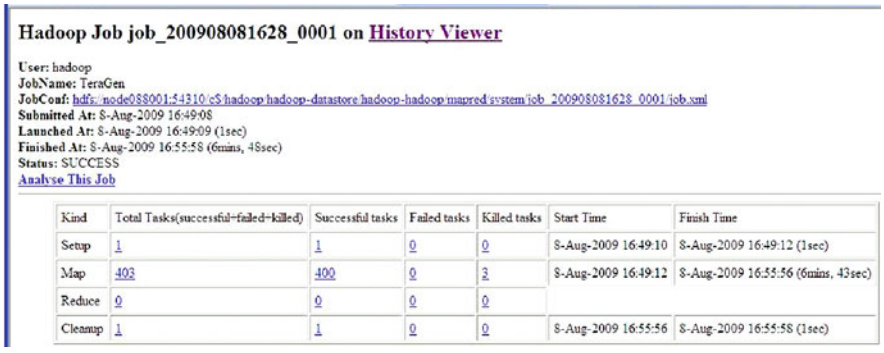
5.4.1 Terabyte Sort Benchmark

The Terabyte sort benchmark has its roots in benchmark tests sorting conducted on computer systems since the 1980s. More recently, a Web site originally sponsored by Microsoft and one of its research scientists Jim Gray has conducted formal competitions each year with the results presented at the SIGMOD (Special Interest Group for Management of Data) conference sponsored by the ACM each year (http://sortbenchmark.org). Several categories for sorting on systems exist including the Terabyte sort which was to measure how fast a file of 1 Terabyte of data formatted in 100 byte records (10,000,000 total records) could be sorted. Two categories were allowed called Daytona (a standard commercial computer system and software

with no modifications) and Indy (a custom computer system with any type of modification). No restrictions existed on the size of the system so the sorting benchmark could be conducted on as large a system as desired. The current 2009 record holder for the Daytona category is Yahoo! using a Hadoop configuration with 1460 nodes with 8 GB Ram per node, 8000 Map tasks, and 2700 Reduce tasks which sorted 1 TB in 62 seconds (O'Malley & Murthy, 2009). In 2008 using 910 nodes, Yahoo! performed the benchmark in 3 minutes 29 seconds. In 2008, LexisNexis using the HPCC architecture on only a 400-node system performed the Terabyte sort benchmark in 3 minutes 6 seconds. In 2009, LexisNexis again using only a 400-node configuration performed the Terabyte sort benchmark in 102 seconds.

However, a fair and more logical comparison of the capability of data-intensive computer system and software architectures using computing clusters would be to conduct this benchmark on the same hardware configuration. Other factors should also be evaluated such as the amount of code required to perform the benchmark which is a strong indication of programmer productivity, which in itself is a significant performance factor in the implementation of data-intensive computing applications.

On August 8, 2009 a Terabyte Sort benchmark test was conducted on a development configuration located at LexisNexis Risk Solutions offices in Boca Raton, FL in conjunction with and verified by Lawrence Livermore National Labs (LLNL). The test cluster included 400 processing nodes each with two local 300 MB SCSI disk drives, Dual Intel Xeon single core processors running at 3.00 GHz, 4 GB memory per node, all connected to a single Gigabit ethernet switch with 1.4 Terabytes/sec throughput. Hadoop Release 0.19 was deployed to the cluster and the standard Terasort benchmark written in Java included with the release was used for the benchmark. Hadoop required 6 minutes 45 seconds to create the test data, and the Terasort benchmark required a total of 25 minutes 28 seconds to complete the sorting test as shown in Fig. 5.13. The HPCC system software deployed to the same platform and using standard ECL required 2 minutes and 35 seconds to create the test data, and a total of 6 minutes and 27 seconds to complete the sorting test as shown in



Hadoop Job job_200908081628_0001 on History Viewer

User: hadoop
JobName: TeraGen
JobConf: hdfs://node088001:54310/cS/hadoop/hadoop-datastore/hadoop-hadoop/mapred/system/job_200908081628_0001/job.xml
Submitted At: 8-Aug-2009 16:49:08
Launched At: 8-Aug-2009 16:49:09 (1sec)
Finished At: 8-Aug-2009 16:55:58 (6mins, 48sec)
Status: SUCCESS
Analyse This Job

Kind	Total Tasks(successful+failed+killed)	Successful tasks	Failed tasks	Killed tasks	Start Time	Finish Time
Setup	1	1	0	0	8-Aug-2009 16:49:10	8-Aug-2009 16:49:12 (1sec)
Map	403	400	0	3	8-Aug-2009 16:49:12	8-Aug-2009 16:55:56 (6mins, 43sec)
Reduce	0	0	0	0		
Cleanup	1	1	0	0	8-Aug-2009 16:55:56	8-Aug-2009 16:55:58 (1sec)

Fig. 5.13 Hadoop terabyte sort benchmark results

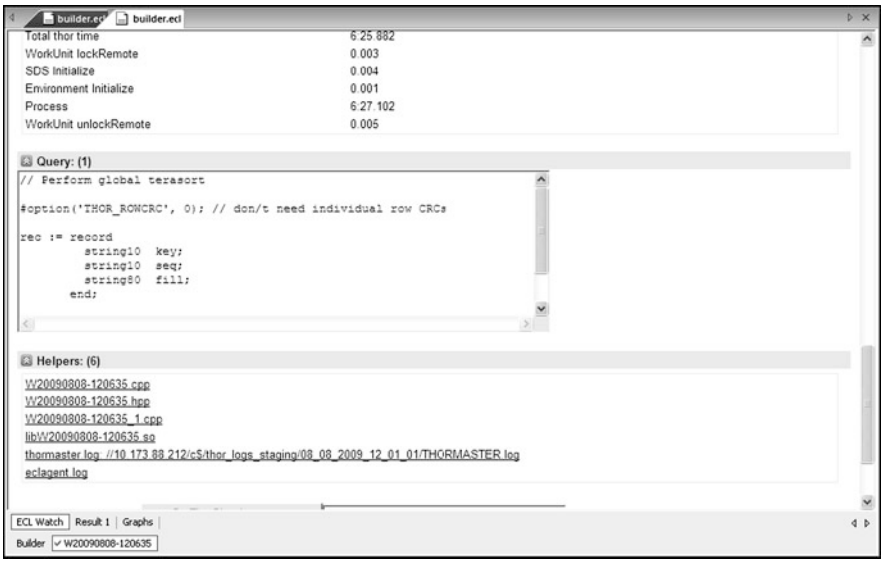


Fig. 5.14 HPCC terabyte sort benchmark results

Fig. 5.14. Thus the Hadoop implementation using Java running on the same hardware configuration took 3.95 times longer than the HPCC implementation using ECL.

The Hadoop version of the benchmark used hand-tuned Java code including custom TeraSort, TeraInputFormat and TeraOutputFormat classes with a total of 562 lines of code required for the sort. The HPCC system required only 10 lines of ECL code for the sort, a 50-times reduction in the amount of code required.

5.4.2 Pig vs. ECL

Although many Hadoop installations implement applications directly in Java, the Pig Latin language is now being used to increase programmer productivity and further simplify the programming of data-intensive applications at Yahoo! and other major users of Hadoop (Gates et al., 2009). Google also added a high-level language for similar reasons called Sawzall to its implementation of MapReduce to facilitate data analysis and data mining (Pike et al., 2004). The HPCC platform includes a high-level language discussed previously which is analogous to Pig and Sawzall called ECL. ECL is the base programming language used for applications on the HPCC platform even though it is compiled into C++ for execution. When comparing the Hadoop and HPCC platforms, it is useful to compare the features and functionality of these high-level languages.

Both Pig and ECL are intrinsically parallel, supporting transparent data-parallelism on the underlying platform. Pig and ECL are translated into programs

that automatically process input data for a process in parallel with data distributed across a cluster of nodes. Programmers of both languages do not need to know the underlying cluster size or use this to accomplish data-parallel execution of jobs. Both Pig and ECL are dataflow-oriented, but Pig is an imperative programming language and ECL is a declarative programming language. A declarative language allows programmers to focus on the data transformations required to solve an application problem and hides the complexity of the underlying platform and implementation details, reduces side effects, and facilitates compiler optimization of the code and execution plan. An imperative programming language dictates the control flow of the program which may not result in an ideal execution plan in a parallel environment. Declarative programming languages allow the programmer to specify "what" a program should accomplish, instead of "how" to accomplish it. For more information, refer to the discussions of declarative (http://en.wikipedia.org/wiki/Declarative_programming) and imperative (http://en.wikipedia.org/wiki/Imperative_programming) programming languages on Wikipedia.

The source code for both Pig and ECL is compiled or translated into another language – Pig source programs are translated into Java language MapReduce jobs for execution and ECL programs are translated into C++ source code which is then compiled into a DLL for execution. Pig programs are restricted to the MapReduce architecture and HDFS of Hadoop, but ECL has no fixed framework other than the DFS (Distributed File System) used for HPCC and therefore can be more flexible in implementation of data operations. This is evident in two key areas: (1) ECL allows operations to be either global or local, where standard MapReduce is restricted to local operations only in both the Map and Reduce phases. Global operations process the records in a dataset in order across all nodes and associated file parts in sequence maintaining the records in sorted order as opposed to only the records contained in each local node which may be important to the data processing procedure; (2) ECL has the flexibility to implement operations which can process more than one record at a time such as its ITERATE operation which uses a sliding window and passes two records at a time to an associated transform function. This allows inter-record field-by-field dependencies and decisions which are not available in Pig. For example the DISTINCT operation in Pig which is used to remove duplicates does not allow this on a subset of fields. ECL provides both DEDUP and ROLLUP operations which are usually preceded by a SORT and operate on adjacent records in a sliding window mode and any condition relating to the field contents of the left and right record of adjacent records can be used to determine if the record is removed. ROLLUP allows a custom transformation to be applied to the de-duplication process.

An important consideration of any software architecture for data is the underlying data model. Pig incorporates a very flexible nested data model which allows non-atomic data types (atomic data types include numbers and strings) such as set, map, and tuple to occur as fields of a table (Olston, Reed, Srivastava, Kumar, & Tomkins, 2008b). Tuples are sequences of fields, bags are collections of tuples, and maps are a collection of data items where each data item has a key with which it can be looked up. A data record within Pig is called a relation which is an outer bag,

the bag is a collection of tuples, each tuple is an ordered set of fields, and a field is a piece of data. Relations are referenced by a name assigned by a user. Types can be assigned by the user to each field, but if not assigned will default to a bytearray and conversions are applied depending on the context in which the field is used. The ECL data model also offers a nested data structure using child datasets. A user-specified RECORD definition defines the content of each record in a dataset which can contain fixed or variable length fields or child datasets which in turn contain fields or child datasets etc. With this format any type of data structure can be represented. ECL offers specific support for CSV and XML formats in addition to flat file formats. Each field in a record has a user-specified identifier and data type and an optional default value and optional field modifiers such as MAXLENGTH that enhance type and use checking during compilation. ECL will perform implicit casting and conversion depending on the context in which a field is used, and explicit user casting is also supported. ECL also allows in-line datasets allowing sample data to be easily defined and included in the code for testing rather than separately in a file.

The Pig environment offers several programmer tools for development, execution, and debugging of Pig Latin programs (Pig Latin is the formal name for the language, and the execution environment is called Pig, although both are commonly referred to as Pig). Pig provides command line execution of scripts and an interactive shell called Grunt that allows you to execute individual Pig commands or execute a Pig script. Pig programs can also be embedded in Java programs. Although Pig does not provide a specific IDE for developing and executing PIG programs, add-ins are available for several program editing environments including Eclipse, Vim, and Textmate to perform syntax checking and highlighting (White, 2009). PigPen is an Eclipse plug-in that provides program editing, an example data generator, and the capability to run a Pig script on a Hadoop cluster.

The HPCC platform provides an extensive set of tools for ECL development including a comprehensive IDE called QueryBuilder which allows program editing, execution, and interactive graph visualization for debugging and profiling ECL programs. The common code repository tree is displayed in QueryBuilder and tools are provided for source control, accessing and searching the repository. ECL jobs can be launched to an HPCC environment or specific cluster, and execution can be monitored directly from QueryBuilder. External tools are also provided including ECLWatch which provides complete access to current and historical workunits (jobs executed in the HPCC environment are packaged into workunits), queue management and monitoring, execution graph visualization, distributed filesystem utility functions, and system performance monitoring and analysis.

Although Pig Latin and the Pig execution environment provide a basic high-level language environment for data-intensive processing and analysis and increases the productivity of developers and users of the Hadoop MapReduce environment, ECL is a significantly more comprehensive and mature language that generates highly optimized code, offers more advanced capabilities in a robust, proven, integrated data-intensive processing architecture. Table 5.1 provides a feature to feature comparison between the Pig and ECL languages and their execution environments.

Table 5.1 Pig vs. ECL feature comparison

Language feature or capability	Pig	ECL
Language type	Data-flow oriented, imperative, parallel language for data-intensive computing. All Pig statements perform actions in sequentially ordered steps. Pig programs define a sequence of actions on the data.	Data-flow oriented, declarative, non-procedural, parallel language for data-intensive computing. Most ECL statements are definitions of the desired result which allows the execution plan to be highly optimized by the compiler. ECL actions such as OUTPUT cause execution of the dataflows to produce the result defined by the ECL program.
Compiler	Translated into a sequence of MapReduce Java programs for execution on a Hadoop Cluster. Runs as a client application.	Compiled and optimized into C++ source code which is compiled into DLL for execution on an HPCC cluster. Runs as a server application.
User-defined functions	Written in Java to perform custom processing and transformations as needed in Pig language statements. REGISTER is used to register a JAR file so that UDFs can be used.	Processing functions or TRANSFORM functions are written in ECL. ECL supports inline C++ in functions and external Services compiled into DLL libraries written in any language
Macros	Not supported	Extensive support for ECL macros to improve code reuse of common procedures. Additional template language for use in macros provides unique naming and conditional code generation capabilities.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
Data model	Nested data model with named relations to define data records. Relations can include nested combinations of bags, tuples, and fields. Atomic data types include int, long, float, double, chararray, bytearray, tuple, bag, and map. If types not specified, default to bytearray then converted during expressions evaluation depending on the context as needed.	Nested data model using child datasets. Datasets contain fields or child datasets containing fields or additional child datasets. Record definitions describe the fields in datasets and child datasets. Indexes are special datasets supporting keyed access to data. Data types can be specified for fields in record definitions and include Boolean, integer, real, decimal, string, qstring, Unicode, data, varstring, varunicode, and related operators including set of (type), type of (expression) and record of (dataset) and ENUM (enumeration). Explicit type casting is available and implicit type casting may occur during evaluation of expressions by ECL depending on the context . Type transfer between types is also supported. All datasets can have an associated filter express to include only records which meet the filter condition, in ECL a filtered physical dataset is called a recordset.
Distribution of data	Controlled by Hadoop MapReduce architecture and HDFS, no explicit programmer control provided. PARALLEL allows number of Reduce tasks to be specified. Local operations only are supported, global operations require custom Java MapReduce programs.	Explicit programmer control over distribution of data across cluster using DISTRIBUTE function. Helps avoid data skew. ECL supports both local (operations are performed on data local to node) and global (operations performed across nodes) modes.
Operators	Standard comparison operators; standard arithmetic operators and modulus division, Boolean operators AND, OR, NOT; null operators (is null, is not null); dereference operators for tuples and maps; explicit cast operator; minus and plus sign operators; matches operator.	Supports arithmetic operators including normal division, integer division, and modulus division; bitwise operators for AND, OR, and XOR; standard comparison operators; Boolean operators NOT, AND, OR; explicit cast operator; minus and plus sign operators; set and record set operators; string concatenation operator; sort descending and ascending operator; special operators IN, BETWEEN, WITHIN.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
Conditional expression evaluation	The bincond operator is provided (condition ? true_value : false_value)	ECL includes an IF statement for single expression conditional evaluation, and MAP, CASE, CHOOSE, WHICH, and REJECTED for multiple expression evaluation. The ASSERT statement can be used to test a condition across a dataset. EXISTS can be used to determine if records meeting the specified condition exist in a dataset. ISVALID determines if a field contains a valid value.
Program loops	No capability exists other than the standard relation operations across a dataset. FOREACH . . . GENERATE provides nested capability to combine specific relation operations.	In addition to built-in data transform functions, ECL provides LOOP and GRAPH statements which allow looping of dataset operations or iteration of a specified process on a dataset until a loopfilter condition is met or a loopcount is satisfied.
Indexes	Not supported directly by Pig. HBase and Hive provide indexed data capability for Hadoop MapReduce which are accessible through custom user-defined functions in Pig.	Indexes can be created on datasets to support keyed access to data to improve data processing performance and for use on the Roxie data delivery engine for query applications.
Language statement types	Grouped into relational operators, diagnostic operators, UDF (user-defined function) statements, Eval functions, and load/store functions. The Grunt shell offers additional interactive file commands.	Grouped into dataset, index and record definitions, built-in functions to define processing and dataflows and workflow management, and actions which trigger execution. Functions include transform functions such as JOIN which operate on data records, and aggregation functions such as SUM. Action statements result in execution based on specified ECL definitions describing the dataflows and results for a process.
External program calls	PIG includes the STREAM statement to send data to an external script or program. The SHIP statement can be used to ship program binaries, jar files, or data to the Hadoop cluster compute nodes. The DEFINE statement, with INPUT, OUTPUT, SHIP, and CACHE clauses allow functions and commands to be associated with STREAM to access external programs.	ECL includes PIPE option on DATASET and OUTPUT and a PIPE function to execute external 3rd-party programs in parallel on nodes across the cluster. Most programs which receive an input file and parameters can adapted to run in the HPCC environment.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
External web services access	Not supported directly by the Pig language. User-defined functions written in Java can provide this capability.	Built-in ECL function SOAPCALL for SOAP calls to access external Web Services. An entire dataset can be processed by a single SOAPCALL in an ECL program.
Data aggregation	Implemented in Pig using the GROUP, and FOREACH . . . GENERATE statements performing EVAL functions on fields. Built-in EVAL functions include AVG, CONCAT, COUNT, DIFF, ISEMPTY, MAX, MIN, SIZE, SUM, TOKENIZE.	Implemented in ECL using the TABLE statement with group by fields specified and an output record definition that includes computed fields using expressions with aggregation functions performed across the specified group. Built-in aggregation functions which work across datasets or groups include AVE, CORRELATION, COUNT, COVARIANCE, MAX, MIN, SUM, VARIANCE.
Natural language processing	The TOKENIZE statement splits a string and outputs a bag of words. Otherwise no direct language support for parsing and other natural language processing. User-defined functions are required.	Includes PATTERN, RULE, TOKEN, and DEFINE statements for defining parsing patterns, rules, and grammars. Patterns can include regular expression definitions and user-defined validation functions. The PARSE statement provides both regular expression type parsing or Tomita parsing capability and recursive grammars. Special parsing syntax is included specifically for XML data.
Scientific function support	Not supported directly by the Pig language. Requires the definition and use of a user-defined function.	ECL provides built-in functions for ABS, ACOS, ASIN, ATAN, ATAN2, COS, COSH, EXP, LN, LOG, ROUND, ROUNDUP, SIN, SINH, SQRT, TAN, TANH.
Hashing functions for dataset distribution	No explicit programmer control for dataset distribution. PARALLEL option on relational operations allows the number of Reduce tasks to be specified.	Hashing functions available for use with the DISTRIBUTE statement include HASH, HASH32 (32-bit FNV), HASH64 (64-bit FNV), HASHCRC, HASHMD5 (128-bit MD5)

Table 5.1 (continued)

Language feature or capability	Pig	ECL
Creating sample datasets	The SAMPLE operation selects a random data sample with a specified sample size.	ECL provides ENTH which selects every nth record of a dataset, SAMPLE which provides the capability to select non-overlapping samples on a specified interval, CHOOSEN which selects the first n records of a dataset and CHOOSESETS which allows multiple conditions to be specified and the number of records that meet the condition or optionally a number of records that meet none of the conditions specified. The base dataset for each of the ENTH, SAMPLE, CHOOSEN, and CHOOSETS can have a associated filter expression.
Workflow management	No language statements in Pig directly affect Workflow. The Hadoop cluster does allow Java MapReduce programs access to specific workflow information and scheduling options to manage execution.	Workflow Services in ECL include the CHECKPOINT and PERSIST statements allow the dataflow to be captured at specific points in the execution of an ECL program. If a program must be rerun because of a cluster failure, it will resume at last Checkpoint which is deleted after completion. The PERSIST files are stored permanently in the filesystem. If a job is repeated, persisted steps are only recalculated if the code has changed, or any underlying data has changed. Other workflow statements include FAILURE to trap expression evaluation failures, PRIORITY, RECOVERY, STORED, SUCCESS, WHEN for processing events, GLOBAL and INDEPENDENT.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
PIG relation operations:		
Cogroup	The COGROUP operation is similar to the JOIN operation and groups the data in two or more relations (datasets) based on common field values. COGROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples. INNER and OUTER joins are supported. Fields from each relation are specified as the join key. No support exists for conditional processing other than field equality.	In ECL, this is accomplished using the DENORMALIZE function joining to each dataset and adding all records matching the join key to a new record format with a child dataset for each child file. The DENORMALIZE function is similar to a JOIN and is used to form a combined record out of a parent and any number of children.
Cross	Creates the cross product of two or more relations (datasets).	In ECL the JOIN operation can be used to create cross products using a join condition that is always true.
Distinct	Removes duplicate tuples in a relation. All fields in the tuple must match. The tuples are sorted prior to this operation. Cannot be used on a subset of fields. A FOREACH . . . GENERATE statement must be used to generate the fields prior to a DISTINCT operation in this case.	The ECL DEDUP statement compares adjacent records to determine if a specified conditional expression is met, in which case the duplicate record is dropped and the remaining record is compared to the next record in a sliding window manner. This provides a much more flexible deduplication capability than the Pig DISTINCT operation. A SORT is required prior to a DEDUP unless using the ALL option. Conditions can use any expression and can reference values from the left and right adjacent records. DEDUP can use any subset of fields.
Dump	Displays the contents of a relation.	ECL provides an OUTPUT statement that can either write files to the filesystem or for display. Display files can be named and are stored in the Workunit associated with the job. Workunits are archived on a management server in the HPCC platform.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
Filter	Selects tuples from a relation based on a condition. Used to select the data you want or conversely to filter out remove the data you don't want.	Filter expressions can be used any time a dataset or recordset is referenced in any ECL statement with the filter expression in parenthesis following the dataset name as dataset_name(filter_expression). The ECL compiler optimizes filtering of the data during execution based on the combination of filtering expressions.
Foreach . . . Generate	Generates data transformations based on columns of data. This action can be used for projection, aggregation, and transformation, and can include other operations in the generation clause such as FILTER, DISTINCT, GROUP, etc.	Each ECL transform operation such as PROJECT, JOIN, ROLLUP, etc. include a TRANSFORM function which implicitly provides the FOREACH . . . GENERATE operation as records are processed by the TRANSFORM function. Depending on the function, the output record of the transform can include fields from the input and computed fields selectively as needed and does not have to be identical to the input record.
Group	Groups together the tuples in a single relation that have the same group key fields.	The GROUP operation in ECL fragments a dataset into a set of sets based on the break criteria which is a list of fields or expressions based on fields in the record which function as the group by keys. This allows aggregations and transform operations such as ITERATE, SORT, DEDUP, ROLLUP and others to occur within defined subsets of the data as it executes on each subset individually.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
Join	Joins two or more relations based on common field values. The JOIN operator always performs an inner join. If one relation is small and can be held in memory, the "replicated" option can be used to improve performance.	The ECL JOIN operation works on two datasets or a set of datasets. For two datasets INNER, FULL OUTER, LEFT OUTER, RIGHT OUTER, LEFT ONLY and RIGHT ONLY joins are permitted. For the set of datasets JOIN, INNER, LEFT OUTER, LEFT ONLY, and MOFN (min, max) joins are permitted. Any type of conditional expression referencing fields in the datasets to be joined can be used as a join condition. JOIN can be used in both a global and local modes also provides additional options for distribution including HASH which distributes the datasets by the specified join keys, and LOOKUP which copies one dataset if small to all nodes and is similar to the "replicated" join feature of Pig. Joins can also use keyed indexes to improve performance and self-joins (joining the same dataset to itself) is supported. Additional join-type operations provided by ECL include MERGEJOIN which joins and merges in a single operation, and smart stepping using STEPPED which provides a method of doing n-ary join/merge-join operations.
Limit	Used to limit the number of output tuples in a relation. However, there is no guarantee of which tuples will be output unless preceded by an ORDER statement.	The LIMIT function in ECL is to restrict the output of a recordset resulting from processing to a maximum number or records, or to fail the operation if the limit is exceeded. The CHOOSEN function can be use to select a specified number of records in a dataset.
Load	Loads data from the filesystem.	Since ECL is declarative, the equivalent of the Pig LOAD operation is a DATASET definition which also includes a RECORD definition. The examples shown in Figs 5.7 and Fig. 5.11 demonstrate this difference.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
Order	Sorts a relation based on one or more fields. Both ascending and descending sorts are supported. Relations will be in order for a DUMP, but if the result of an ORDER is further processed by another relation operation, there is no guarantee the results will be processed in the order specified. Relations are considered to be unordered in Pig.	The ECL SORT function sorts a dataset according to a list of expressions or key fields. The SORT can be global in which the dataset will be ordered across the nodes in a cluster, or local in which the dataset will be ordered on each node in the cluster individually. For grouped datasets, the SORT applies to each group individually. Sorting operations can be performed using a quicksort, insertionsort, or heapsort, and can be stable or unstable for duplicates.
Split	Partitions a relation into two or more relations.	Since ECL is declarative, partitions are created by simply specifying filter expressions on the base dataset. Example for dataset DS1, you could define DS2 := DS1(filter_expression _1), DS3 := DS1(filter_expression _2), etc.
Store	Stores data to the file system.	The OUTPUT function in ECL is used to write a dataset to the filesystem or to store it in the workunit for display. Output files can be compressed using LZW compression. Variations of OUTPUT support flat file, CSV, and XML formats. Output can also be written to a PIPE as the standard input to the command specified for the PIPE operation. Output can write not only the filesystem on the local cluster, but to any cluster filesystem in the HPCC processing environment.
Union	The UNION operator is used to merge the contents of two or more relations into a single relation. Order of tuples is not preserved, both input and output relations are interpreted as an unordered bag of tuples. Does not eliminate duplicate tuples.	The MERGE function returns a single dataset or index containing all the datasets or indexes specified in a list of datasets. Datasets must have the same record format. A SORTED option allows the merge to be ordered according to a field list that specifies the sort order. A DEDUP option causes only records with unique keys to be included. The REGROUP function allows multiple datasets which have been grouped using the same fields to be merged into a single dataset.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
Additional ECL transformation functions		ECL includes many additional functions providing important data transformations that are not available in Pig without implementing custom user-defined processing.
Combine	Not available	The COMBINE function combines two datasets into a single dataset on a record-by-record basis in the order in which they appear in each. Records from each are passed to the specified transform function, and the record format of the output dataset can contain selected fields from both input datasets and additional fields as needed.
Fetch	Not available	The FETCH function processes through all the records in an index dataset in the order specified by the index fetching the corresponding record from the base dataset and passing it through a specified transform function to create a new dataset.
Iterate	Not available	The ITERATE function processes through all records in a dataset one pair of records at a time using a sliding window method performing the transform record on each pair in turn. If the dataset is grouped, the ITERATE processes each group individually. The ITERATE function is useful in propagating information and calculating new information such as running totals since it allows inter-record dependencies to be considered.
Normalize	Use of FOREACH . . . GENERATE is required	The NORMALIZE function normalizes child records out of a dataset into a separate dataset. The associated transform and output record format does not have to be the same as the input.

Table 5.1 (continued)

Language feature or capability	Pig	ECL
Process	Not available	The PROCESS function is similar to ITERATE and processes through all records in a dataset one pair of records at a time (left record, right record) using a sliding window method performing the associated transform function on each pair of records in turn. A second transform function is also specified that constructs the right record for the next comparison.
Project	Use of FOREACH . . . GENERATE is required	The PROJECT processes through all the records in a dataset performing the specified transform on each record in turn.
Rollup	Not available	The ROLLUP function is similar to the DEDUP function but includes a specified transform function to process each pair of duplicate records. This allows you to retrieve and use valuable information from the duplicate record before it is thrown away. Depending on how the ROLLUP is defined, either the left or right record passed to the transform can be retained, or any mixture of data from both.
Diagnostic operators	Pig includes diagnostic operators to aid in the visualization of data structures. The DESCRIBE operator returns the schema of a relation. The EXPLAIN operator allows you to review the logical, physical, and MapReduce execution plans that are used to compute an operation in a Pig script. The ILLUSTRATE operator displays a step-by-step execution of a sequence of statements allow you to see how data is transformed through a sequence of Pig Latin statements essentially dumping the output of each statement in the script.	The DISTRIBUTION action produces a crosstab report in XML format indicating how many records there are in a dataset for each value in each field in the dataset to aid in the analysis of data distribution in order to avoid skews. The QueryBuilder and ECLWatch program development environment tools provide a complete visualization tool for analyzing, debugging, and profiling execution of ECL jobs. During the execution of a job, the dataflows expressed by ECL can be viewed as a directed acyclic graph (DAG) which shows the execution plan, dataflows as they occur, and the results of each processing step. Users can double click on the graph to drill down for additional information. An example of the graph corresponding to the ECL code shown in Fig. 5.11 is shown in Fig. 5.12.

5.4.3 Architecture Comparison

Hadoop MapReduce and the LexisNexis HPCC platform are both scalable architectures directed towards data-intensive computing solutions. Each of these system platforms has strengths and weaknesses and their overall effectiveness for any application problem or domain is subjective in nature and can only be determined through careful evaluation of application requirements versus the capabilities of the solution. Hadoop is an open source platform which increases its flexibility and adaptability to many problem domains since new capabilities can be readily added by users adopting this technology. However, as with other open source platforms, reliability and support can become issues when many different users are contributing new code and changes to the system. Hadoop has found favor with many large Web-oriented companies including Yahoo!, Facebook, and others where data-intensive computing capabilities are critical to the success of their business. Amazon has implemented new cloud computing services using Hadoop as part of its EC2 called Amazon Elastic MapReduce. A company called Cloudera was recently formed to provide training, support and consulting services to the Hadoop user community and to provide packaged and tested releases which can be used in the Amazon environment. Although many different application tools have been built on top of the Hadoop platform like Pig, HBase, Hive, etc., these tools tend not to be well-integrated offering different command shells, languages, and operating characteristics that make it more difficult to combine capabilities in an effective manner.

However, Hadoop offers many advantages to potential users of open source software including readily available online software distributions and documentation, easy installation, flexible configurations based on commodity hardware, an execution environment based on a proven MapReduce computing paradigm, ability to schedule jobs using a configurable number of Map and Reduce tasks, availability of add-on capabilities such as Pig, HBase, and Hive to extend the capabilities of the base platform and improve programmer productivity, and a rapidly expanding user community committed to open source. This has resulted in dramatic growth and acceptance of the Hadoop platform and its implementation to support data-intensive computing applications.

The LexisNexis HPCC platform is an integrated set of systems, software, and other architectural components designed to provide data-intensive computing capabilities from raw data processing and ETL applications, to high-performance query processing and data mining. The ECL language was specifically implemented to provide a high-level dataflow parallel processing language that is consistent across all system components and has extensive capabilities developed and optimized over a period of almost 10 years. The LexisNexis HPCC is a mature, reliable, well-proven, commercially supported system platform used in government installations, research labs, and commercial enterprises. The comparison of the Pig Latin language and execution system available on the Hadoop MapReduce platform to the ECL language used on the HPCC platform presented here reveals that ECL provides significantly more advanced capabilities and functionality without the need

for extensive user-defined functions written in another language or resorting to a native MapReduce application coded in Java.

The following comparison of overall features provided by the Hadoop and HPCC system architectures reveals that the HPCC architecture offers a higher level of integration of system components, an execution environment not limited by a specific computing paradigm such as MapReduce, flexible configurations and optimized processing environments which can provide data-intensive applications from data analysis to data warehousing and high-performance online query processing, and high programmer productivity utilizing the ECL programming language and tools. Table 5.2 provides a summary comparison of the key features of the hardware and software architectures of both system platforms based on the analysis of each architecture presented in this chapter.

5.5 Conclusions

As a result of the continuing information explosion, many organizations are drowning in data and the data gap or inability to process this information and use it effectively is increasing at an alarming rate. Data-intensive computing represents a new computing paradigm which can address the data gap and allow government and commercial organizations and research environments to process massive amounts of data and implement applications previously thought to be impractical or infeasible. Some organizations with foresight recognized early that new parallel-processing architectures were needed including Google who initially developed the MapReduce architecture and LexisNexis who developed the HPCC architecture. More recently the Hadoop platform has emerged as an open source alternative for the MapReduce approach. Hadoop has gained momentum quickly, and additional add-on capabilities to enhance the platform have been developed including a dataflow programming language and execution environment called Pig. These architectures, their relative strengths and weaknesses, and their applicability to cloud computing are described in this chapter, and a direct comparison of the Pig language of Hadoop to the ECL language used with the LexisNexis HPCC platform was presented. Availability of a high-level parallel dataflow-oriented programming language has proven to be a critical success factor in data-intensive computing.

The suitability of a processing platform and architecture for an organization and its application requirements can only be determined after careful evaluation of available alternatives. Many organizations have embraced open source platforms while others prefer a commercially developed and supported platform by an established industry leader. The Hadoop MapReduce platform is now being used successfully at many so-called Web companies whose data encompasses massive amounts of Web information as its data source. The LexisNexis HPCC platform is at the heart of a premier information services provider and industry leader, and has been adopted by government agencies, commercial organizations, and research laboratories because of its high-performance cost-effective implementation. Existing HPCC applications

Table 5.2 Hadoop vs. HPCC feature comparison

Architecture characteristic	Hadoop	HPCC
Hardware type	Processing clusters using commodity off-the-shelf (COTS) hardware. Typically rack-mounted blade servers with Intel or AMD processors, local memory and disk connected to a high-speed communications switch (usually Gigabit Ethernet connections) or hierarchy of communications switches depending on the total size of the cluster. Clusters are usually homogenous (all processors are configured identically), but this is not a requirement.	Same
Operating system	Unix/Linux and Windows (requires the installation of Cygwin)	Linux/Windows.
System configurations	Hadoop system software implements cluster with MapReduce processing paradigm. The cluster also functions as a distributed file system running HDFS. Other capabilities are layered on top of the Hadoop MapReduce and HDFS system software including HBase, Hive, etc.	HPCC clusters can be implemented in two configurations: Data Refinery (Thor) is analogous to the Hadoop MapReduce Cluster; Data Delivery Engine (Roxie) provides separate high-performance online query processing and data warehouse capabilities. Both configurations also function as distributed file systems but are implemented differently based on the intended use to improve performance. HPCC environments typically consist of multiple clusters of both configuration types. Although filesystems on each cluster are independent, a cluster can access files the filesystem on any other cluster in the same environment.
Licensing cost	None. Hadoop is an open source platform and can be freely downloaded and used.	License fees currently depend on size and type of system configurations. Does not preclude a future open source offering.

Table 5.2 (continued)

Architecture characteristic	Hadoop	HPCC
Core software	Core software includes the operating system and Hadoop MapReduce cluster and HDFS software Each slave node includes a Tasktracker service and Datanode service. A master node includes a Jobtracker service which can be configured as a separate hardware node or run on one of the slave hardware nodes. Likewise, for HDFS, a master Namenode service is also required to provide name services and can be run on one of the slave nodes or a separate node	For a Thor configuration, core software includes the operating system and various services installed on each node of the cluster to provide job execution and distributed file system access. A separate server called the Dali server provides filesystem name services and manages Workunits for jobs in the HPCC environment. A Thor cluster is also configured with a master node and multiple slave nodes. A Roxie cluster is a peer-coupled cluster where each node runs Server and Agent tasks for query execution and key and file processing. The filesystem on the Roxie cluster uses a distributed B+Tree to store index and data and provides keyed access to the data. Additional middleware components are required for operation of Thor and Roxie clusters.
Middleware components	None. Client software can submit jobs directly to the Jobtracker on the master node of the cluster. A Hadoop Workflow Scheduler (HWS) which will run as a server is currently under development to manage jobs which require multiple MapReduce sequences.	Middleware components include an ECL code repository implemented on a MySQL server, and ECL server for compiling of ECL programs and queries, an ECLAgent acting on behalf of a client program to manage the execution of a job on a Thor cluster, an ESPServer (Enterpise Services Platform) providing authentication, logging, security, and other services for the job execution and Web services environment, and the Dali server which functions as the system data store for job workunit information and provides naming services for the distributed filesystems. Flexibility exists for running the middleware components on one to several nodes. Multiple copies of these servers can provide redundancy and improve performance.

Table 5.2 (continued)

Architecture characteristic	Hadoop	HPCC
System tools	The dfsadmin tool provides information about the state of the filesystem; fsck is a utility for checking the health of files in HDFS; datanode block scanner periodically verifies all the blocks stored on a datanode; balancer re-distributes blocks from over-utilized datanodes to underutilized datanodes as needed. The MapReduce Web UI includes the JobTracker page which displays information about running and completed jobs, drilling down on a specific job displays detailed information about the job. There is also a Tasks page that displays info about Map and Reduce tasks.	HPCC includes a suite of client and operations tools for managing, maintaining, and monitoring HPCC configurations and environments. These include QueryBuilder the program development environment, an Attribute Migration Tool, Distributed File Utility (DFU), an Environment Configuration Utility, Roxie Configuration Utility. Command line versions are also available. ECLWatch is a Web based utility program for monitoring the HPCC environment and includes queue management, distributed file system management, job monitoring, and system performance monitoring tools. Additional tools are provided through Web services interfaces.
Ease of deployment	Assisted by online tools provided by Cloudera utilizing Wizards. Requires a manual RPM deployment.	Environment configuration tool. A Genesis server provides a central repository to distribute OS level settings, services, and binaries to all net-booted nodes in a configuration
Distributed file system	Block-oriented, uses large 64 MB or 128 MB blocks in most installations. Blocks are stored as independent units/local files in the local Unix/Linux filesystem for the node. Metadata information for blocks is stored in a separate file for each block. Master/Slave architecture with a single Namenode to provide name services and block mapping and multiple Datanodes. Files are divided into blocks and spread across nodes in the cluster. Multiple local files (1 containing the block, 1 containing metadata) for each logical block stored on a node are required to represent a distributed file.	The Thor DFS is record-oriented, uses local Linux filesystem to store file parts. Files are initially loaded (Sprayed) across nodes and each node has a single file part which can be empty for each distributed file. Files are divided on even record/document boundaries specified by the user. Master/Slave architecture with name services and file mapping information stored on a separate server. Only one local file per node required to represent a distributed file. Read/write access is supported between clusters configured in the same environment. Utilizing special adaptors allows files from external databases such as MySQL to be accessed, allowing transactional data to be integrated with DFS data and incorporated into batch jobs. The Roxie DFS utilizes distributed B+Tree index files containing key information and data stored in local files on each node.

Table 5.2 (continued)

Architecture characteristic	Hadoop	HPCC
Fault resilience	HDFS stores multiple replicas (user-specified) of data blocks on other nodes (configurable) to protect against disk and node failure with automatic recovery. MapReduce architecture includes speculative execution, when a slow or failed Map task is detected, additional Map tasks are started to recover from node failures	The DFS for Thor and Roxie stores replicas of file parts on other nodes (configurable) to protect against disk and node failure. Thor system offers either automatic or manual node swap and warm start following a node failure, jobs are restarted from last checkpoint or persist. Replicas are automatically used while copying data to the new node. Roxie system continues running following a node failure with a reduced number of nodes.
Job execution environment	Uses MapReduce processing paradigm with input data in key-value pairs. Master/Slave processing architecture. A Jobtracker runs on the master node, and a TaskTracker runs on each of the slave nodes. Each TaskTracker can be configured with a fixed number of slots for Map and Reduce tasks depending on available memory resources. Map tasks are assigned to input splits of the input file, usually 1 per block. The number of Reduce tasks is assigned by the user. Map processing is local to assigned node. A shuffle and sort operation is done following Map phase to distribute and sort key-value pairs to Reduce tasks based on key regions so that pairs with identical keys are processed by same Reduce tasks. Multiple MapReduce processing steps are typically required for most procedures and must be sequenced and chained separately by the user or language such as Pig. Job schedulers include FIFO (default), Fair, and Capacity depending on job sharing requirements, as well as Hadoop on Demand (HOD) for creating virtual clusters within a physical cluster.	Thor utilizes a Master/Slave processing architecture. Processing steps defined in an ECL job can specify local (data processed separately on each node) or global (data is processed across all nodes) operation. Multiple processing steps for a procedure are executed automatically as part of a single job based on an optimized execution graph for a compiled ECL dataflow program. A single Thor cluster can be configured to run multiple jobs concurrently reducing latency if adequate CPU and memory resources are available on each node. Middleware components including an ECLAgent, ECLServer, and DaliServer provide the client interface and manage execution of the job which is packaged as a Workunit. Roxie utilizes a multiple Server/Agent architecture to process ECL programs accessed by queries using Server tasks acting as a manager for each query and multiple Agent tasks as needed to retrieve and process data for the query.

Table 5.2 (continued)

Architecture characteristic	Hadoop	HPCC
Programming languages	Hadoop MapReduce jobs are usually written in Java. Other languages are supported through a streaming or pipe interface. Other processing environments execute on top of Hadoop MapReduce such as HBase and Hive which have their own language interface. The Pig Latin language and Pig execution environment provides a high-level dataflow language which is then mapped into multiple Java MapReduce jobs.	ECL is the primary programming language for the HPCC environment. ECL is compiled into optimized C++ which is then compiled into DLLs for execution on the Thor and Roxie platforms. ECL can include inline C++ code encapsulated in functions. External services can be written in any language and compiled into shared libraries of functions callable from ECL. A Pipe interface allows execution of external programs written in any language to be incorporated into jobs.
Integrated program development environment	Hadoop MapReduce utilizes the Java programming language and there are several excellent program development environments for Java including Netbeans and Eclipse which offer plug-ins for access to Hadoop clusters. The Pig environment does not have its own IDE, but instead uses Eclipse and other editing environments for syntax checking. A PigPen add-in for Eclipse provides access to Hadoop Clusters to run Pig programs and additional development capabilities.	The HPPC platform is provided with QueryBuilder, a comprehensive IDE specifically for the ECL language. QueryBuilder provides access to shared source code repositories and provides a complete development and testing environment for ECL dataflow programs. Access to the ECLWatch tool is built-in, allowing developers to watch job graphs as they are executing. Access to current and historical job Workunits allows developers to easily compare results from one job to the next during development cycles.
Database capabilities	The basic Hadoop MapReduce system does not provide any keyed access indexed database capabilities. An add-on system for Hadoop called HBase provides a column-oriented database capability with keyed access. A custom script language and Java interface is provided. Access to HBase is not directly supported by the Pig environment and requires user-defined functions or separate MapReduce procedures.	The HPCC platform includes the capability to build multi-key, multivariate indexes on DFS files. These indexes can be used to improve performance and provide keyed access for batch jobs on a Thor system, or be used to support development of queries deployed to Roxie systems. Keyed access to data is supported directly in ECL

Table 5.2 (continued)

Architecture characteristic	Hadoop	HPCC
Online query and data warehouse capabilities	The basic Hadoop MapReduce system does not provide any data warehouse capabilities. An add-on system for Hadoop called Hive provides data warehouse capabilities and allows HDFS data to be loaded into tables and accessed with an SQL-like language. Access to Hive is not directly supported by the Pig environment and requires user-defined functions or separate MapReduce procedures.	The Roxie system configuration in the HPCC platform is specifically designed to provide data warehouse capabilities for structured queries and data analysis applications. Roxie is a high-performance platform capable of supporting thousands of users and providing sub-second response time depending on the application.
Scalability	1 to thousands of nodes. Yahoo! has production clusters as large as 4000 nodes.	1 to several thousand nodes. In practice, HPCC configurations require significantly fewer nodes to provide the same processing performance as Hadoop. Sizing of clusters may depend however on the overall storage requirements for the distributed file system.
Performance	Currently the only available standard performance benchmarks are the sort benchmarks sponsored by http://sortbenchmark.org. Yahoo! has demonstrated sorting 1 TB on 1460 nodes in 62 seconds, 100 TB using 3452 nodes in 173 minutes, and 1 PB using 3658 nodes in 975 minutes.	The HPPC platform has demonstrated sorting 1 TB on a high-performance 400-node system in 102 seconds. In a recent head-to-head benchmark versus Hadoop on a another 400-node system conducted with LLNL, HPPC performance was 6 minutes 27 seconds and Hadoop performance was 25 minutes 28 seconds. This result on the same hardware configuration showed that HPCC was 3.95 times faster than Hadoop for this benchmark.
Training	Hadoop training is offered through Cloudera. Both beginning and advanced classes are provided. The advanced class includes Hadoop add-ons including HBase and Pig. Cloudera also provides a VMWare based learning environment which can be used on a standard laptop or PC. Online tutorials are also available.	Basic and advanced training classes on ECL programming are offered monthly in several locations or on customer premises. A system administration class is offered and scheduled as needed. A CD with a complete HPCC and ECL learning environment which can be used on a single PC or laptop is also available.

include raw data processing, ETL, and linking of enormous amounts of data to support online information services such as LexisNexis and industry-leading information search applications such as Accurint; entity extraction and entity resolution of unstructured and semi-structured data such as Web documents to support information extraction; statistical analysis of Web logs for security applications such as intrusion detection; online analytical processing to support business intelligence systems (BIS); and data analysis of massive datasets in educational and research environments and by state and federal government agencies.

There are many tradeoffs in making the right decision in choosing a new computer systems architecture, and often the best approach is to conduct a specific benchmark test with a customer application to determine the overall system effectiveness and performance. The relative cost-performance characteristics of the system in additional to suitability, flexibility, scalability, footprint, and power consumption factors which impact the total cost of ownership (TCO) must be considered. Cloud computing alternatives which reduce or eliminate up-front infrastructure investment should also be considered if internal resources are limited.

A comparison of the Hadoop MapReduce architecture to the HPCC architecture in this chapter reveals many similarities between the platforms including the use of a high-level dataflow-oriented programming language to implement transparent data-parallel processing. Both platforms are adaptable to cloud computing to provide platform as a service (PaaS). A key advantage to using the Hadoop architecture is its availability in a public cloud computing service offering. However, private cloud computing which utilizes persistent configurations with dedicated infrastructure instead of virtualized servers shared with other users common in public cloud computing can have a significant performance advantage for data-intensive computing applications. Some additional advantages of choosing the LexisNexis HPCC platform which can be utilized in private cloud computing include: (1) an architecture which implements a highly integrated system environment with capabilities from raw data processing to high-performance queries and data analysis using a common language; (2) an architecture which provides equivalent performance at a much lower system cost based on the number of processing nodes required as demonstrated with the Terabyte Sort benchmark where the HPCC platform was almost 4 times faster than Hadoop running on the same hardware resulting in significantly lower total cost of ownership (TCO); (3) an architecture which has been proven to be stable and reliable on high-performance data processing production applications for varied organizations over a 10-year period; (4) an architecture that uses a dataflow programming language (ECL) with extensive built-in capabilities for data-parallel processing which allows complex operations without the need for extensive user-defined functions and automatically optimizes execution graphs with hundreds of processing steps into single efficient workunits; (5) an architecture with a high-level of fault resilience and language capabilities which reduce the need for re-processing in case of system failures; and (6) an architecture which is available from and supported by a well-known leader in information services and risk solutions (LexisNexis) who is part of one of the world's largest publishers of information ReedElsevier.

References

Abbas, A. (2004). *Grid computing: A practical guide to technology and applications*. Hingham, MA: Charles River Media.

Agichtein, E. (2005). Scaling information extraction to large document collections. *IEEE Data Engineering Bulletin*, 28, 3–10.

Agichtein, E., & Ganti, V. (2004). Mining reference tables for automatic text segmentation. *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, *Seattle*, *WA*, 20–29.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., et al. (2009). *Above the clouds: A Berkeley view of cloud computing* (University of California at Berkely, Tech. Rep. UCB/EECS-2009-28).

Berman, F. (2008). Got data? A guide to data preservation in the information age. *Communications of the ACM, 51*(12), 50–56.

Borthakur, D. (2008). *Hadoop distributed file system*. Available from: http://www.opendocs.net/apache/hadoop/HDFSDescription.pdf.

Bryant, R. E. (2008). *Data intensive scalable computing.* Retrieved January 5, 2010, from: http://www.cs.cmu.edu/~bryant/presentations/DISC-concept.ppt.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., & Brandic, I. (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems, 25*(6), 599–616.

Cerf, V. G. (2007). An information avalanche. *IEEE Computer, 40*(1), 104–105.

Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., et al. (2008). SCOPE: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment, New York, NY*.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., et al. (2006). Bigtable: A distributed storage system for structured data. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06), Seattle, WA*.

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI), Boston, MA*.

Gantz, J. F., Reinsel, D., Chute, C., Schlichting, W., McArthur, J., Minton, S., et al. (2007). *The expanding digital universe.* IDC, White Paper.

Gates, A. F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S. M., Olston, C., et al. (2009). Building a high-level dataflow system on top of map-reduce: The pig experience. *Proceedings of the 35th International Conference on Very Large Databases (VLDB 2009), Lyon, France*.

Gokhale, M., Cohen, J., Yoo, A., & Miller, W. M. (2008). Hardware technologies for high-performance data-intensive computing. *IEEE Computer, 41*(4), 60–68.

Gorton, I., Greenfield, P., Szalay, A., & Williams, R. (2008). Data-intensive computing in the 21st century. *IEEE Computer, 41*(4), 30–32.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The google file system. *Proceedings of the 19th ACM Symposium on Operating Systems Principles, New York, NY*.

Gray, J. (2008). Distributed computing economics. *ACM Queue, 6*(3), 63–68.

Grossman, R. L. (2009). The case for cloud computing. *IT Professional,11*(2), 23–27.

Grossman, R., & Gu, Y. (2008). Data mining using high performance data clouds: Experimental studies using sector and sphere. *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY*.

Grossman, R. L., & Gu, Y. (2009). On the varieties of clouds for data intensive computing. Available from: http://sites.computer.org/debull/A09mar/grossman.pdf, 2009.

Grossman, R. L., Gu, Y., Sabala, M., & Zhang, W. (2009). Compute and storage clouds using wide area high performance networks. *Future Generation Computer Systems, 25*(2), 179–183.

Gu, Y., & Grossman, R. L. (2009). Lessons learned from a year's worth of benchmarks of large data clouds. *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, Portland, OR*.

Hayes, B. (2008). Cloud computing. *Communications of the ACM, 51*(7), 9–11.

Johnston,W. E. (1998). High-speed, wide area, data intensive computing: A ten year retrospective. *Proceedings of the 7th IEEE International Symposium on High-Performance Distributed Computing. Chicago, Illinois*, 280.

Kouzes, R. T., Anderson, G. A., Elbert, S. T., Gorton, I., & Gracio, D. K. (2009). The changing paradigm of data-intensive computing. *Computer, 42*(1), 26–34.

Lenk, A., Klems, M., Nimis, J., Tai, S., & Sandholm, T. (2009). What's inside the cloud? An architectural map of the cloud landscape. *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing. Vancouver, Canada*, 23–31.

Levitt, N. (2009). Is cloud computing really ready for prime time? *Computer, 42*(1), 15–20.

Liu, H., & Orban, D. (2008). GridBatch: Cloud computing for large-scale data-intensive batch applications. *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid, Cardiff*.

Llor, X., Acs, B., Auvil, L. S., Capitanu, B., Welge, M. E., & Goldberg, D. E. (2008). Meandre: Semantic-driven data-intensive flows in the clouds. *Proceedings of the 4th IEEE International Conference on eScience, Nottingham*.

Lyman, P., & Varian, H. R. (2003). *How much information?* (School of Information Management and Systems, University of California at Berkeley, Research Rep.).

Mell, P., & Grance, T. (2009). *The NIST definition of cloud computing.* Retrieved January 5, 2010, from: http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc.

Napper, J., & Bientinesi, P. (2009). Can cloud computing reach the Top500?. *Conference On Computing Frontiers. Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop, Ischia, Italy*.

Nicosia, M. (2009). *Hadoop cluster management*. Retrieved January 5, 2010, from: http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/Hadoop-USENIX09.pdf.

Nyland, L. S., Prins, J. F., Goldberg, A., & Mills, P. H. (2000). A design methodology for data-parallel applications. *IEEE Transactions on Software Engineering, 26*(4), 293–314.

NSF. (2009). *Data-intensive computing*. Retrieved January 5, 2010, from: http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503324&org=IIS.

O'Malley, O. (2008). *Introduction to Hadoop*. Available from: http://wiki.apache.org/hadoop/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf.

O'Malley, O., & Murthy, A. C. (2009). *Winning a 60 second dash with a yellow elephant*. Retrieved January 5, 2010, from: http://sortbenchmark.org/Yahoo2009.pdf.

Olston, C. (2009). *Pig overview presentation – Hadoop summit*. Retrieved January 5, 2010, from: http://infolab.stanford.edu/~olston/pig.pdf.

Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008a). *Pig Latin: A not-so-foreign language for data processing (Presentation at SIGMOD 2008)*. Retrieved January 5, 2010, from: http://i.stanford.edu/~usriv/talks/sigmod08-pig-latin.ppt#283,18,User-Code as a First-Class Citizen.

Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008b). Pig Latin: A not-so_foreign language for data processing. *Proceedings of the 28th ACM SIGMOD/PODS International Conference on Management of Data/Principles of Database Systems, Vancouver, BC*.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., Dewitt, D. J., Madden, S., et al. (2009). A comparison of approaches to large-scale data analysis. *Proceedings of the 35th SIGMOD International Conference on Management of Data, New York, NY*.

PNNL. (2008). *Data intensive computing*. Retrieved January 5, 2010, from: http://www.cs.cmu.edu/~bryant/presentations/DISC-concept.ppt.

Pike, R., Dorward, S., Griesemer, R., & Quinlan, S. (2004). Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal, 13*(4), 227–298.

Ravichandran, D., Pantel, P., & Hovy, E. (2004). The terascale challenge. *Proceedings of the KDD Workshop on Mining for and from the Semantic Web, Boston, MA*.

Rencuzogullari, U., & Dwarkadas, S. (2001). Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, San Diego, CA*, 72–81.

Reese, G. (2009). *Cloud application architectures.* Sebastopol, CA: O'Reilly.

Skillicorn, D. B., & Talia, D. (1998). Models and languages for parallel computation. *ACM Computing Surveys, 30*(2), 123–169.

Vaquero, L. M., Rodero-Merino, L., Caceres, J., & Lindner, M. (2009). A break in the clouds: Towards a cloud definition. *SIGCOMM Computer Communication Review, 39*(1), 50–55.

Velte, A. T., Velte, T. J., & Elsenpeter, R. (2009). *Cloud computing: A practical approach*. New York, NY: McGraw Hill.

Venner, J. (2009). *Pro Hadoop*. New York, NY: Apress.

Viega, J. (2009). Cloud computing and the common man. *Computer, 42*(8), 106–108.

Weiss, A. (2007). Computing in the clouds. *netWorker, 11*(4), 16–25.

White, T. (2008). Understanding map reduce with Hadoop. Available from: http://wiki.apache.org/hadoop/HadoopPresentations.

White, T. (2009). Hadoop: The definitive guide. Sebastopol, CA: O'Reilly Media.

Yu, Y., Gunda, P. K., & Isard, M. (2009). Distributed aggregation for data-parallel computing: Interfaces and implementations. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT*.