Chapter 6
The HPCC/ECL Platform for Big Data

Anthony M. Middleton, David Alan Bayliss, Gavin Halliday, Arjuna Chala and Borko Furht

Introduction

As a result of the continuing information explosion, many organizations are experiencing what is now called the "Big Data" problem. This results in the inability of organizations to effectively use massive amounts of their data in datasets which have grown to big to process in a timely manner. Data-intensive computing represents a new computing paradigm [1] which can address the big data problem using high-performance architectures supporting scalable parallel processing to allow government, commercial organizations, and research environments to process massive amounts of data and implement new applications previously thought to be impractical or infeasible.

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 are:

(1) a scalable, integrated computer systems hardware and software architecture designed for parallel processing of data-intensive computing applications, and
(2) a new programming paradigm in the form of a high-level declarative data-centric programming language designed specifically for big data processing.

This chapter explores the challenges of data-intensive computing from a programming perspective, and describes the ECL programming language and the open source High-Performance Cluster Computing (HPCC) architecture designed for

© Springer International Publishing Switzerland 2016

B. Furht and F. Villanustre, *Big Data Technologies and Applications*,
DOI 10.1007/978-3-319-44550-2_6

data-intensive exascale computing applications. ECL is also compared to Pig Latin, a high-level language developed for the Hadoop MapReduce architecture.

Data-Intensive Computing Applications

High-Performance Computing (HPC) is used to describe computing environments which utilize supercomputers and computer clusters to address complex computational requirements or applications with significant processing time requirements or which require processing of significant amounts of data. Computing approaches can be generally classified as either *compute-intensive*, or *data-intensive* [2–4]. HPC has generally been associated with scientific research and compute-intensive types of problems, but more and more HPC technology is appropriate for both compute-intensive and data-intensive applications. HPC platforms utilize a high-degree of internal parallelism and tend to use specialized multi-processors with custom memory architectures which have been highly-optimized for numerical calculations [5]. Supercomputers also require special parallel programming techniques to take advantage of its performance potential.

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. HPC approaches to 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 [6].

Data-intensive is used to describe applications that are I/O bound or with a need to process large volumes of data [2, 3, 7]. Such applications devote most of their processing time to I/O and movement of data. HPC approaches to data-intensive applications typically use parallel system architectures and 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 [8]. The greater the aggregate distribution of the data, the more benefit there is in parallel processing of the data. Gorton et al. [2] 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. [2] 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.

Data-Parallelism

According to Agichtein [9], parallelization is considered to be an attractive alternative for processing extremely large collections of data such as the billions of documents on the Web [10]. Nyland et al. [8] 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. [8], 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 [8, 11]. Nyland et al. [8] 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 [10].

The "Big Data" Problem

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" [12]. 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 [13]. 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 [14]. 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 [15]. These challenges are now simple described in the literature as the "Big Data" problem. In the next section, we will enumerate some of the characteristics of data-intensive computing systems which can address the problems associated with processing big data.

Data-Intensive Computing Platforms

The National Science Foundation believes that data-intensive computing requires a "fundamentally different set of principles" than current computing approaches [16]. 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" [1, 17]. 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 ECL programming language and HPCC architecture developed by LexisNexis represents such an advance in capabilities.

Cluster Configurations

Current data-intensive computing platforms use a "divide and conquer" parallel processing approach combining multiple processors and disks configured 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. 6.1). Buyya et al. [18] define a cluster as "a type of parallel and distributed system, which consists of a collection 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

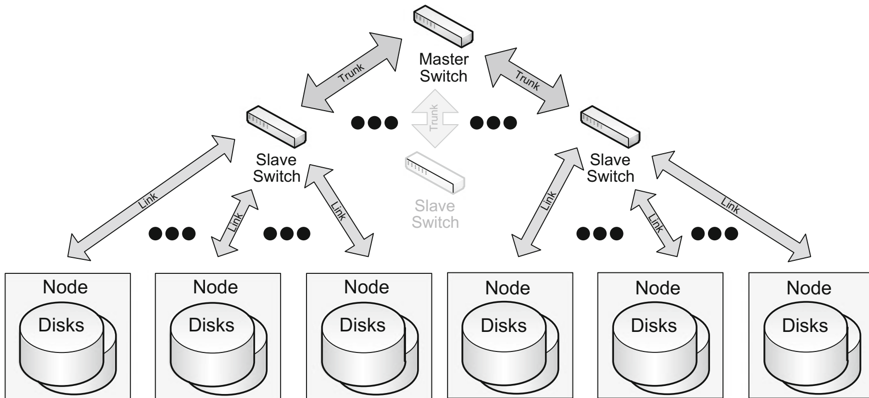

Fig. 6.1 Commodity hardware cluster [31]

problems are inherently adaptable to various forms of distributed computing including clusters and data grids and cloud computing.

Common Platform 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 [19]. In direct contrast to other types of computing and high-performance computing 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"—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 [2].

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 [20]. 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 high-performance computing 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. The ECL programming language described in this chapter was specifically designed to address data-intensive computing requirements.

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 high processing rates and throughput. 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.

HPCC Platform

HPCC System Architecture

The development of the open source HPCC computing platform by the Seisint subsidiary of LexisNexis began in 1999 and applications were in production by late 2000. The conceptual vision for this computing platform is depicted in Fig. 6.2. The LexisNexis approach also utilizes commodity clusters of hardware running the

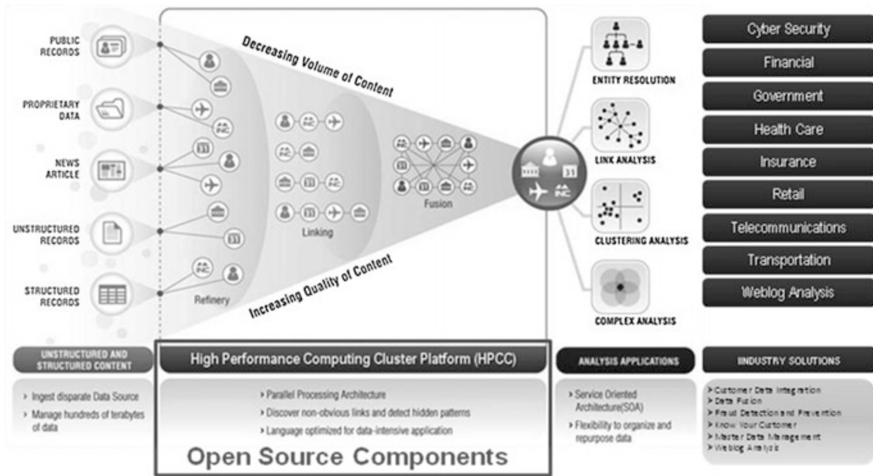

Fig. 6.2 LexisNexis vision for a data-intensive supercomputer

Linux operating system as shown in Figure 4.1. 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 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.

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 Thor processing cluster is depicted in Fig. 6.3. In addition to the Thor master and slave nodes, additional auxiliary and common components are needed to implement a complete HPCC processing environment. The actual number of physical nodes required for the auxiliary components is determined during the configurations process.

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 Roxie processing cluster is depicted in Fig. 6.4.

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 Hadoop where the MapReduce architecture must be overlayed with additional 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

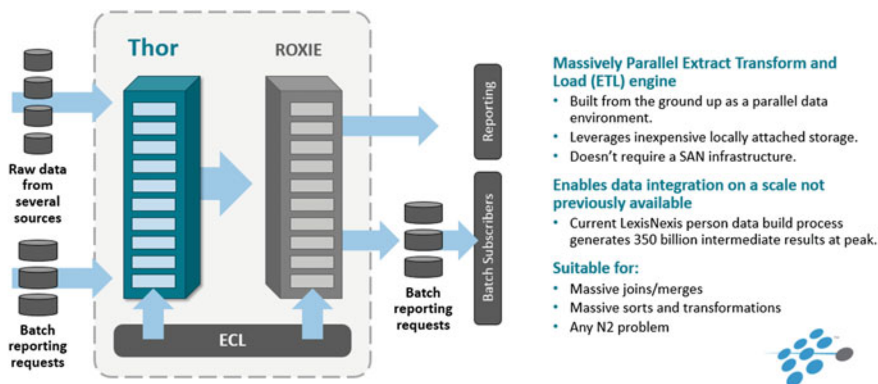

Fig. 6.3 HPCC Thor processing cluster

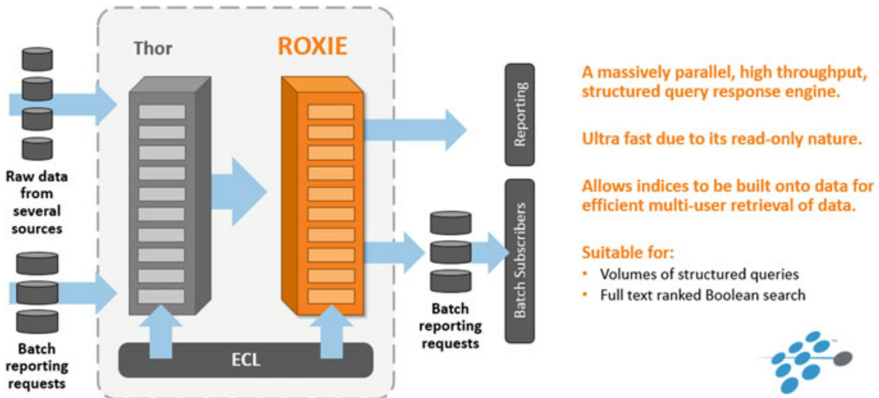

Fig. 6.4 HPCC Roxie processing cluster

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 significant flexibility when compared to Hadoop clusters which tend to be independent islands of processing. For additional information and a detailed comparison of the HPCC system platform to Hadoop, see [21].

HPCC Thor System Cluster

The Thor system cluster is implemented using a master/slave approach with a single master node and multiple slave nodes which provides a parallel job execution environment for programs coded in ECL. ECL is a declarative programming language, developed at LexisNexis, which is easy to use, data-centric and optimized for large-scale data management and query processing (Fig. 6.5). ECL is described in detail in "ECL Programming Language".

Each of the slave nodes is also a data node within the distributed file system for the cluster. Multiple Thor clusters can exist in an HPCC system 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 program is compiled by the ECL server which interacts with an additional server called the ECL Repository which is a source code repository and

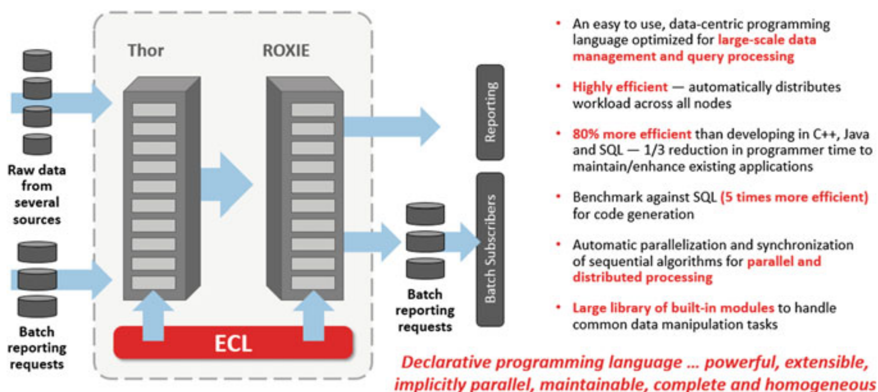

Fig. 6.5 ECL declarative programming language

contains shared, reusable ECL code. ECL code can also be stored in local source files and managed with a conventional version control system. ECL programs are compiled into optimized C++ source code, which is subsequently linked 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.

The distributed filesystem (DFS) used in a Thor cluster is record-oriented which is somewhat different from the block format used in MapReduce 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 with records in sequential 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. Index files generated on Thor clusters can also be directly copied to Roxie clusters to support online queries.

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. 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. Each node contains an integral number of records (individual records are not split across nodes), and I/O is completely localized to the processing node for local processing operations. 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 blocked data and input splits used in the MapReduce 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.

HPCC Roxie System 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 on a Roxie cluster 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 5000 transactions per second on a 100-node cluster. 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.

ECL Programming Language

Several well-known companies experiencing the big data problem have implemented high-level programming or script languages oriented toward data analysis. In Google's MapReduce programming environment, native applications are coded in C++ [22]. The MapReduce programming model allows group aggregations in parallel over a commodity cluster of machines similar to Figure 4.1. Programmers provide a Map function that processes input data and groups the data according to a key-value pair, and a Reduce function that performs aggregation by key-value on the output of the Map function. According to Dean and Ghemawat [22, 23], the processing is automatically parallelized by the system on the cluster, and takes care of details like partitioning the input data, scheduling and executing tasks across a processing cluster, and managing the communications between nodes, allowing programmers with no experience in parallel programming to use a large parallel processing environment. For more complex data processing procedures, multiple MapReduce calls must be linked together in sequence.

Google also implemented a high-level language named Sawzall for performing parallel data analysis and data mining in the MapReduce environment and a workflow management and scheduling infrastructure for Sawzall jobs called Workqueue [24]. For most applications implemented using Sawzall, the code is much simpler and smaller than the equivalent C++ by a factor of 10 or more. Pike et al. [24] cite several reasons 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.

An open source implementation of MapReduce pioneered by Yahoo! called Hadoop is functionally similar to the Google implementation except that the base programming language for Hadoop is Java instead of C++. Yahoo! also implemented a high-level dataflow-oriented language called Pig Latin and execution environment 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 [25]. Pig Latin programs are automatically translated into sequences of MapReduce programs if needed in the execution environment.

Both Google with its Sawzall language and Yahoo with its Pig system and language for Hadoop address some of the limitations of the MapReduce model by providing an external dataflow-oriented programming language which translates language statements into MapReduce processing sequences [24, 26, 27]. These languages provide many standard data processing operators so users do not have to implement custom Map and Reduce functions, improve reusability, and provide some optimization for job execution. However, these languages are externally implemented executing on client systems and not integral to the MapReduce architecture, but still rely on the on the same infrastructure and limited execution model provided by MapReduce.

ECL Features and Capabilities

The open source ECL programming language represents a new programming paradigm for data-intensive computing. ECL was specifically 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 Latin provides to Hadoop users, a 20 times increase in programmer productivity is typical which can significantly reduce 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 definition. 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. ECL can also be stored in local source files as with other programming languages. With ECL a rich set of programming tools is provided including an interactive IDE similar to Visual C++, Eclipse (an ECL add-in for Eclipse is available) and other code development environments.

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

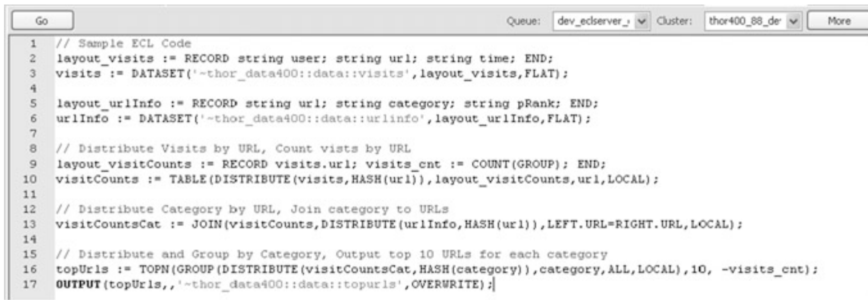

Fig. 6.6 ECL code example

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 6.6 shows a sample ECL program using the LOCAL mode of operation.

Figure 6.7 shows the corresponding execution graph. 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 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.

ECL Compilation, Optimization, and Execution

The ECL language compiler takes the ECL source code and produces an output with three main elements. The first is an XML representation of the execution graph, detailing the activities to be executed and the dependencies between those activities. The second is a C++ class for each of the activities in the graph, and the third contains code and meta information to control the workflow for the ECL program. These different elements are embedded in a single shared object that

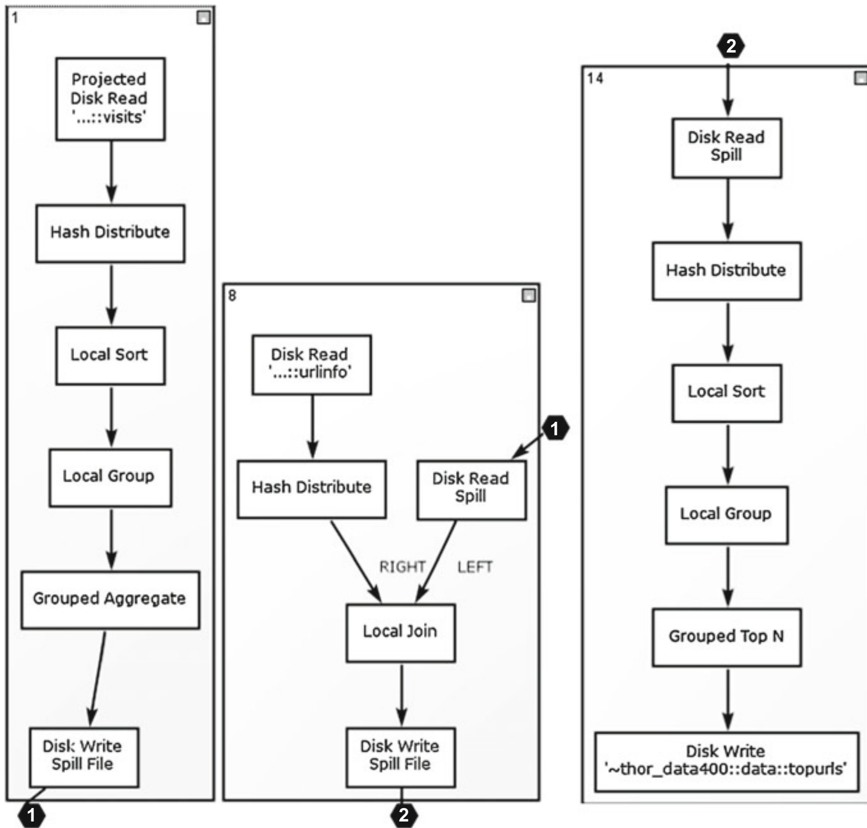

Fig. 6.7 ECL code example execution graph

contains all the information about the particular query. That shared object is passed to the execution engines, which take that shared object and execute the program it contains.

The process of compiling, optimizing, and executing the ECL is broken into several stages: (1) parsing, (2) optimization, (3) transforming, (4) generating, and (5) execution.

Parsing

The sources for an ECL program can come from a local directory tree, an external repository, or a single-source archive. The ECL compiler reads the ECL source, parses it, and converts it into an abstract graph representation of the program. The representation is then normalized to resolve ambiguities and ensure is it is suitable for subsequent processing. All of the subsequent operations within the compiler work on, and create, this same abstract representation.

Optimizations

The design of the ECL language provides abundant scope for optimizations. When reusable attributes are combined it often creates the scope for optimizations that would be hard, if not impossible, to be spotted by a programmer. Its declarative design allows many optimizations without the concerns about side-effects associated with imperative languages. Many different optimizations are performed on the program, some of the key ones are:

- *Constant folding*. This includes simple purely constant expressions like 12 * 3 => 36, and more complex changes e.g. IF(a, 'b', 'c') IN ['a','c'] => NOT a
- *Tracking and propagating constant field values*. This can often lead to further constant folding, or reduce the lifetime of a field. Minimizing the fields in a row at each stage of the processing. This saves the programmer from unnecessary optimization, and often benefits from the other optimizations (e.g., constant propagation).
- *Reordering operations*. Sometimes changing the order of operations can significantly reduce the data processed by complex activities. Examples include ensuring a filter is done before a sort, or replacing a filter on a joined dataset with a filter on one (or both) of the inputs.
- *Tracking meta information including sort orders and record counts, and removing redundant operations*. This is an example of an optimization which often comes into play when reusable attributes are combined. A particular sort order may not be part of the specification of an attribute, but the optimizer can make use of the current implementation.
- *Minimizing data transferred between slave nodes*. There is sufficient scope for many additional optimizations. For example, a currently planned optimization would analyze and optimize the distribution and sort activities used in a program to maximize overlap and minimize data redistribution.

A key design goal is for the ECL programmer to be able to describe the problem, and rely on the ECL compiler to solve the problem efficiently.

Transforming

The ECL compiler needs to transform the abstract declarative ECL (what it should do) to a concrete imperative implementation (how it should do it). This again has several different elements:

- *Convert the logical graph into an execution graph*. This includes introducing activities to split the data stream, ensure dependencies between activities will be executed in the correct order, and resolving any global resourcing constraints.

- *Extracting context-invariant expressions to ensure they are evaluated a minimal number of times*. This is similar to spotting loop invariant code in an imperative language.
- *Selecting between different implementations of a sequence of activities*. For example generating either inline code or a nested graph of activities.
- *Common sub-expression elimination*. Both globally across the whole program, and locally the expressions used within the methods of the activity classes.
- *Mapping complex ECL statements into the activities supported by the target engine*. For instance a JOIN may be implemented differently depending on how the inputs are sorted, distributed, and the likely size of the datasets. Similarly an ECL DEDUP operation may sometimes be implemented as a local dedup activity followed by a global dedup activity.
- *Combining multiple logical operations into a single activity*. Compound activities have been implemented in the engines where it can significantly reduce the data being copied, or because there are likely to be expressions shared between the activities. One of the commonest examples is disk read, filter and project.

Generating

Following the transforming stage, the XML and C++ associated with the ECL program is generated. The C++ code is built using a data structure that allows peephole optimizations to be applied to the C++ that will be generated. Once the processing is complete, the C++ is generated from the structure, and the generated source files are passed to the system C++ compiler to create a shared object.

In practice, the optimization, transforming and generation is much more of an iterative process rather than sequential.

Execution

The details of executing ECL program vary depending on the specific HPCC system platform and its execution engine, but they follow the same broad sequence.

The engine extracts resources from the shared object that describe the workflow of the query. The workflow can include waiting for particular events, conditionally re-evaluating expressions, and executing actions in a particular order. Each workflow item is executed independently, but can have dependencies on other workflow items. A workflow item may contain any number of activity graphs which evaluate a particular part of the ECL program.

To execute a graph of activities the engine starts at the outputs and recursively walks the graph to evaluate any dependencies. Once the graph is prepared the graph of activities is executed. Generally multiple paths within the graph are executed in parallel, and multiple slave nodes in a cluster will be executing the graphs on different subsets of the data. Records are streamed through the graphs from the inputs

to the outputs. Some activities execute completely locally, and others co-ordinate their execution with other slave nodes.

ECL Development Tools and User Interfaces

The HPCC platform includes a suite of development tools and utilities for data analysts, programmers, administrators, and end-users. These include ECL IDE, an integrated programming development environment similar to those available for other languages such as C++ and Java, which encompasses source code editing, source code version control, access to the ECL source code repository, and the capability to execute and debug ECL programs.

ECL IDE provides a full-featured Windows-based GUI for ECL program development and direct access to the ECL repository source code. ECL IDE allows you to create and edit ECL attributes which can be shared and reused in multiple ECL programs or to enter an ECL query which can be submitted directly to a Thor cluster as an executable job or deployed to a Roxie cluster. An ECL query can be self-contained or reference other sharable ECL code in the attribute repository. ECL IDE also allows you to utilize a large number of built-in ECL functions from included libraries covering string handling, data manipulation, file handling, file spray and despray, superfile management, job monitoring, cluster management, word handling, date processing, auditing, parsing support, phonetic (metaphone) support, and workunit services.

ECL Advantages and Key Benefits

ECL a heavily optimized, data-centric declarative programming language. It is a language specifically designed to allow data operations to be specified in a manner which is easy to optimize and parallelize. With a declarative language, you specify what you want done rather than how to do it. A distinguishing feature of declarative languages is that they are extremely succinct; it is common for a declarative language to require an order of magnitude (10×) less code than a procedural equivalent to specify the same problem [28]. The SQL language commonly used for data access and data management with RDBMS systems is also a declarative language. Declarative languages have many benefits including conciseness, freedom from side effects, parallelize naturally, and the executable code generated can be highly optimized since the compiler can determine the optimum sequence of execution instead of the programmer.

ECL extends the benefits of declarative in three important ways [28]: (1) It is data-centric which means it addresses computing problems that can be specified by some form of analysis upon data. It has defined a simple but powerful data algebra to allow highly complex data manipulations to be constructed; (2) It is extensible. When a programmer defines new code segments (called attributes) which can include macros, functions, data definitions, procedures, etc., these essentially

become a part of the language and can be used by other programmers. Therefore a new ECL installation may be relatively narrow and generic in its initial scope, but as new ECL code is added, its abilities expand to allow new problems and classes of problems to be stated declaratively; and (3) It is internally abstract. The ECL compiler generates C++ code and calls into many 'libraries' of code, most of which are major undertakings in their own right. By doing this, the ECL compiler is machine neutral and greatly simplified. This allows the ECL compiler writers to focus on making the language relevant and good, and generating highly-optimized executable code. For some coding examples and additional insights into declarative programming with ECL, see [29].

One of the key issues which has confronted language developers is to find solutions to the complexity and difficulty of parallel and distributed programming. Although high-performance computing and cluster architectures such have advanced to provide highly-scalable processing environments, languages designed for parallel programming are still somewhat rare. Declarative, data-centric languages because the parallelize naturally represent solutions to this issue [30]. According to Hellerstein, declarative, data-centric languages parallelizes naturally over large datasets, and programmers can benefit from parallel execution without modifications to their code. ECL code, for example can be used on any size cluster without modification to the code, so performance can be scaled naturally.

The key benefits of ECL can be summarized as follows:

- ECL is a declarative, data-centric, programming language which can expressed concisely, parallelizes naturally, is free from side effects, and results in highly-optimized executable code.
- ECL incorporates transparent and implicit 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 designed for manipulation of data and query processing. Order of magnitude performance increases over other approaches are possible.
- ECL provides a more than 20 times productivity improvement for programmers over traditional languages such as Java and C++. The ECL compiler generates highly optimized C++ for execution.
- 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.

HPCC High Reliability and High Availability Features

Thor and Roxie architectures of the HPCC system provide both high reliability and availability. The HPCC system in Fig. 6.8 shows the highly available architecture. In this architecture, Thor has several layers of redundancy:

1. Uses hardware RAID redundancy to isolate disk drive failure.
2. Two copies of the same data can exist on multiple nodes. This again is used to isolate against disk failure in one node or a complete node failure.
3. Multiple independent Thor clusters (as shown in Fig. 6.8) can be configured to subscribe to the same Job Queue. This is the highest form on redundancy available within Thor and this isolates you from disk failure, node failure and network failure within the same cluster.

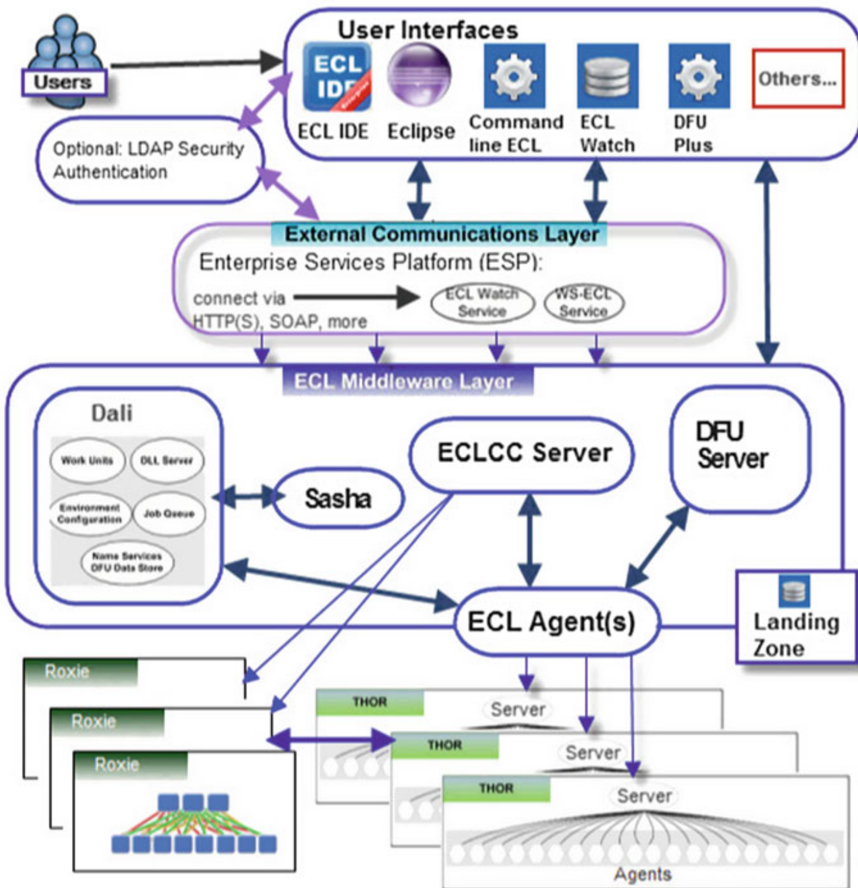

Fig. 6.8 High availability HPCC system architecture

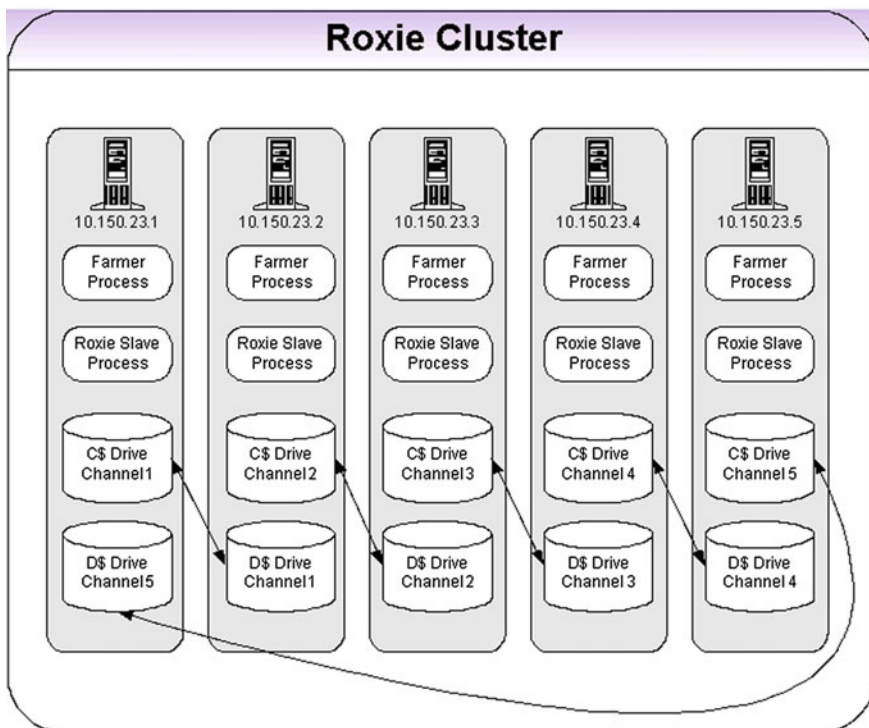

Fig. 6.9 Redundant architecture of the Roxie cluster

Thor cluster accepts jobs from a job queue. If there are two Thor clusters handling the queue, one will continue accepting jobs, if the other one fails. If a single component (Thorslave or Thormaster) fails, the other will continue to process requests. With replication enabled, it will be able to read data from the backup location of the broken Thor. Other components (such as ECL Server, or ESP) can also have multiple instances. The remaining components, such as Dali, or DFU Server, work in a traditional shared storage high availability fail over model.

The Roxie cluster has the highest form of redundancy, as illustrated in Fig. 6.9. Roxie will continue its operation even if half of the nodes are out of operation.

Conclusion

As a result of the continuing information explosion, many organizations are drowning in data and are experiencing the "Big Data" problem making it harder and harder to process and gain useful insights from their data. Data-intensive computing represents a new computing paradigm which can address the big data problem 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. Several organizations developed new parallel-processing architectures using commodity computing clusters including Google who initially developed the MapReduce architecture and LexisNexis who developed the HPCC architecture and the ECL programming language. An open source version of MapReduce called Hadoop was developed with additional capabilities to enhance the platform including a data-oriented programming language and execution environment called Pig. The open source HPCC platform and the ECL programming language are described in this chapter, and a direct comparison of the Pig language of Hadoop to the ECL language was presented along with a representative benchmark. Availability of a high-level declarative, data-centric, dataflow-oriented programming language has proven to be a critical success factor in data-intensive computing.

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 implemented using the ECL language 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 factors in choosing a new computer systems architecture and programming language, and usually the best approach is to conduct a specific benchmark test with a customer application to determine the overall system effectiveness and performance. A comparison of the Hadoop MapReduce architecture using a public benchmark for the Pig programming language to the HPCC architecture and ECL programming language on the same system hardware configuration in this chapter reveals significant performance advantages for the HPCC platform with ECL. Some additional advantages of choosing the LexisNexis HPCC platform with ECL include: (1) an open source 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) a scalable architecture which provides equivalent performance at a much lower system cost based on the number of processing nodes required compared to other data-intensive computing architectures such as MapReduce; (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 declarative, data-centric programming language (ECL) with extensive built-in capabilities for data-parallel processing, 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 in open source 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

1. Kouzes RT, Anderson GA, Elbert ST, Gorton I, Gracio DK. The changing paradigm of data-intensive computing. Computer. 2009;42(1):26–34.
2. Gorton I, Greenfield P, Szalay A, Williams R. Data-intensive computing in the 21st century. IEEE Comput. 2008;41(4):30–2.
3. Johnston WE. High-speed, wide area, data intensive computing: a ten year retrospective. In: Proceedings of the 7th IEEE international symposium on high performance distributed computing: IEEE Computer Society; 1998.
4. Skillicorn DB, Talia D. Models and languages for parallel computation. ACM Comput Surv. 1998;30(2):123–69.
5. Dowd K, Severance C. High performance computing. Sebastopol: O'Reilly and Associates Inc.; 1998.
6. Abbas A. Grid computing: a practical guide to technology and applications. Hingham: Charles River Media Inc; 2004.
7. Gokhale M, Cohen J, Yoo A, Miller WM. Hardware technologies for high-performance data-intensive computing. IEEE Comput. 2008;41(4):60–8.
8. Nyland LS, Prins JF, Goldberg A, Mills PH. A design methodology for data-parallel applications. IEEE Trans Softw Eng. 2000;26(4):293–314.
9. Agichtein E, Ganti V. Mining reference tables for automatic text segmentation. In: Proceedings of the tenth ACM SIGKDD international conference on knowledge discovery and data mining, Seattle, WA, USA; 2004. p. 20–9.
10. Agichtein E. Scaling information extraction to large document collections: Microsoft Research. 2004.
11. Rencuzogullari U, Dwarkadas S. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In: Proceedings of the eighth ACM SIGPLAN symposium on principles and practices of parallel programming, Snowbird, UT; 2001. p. 72–81.
12. Cerf VG. An information avalanche. IEEE Comput. 2007;40(1):104–5.
13. Gantz JF, Reinsel D, Chute C, Schlichting W, McArthur J, Minton S, et al. The expanding digital universe (White Paper): IDC. 2007.
14. Lyman P, Varian HR. How much information? 2003 (Research Report). School of Information Management and Systems, University of California at Berkeley; 2003.
15. Berman F. Got data? A guide to data preservation in the information age. Commun ACM. 2008;51(12):50–6.
16. NSF. Data-intensive computing. National Science Foundation. 2009. http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503324&org=IIS. Retrieved 10 Aug 2009.
17. PNNL. Data intensive computing. Pacific Northwest National Laboratory. 2008. http://www.cs.cmu.edu/~bryant/presentations/DISC-concept.ppt. Retrieved 10 Aug 2009.

18. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I. Cloud computing and emerging it platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comput Syst. 2009;25(6):599–616.
19. Gray J. Distributed computing economics. ACM Queue. 2008;6(3):63–8.
20. Bryant RE. Data intensive scalable computing. Carnegie Mellon University. 2008. http://www.cs.cmu.edu/~bryant/presentations/DISC-concept.ppt. Retrieved 10 Aug 2009.
21. Middleton AM. Data-intensive computing solutions (Whitepaper): LexisNexis. 2009.
22. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. In: Proceedings of the sixth symposium on operating system design and implementation (OSDI); 2004.
23. Dean J, Ghemawat S. Mapreduce: a flexible data processing tool. Commun ACM. 2010;53 (1):72–7.
24. Pike R, Dorward S, Griesemer R, Quinlan S. Interpreting the data: parallel analysis with sawzall. Sci Program J. 2004;13(4):227–98.
25. White T. Hadoop: the definitive guide. 1st ed. Sebastopol: O'Reilly Media Inc; 2009.
26. Gates AF, Natkovich O, Chopra S, Kamath P, Narayanamurthy SM, Olston C, et al. Building a high-level dataflow system on top of map-reduce: the pig experience. In: Proceedings of the 35th international conference on very large databases (VLDB 2009), Lyon, France; 2009.
27. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig latin: a not-so_foreign language for data processing. In: Proceedings of the 28th ACM SIGMOD/PODS international conference on management of data/principles of database systems, Vancouver, BC, Canada; 2008. p. 1099–110.
28. Bayliss DA. Enterrprise control language overview (Whitepaper): LesisNexis. 2010b.
29. Bayliss DA. Thinking declaratively (Whitepaper). 2010c.
30. Hellerstein JM. The declarative imperative. SIGMOD Rec. 2010;39(1):5–19.
31. O'Malley O. Introduction to hadoop. 2008. http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf. Retrieved 10 Aug 2009.
32. Bayliss DA. Aggregated data analysis: the paradigm shift (Whitepaper): LexisNexis. 2010a.
33. Buyya R. High performance cluster computing. Upper Saddle River: Prentice Hall; 1999.
34. Chaiken R, Jenkins B, Larson P-A, Ramsey B, Shakib D, Weaver S, et al. Scope: easy and efficient parallel processing of massive data sets. Proc VLDB Endow. 2008;1:1265–76.
35. Grossman R, Gu Y. Data mining using high performance data clouds: experimental studies using sector and sphere. In: Proceedings of the 14th ACM SIGKDD international conference on knowledge discovery and data mining, Las Vegas, Nevada, USA; 2008.
36. Grossman RL, Gu Y, Sabala M, Zhang W. Compute and storage clouds using wide area high performance networks. Future Gener Comput Syst. 2009;25(2):179–83.
37. Gu Y, Grossman RL. Lessons learned from a year's worth of benchmarks of large data clouds. In: Proceedings of the 2nd workshop on many-task computing on grids and supercomputers, Portland, Oregon; 2009.
38. Liu H, Orban D. Gridbatch: cloud computing for large-scale data-intensive batch applications. In: Proceedings of the eighth IEEE international symposium on cluster computing and the grid; 2008. p. 295–305.
39. Llor X, Acs B, Auvil LS, Capitanu B, Welge ME, Goldberg DE. Meandre: semantic-driven data-intensive flows in the clouds. In: Proceedings of the fourth IEEE international conference on eScience; 2008. p. 238–245.
40. Pavlo A, Paulson E, Rasin A, Abadi DJ, Dewitt DJ, Madden S, et al. A comparison of approaches to large-scale data analysis. In: Proceedings of the 35th SIGMOD international conference on management of data, Providence, RI; 2009. p. 165–68.
41. Ravichandran D, Pantel P, Hovy E. The terascale challenge. In: Proceedings of the KDD workshop on mining for and from the semantic web; 2004.
42. Yu Y, Gunda PK, Isard M. Distributed aggregation for data-parallel computing: interfaces and implementations. In: Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles, Big Sky, Montana, USA; 2009. p. 247–60.