

White Paper

# HPCC Systems: Introduction to HPCC (High-Performance Computing Cluster)

Author: Anthony M. Middleton, Ph.D. LexisNexis Risk Solutions

Date: May 24, 2011

## Executive Summary

As a result of the continuing information explosion, many organizations now have the need to process and analyze massive volumes of data. These data-intensive computing requirements can be addressed by scalable systems based on hardware clusters of commodity servers coupled with system software to provide a distributed file storage system, job execution environment, online query capability, parallel application processing, and parallel programming development tools. The LexisNexis HPCC platform provides all of these capabilities in an integrated, easy to implement and use, commercially-available high-performance computing environment. This paper provides an introduction to the LexisNexis HPCC system architecture also referred to (in the government space) as the LexisNexis Data Analytics Supercomputer (DAS).

LexisNexis Risk Solutions, an industry leader in data content, data aggregation, and information services, independently developed and implemented the HPCC platform as a solution for its own data-intensive computing requirements. In a similar manner to Hadoop (the open source implementation of MapReduce), the LexisNexis approach also uses commodity clusters of hardware running the Linux operating system and includes additional system software and middleware components to provide a complete and comprehensive job execution environment and distributed query and filesystem support needed for data-intensive computing.

The HPCC platform includes a powerful high-level, heavily-optimized, data-centric declarative language for parallel data processing called ECL (Enterprise Data Control Language) which is also described in this paper. The power, flexibility, advanced capabilities, speed of development, maturity, and ease of use of the ECL programming language is a primary distinguishing factor between the LexisNexis HPCC platform and other data-intensive computing solutions.

Advantages of selecting the LexisNexis HPCC platform for data-intensive computing include: (1) a highly integrated system environment with capabilities from raw data processing to high-performance queries and data analysis using a common language; (2) an optimized cluster approach which provides high performance at a much lower system cost than other system alternatives resulting in significantly lower total cost of ownership (TCO); (3) a stable and reliable processing environment proven in production applications for varied organizations over a 10-year period; (4) an innovative data-centric programming language (ECL) with extensive built-in capabilities for data-parallel processing, significantly increasing programmer productivity for application development, which automatically optimizes execution graphs with hundreds of processing steps into single efficient workunits; (5) a high-level of fault resilience and capabilities which reduce the need for re-processing in case of system failures; (6) suitability for a wide range of data-intensive applications from large volume ETL processing to support databases, data warehouses, and high volume online applications to network security analysis of massive amounts of log information; and (7) available from and supported by a well-known leader in information services and “large data” solutions (LexisNexis) which is part of one of the world’s largest publishers of information – ReedElsevier.

## Table of Contents

Executive Summary .....	2
List of Figures .....	4
Introduction.....	5
High-Performance Computing .....	5
Commodity Computing Clusters .....	6
Data-Intensive Computing Applications .....	7
MapReduce .....	8
Hadoop.....	9
Current Limitations of MapReduce .....	11
HPCC Platform Overview .....	13
Cluster Types .....	14
Data Refinery (Thor) .....	17
Rapid Data Delivery Engine (Roxie) .....	17
The ECL Programming Language .....	18
Key Benefits of ECL .....	21
ECL Programming Example .....	21
HPCC Middleware and System Servers.....	27
Development Tools and User Interfaces.....	27
Using a Thor Cluster .....	31
Using a Roxie Cluster .....	32
Thor and Roxie Together: A Complete Solution .....	32
HPCC Performance .....	33
Terabyte Sort Benchmark .....	53
Conclusions .....	35
Glossary .....	36
Reference List.....	38

## List of figures

1	Commodity Computing Cluster .....	6
2	MapReduce Processing Architecture.....	9
3	Hadoop MapReduce .....	10
4	Sample Pig Latin Program .....	11
5	LexisNexis Vision for a Data Analytics Supercomputer .....	13
6	Thor Processing Cluster .....	14
7	Roxie Processing Cluster .....	15
8	HPCC System Architecture.....	16
9	ECL Sample Syntax for JOIN operation.....	19
10	ECL Code Example .....	20
11	ECL Programming Example – Log File Analysis Macro .....	22
12	ECL Programming Example – Log File Output Format .....	23
13	ECL Programming Example – Log File Analysis Job.....	24
14	ECL Programming Example – Log File Analysis Graph.....	25
15	ECL Programming Example – Log File Analysis Output.....	26
16	QueryBuilder IDE .....	28
17	ECLWatch Web-based Utility.....	29
18	ECLWatch Job Execution Graph .....	30
19	Hadoop Terabyte Sort Benchmark Results .....	34
20	HPCC Terabyte Sort Benchmark Results .....	34

## Introduction

Many organizations have large amounts of data which has been collected and stored in massive datasets which needs be processed and analyzed to provide business intelligence, improve products and services for customers, or to meet other internal data processing requirements. For example, Internet companies need to process data collected by Web crawlers as well as logs, click data, and other information generated by Web services. Parallel relational database technology has not proven to be cost-effective or provide the high-performance needed to analyze massive amounts of data in a timely manner [1-3]. As a result several organizations developed technology to utilize large clusters of commodity servers to provide high-performance computing capabilities for processing and analysis of massive datasets. Clusters can consist of hundreds or even thousands of commodity machines connected using high-bandwidth networks. Examples of this type of cluster technology include Google's MapReduce [1, 4], Hadoop [5, 6], SCOPE [2], Sector/Sphere [7], and LexisNexis HPCC platform described in this paper.

This paper will introduce high-performance computing utilizing clusters of commodity hardware, describe the characteristics and requirements of data-intensive applications, and also briefly discuss the MapReduce programming model and Hadoop system as an example of a basic cluster system architecture for comparison. This is followed by an overview of LexisNexis HPCC platform and the ECL Programming language describing its advantages over other approaches.

### *High-Performance Computing*

High-Performance Computing (HPC) is used to describe computing environments which utilize supercomputers and computer clusters to address complex computational requirements, support applications with significant processing time requirements, or require processing of significant amounts of data. Supercomputers have generally been associated with scientific research and compute-intensive types of problems, but more and more supercomputer technology is appropriate for both compute-intensive and data-intensive applications. Supercomputers utilize a high-degree of internal parallelism and typically use specialized multi-processors with custom memory architectures which have been highly-optimized for numerical calculations [8]. Supercomputers also require special parallel programming techniques to take advantage of its performance potential.

Today a higher-end desktop workstation has more computing power than the supercomputers which existed during the early 1990's. This has led to a new trend in supercomputer design for high-performance computing: using clusters of independent processors connected in parallel [9]. Many computing problems are suitable for parallelization, often problems can be divided in a manner so that each independent processing node can work on a portion of the problem in parallel by simply dividing the data to be processed, and then combining the final processing results for each portion. This type of parallelism is often referred to as data-parallelism, and data-parallel applications are a potential solution to petabyte scale data processing requirements [10, 11].

Data-parallelism can be defined 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. The most important reason for developing data-parallel applications is the potential for scalable performance in high-performance computing, 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, communications between processing nodes, and the overall accuracy of the results [10]. Nyland et al. [10] 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.

### Commodity Computing Clusters

The resulting economies of scale in using multiple independent processing nodes for supercomputer design to address high-performance computing requirements led directly to the implementation of commodity computing clusters. A computer cluster is a group of shared individual computers, linked by high-speed communications in a local area network topology using technology such as gigabit network switches or InfiniBand, and incorporating system software which provides an integrated parallel processing environment for applications with the capability to divide processing among the nodes in the cluster. Cluster configurations can not only improve the performance of applications which use a single computer, but provide higher availability and reliability, and are typically much more cost-effective than single supercomputer systems with equivalent performance. The key to the capability, performance, and throughput of a computing cluster is the system software and tools used to provide the parallel job execution environment. Programming languages with implicit parallel processing features and a high-degree of optimization are also needed to insure high-performance results as well as high programmer productivity.

Clusters allow the data used by an application to be partitioned among the available computing resources and processed independently to achieve performance and scalability based on the amount of data. 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 (Figure 1). Clusters are extremely effective when 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.

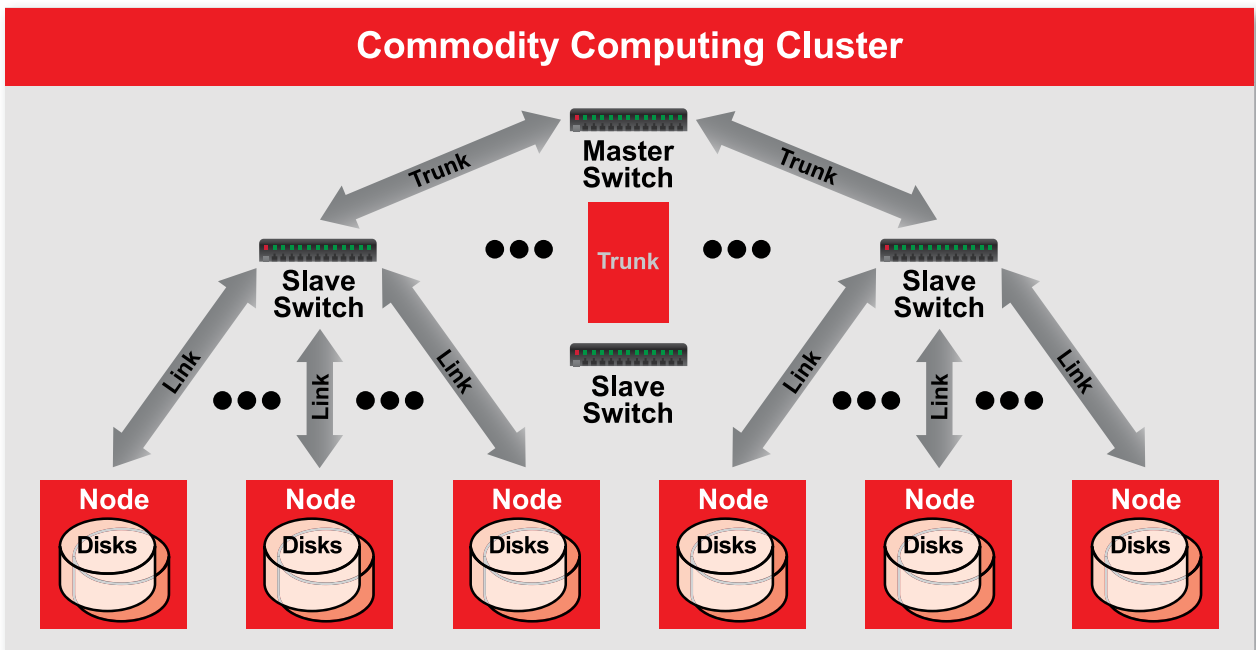


Figure 1

High-performance clusters are usually configured using commercial off-the-shelf (COTS) PC components. Rack-mounted servers or blade servers each with local memory and disk storage are often used as processing nodes to allow high-density small footprint configurations which facilitate the use of very high-speed communications equipment to connect the nodes. Linux is widely used as the operating system for computer clusters [13, 14]. According to Sloan [14], cluster configurations can be symmetric (each node can also function as a separate individual computer) or asymmetric (one computer functions as the master node providing a gateway to users and managing the activity of other nodes) which is the most common architecture. Cluster management, security, and workload distribution are less problematic and optimum performance is usually more easily achieved in asymmetric clusters. The hardware utilized in high-performance computing clusters is typically homogeneous, with each processing node consisting of the same processor, memory, and disk components. This enables the system software to better optimize workloads and deliver more consistent performance for parallel processing applications. In a parallel processing application on a cluster where the workload has been divided evenly, a node which has a slower processor or less memory will lag other nodes in completing its part of an application affecting overall performance.

### *Data-Intensive Computing Applications*

Data-intensive is used to describe computing applications that are I/O bound or with a need to process large volumes of data [15-17]. 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 [10]. The greater the aggregate distribution of the data, the more benefit there is in parallel processing of the data. Data-intensive processing requirements normally scale linearly according to the size of the data and are very amenable to straightforward parallelization.

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. Most other types of computing and supercomputing utilize data stored in a separate repository or servers and transfer the data to the processing system for computation. Data-intensive computing typically 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 [15]. The use of high-bandwidth network switching capabilities also allows file system clusters and processing clusters to be interconnected to provide even more processing flexibility.

A second important characteristic of data-intensive computing systems is the programming model utilized. Data-intensive computing systems typically 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 [18]. The programming abstraction and language tools allow the processing to be expressed in terms of data flows and transformations incorporating new data-centric 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 typically 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<sup>1</sup>). 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.

### *MapReduce*

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 [19]. 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 [20] and new processing paradigms with more flexible data models were needed. Internet companies such as Google, Yahoo, 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. Google MapReduce is an example of a basic system architecture designed for processing and analyzing large datasets on commodity computing clusters and is being used successfully by Google in many applications to process massive amounts of raw Web data [1, 4]. LexisNexis, an acknowledged industry leader in information services and “large data” solutions, also developed and implemented a scalable platform for data-intensive computing called HPCC which offers significantly more capability than MapReduce and has been used for several years by LexisNexis and other commercial and government organizations to process very large volumes of structured and unstructured data.

The MapReduce programming model allows group aggregations in parallel over a cluster of machines. 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 in [1, 4], 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. The overall model for this process is shown in Figure 2. For more complex data processing procedures, multiple MapReduce calls must be linked together in sequence.

<sup>1</sup>) BORPS an acronym for Billions Of Records Per Second first introduced by Seisint, Inc. in 2001.



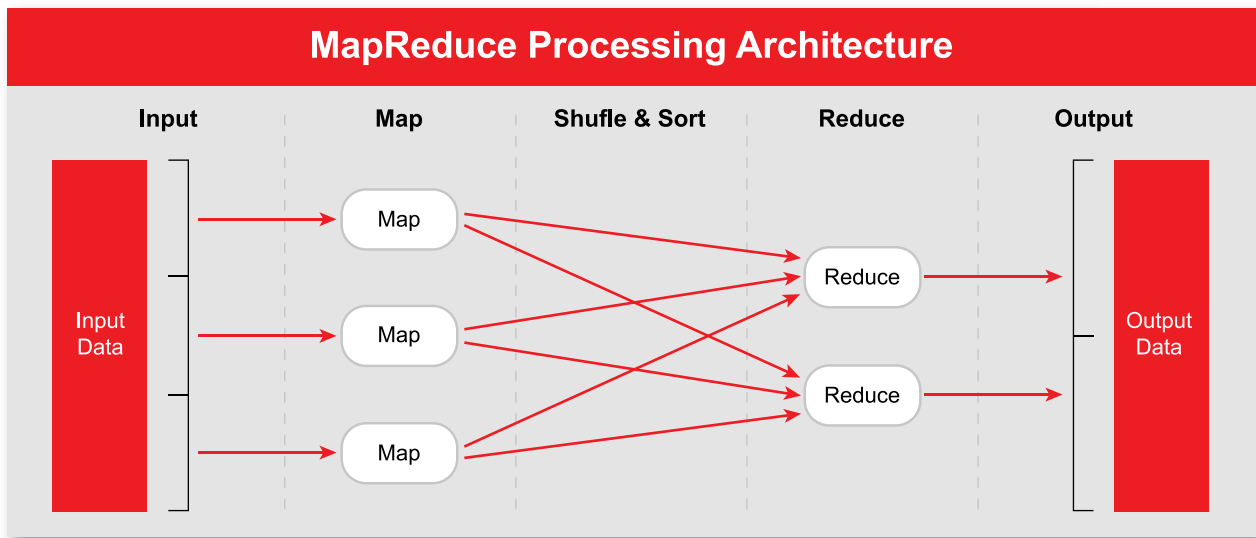


Figure 2

Underlying and overlaid on the same computing cluster 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 [21]. GFS is oriented to very large files dividing and storing them in fixed-size chunks of 64 Mb by default. 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.

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 [22]. 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. in [22] 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 than equivalent C++ MapReduce programs and significantly easier to program.

#### *Hadoop.*

Hadoop is an open source software project sponsored by The Apache Software Foundation (<http://www.apache.org>) initiated to create an open source implementation of the MapReduce architecture [6]. The Hadoop MapReduce architecture shown in Figure 3 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 utilizing Linux as the operating system environment.

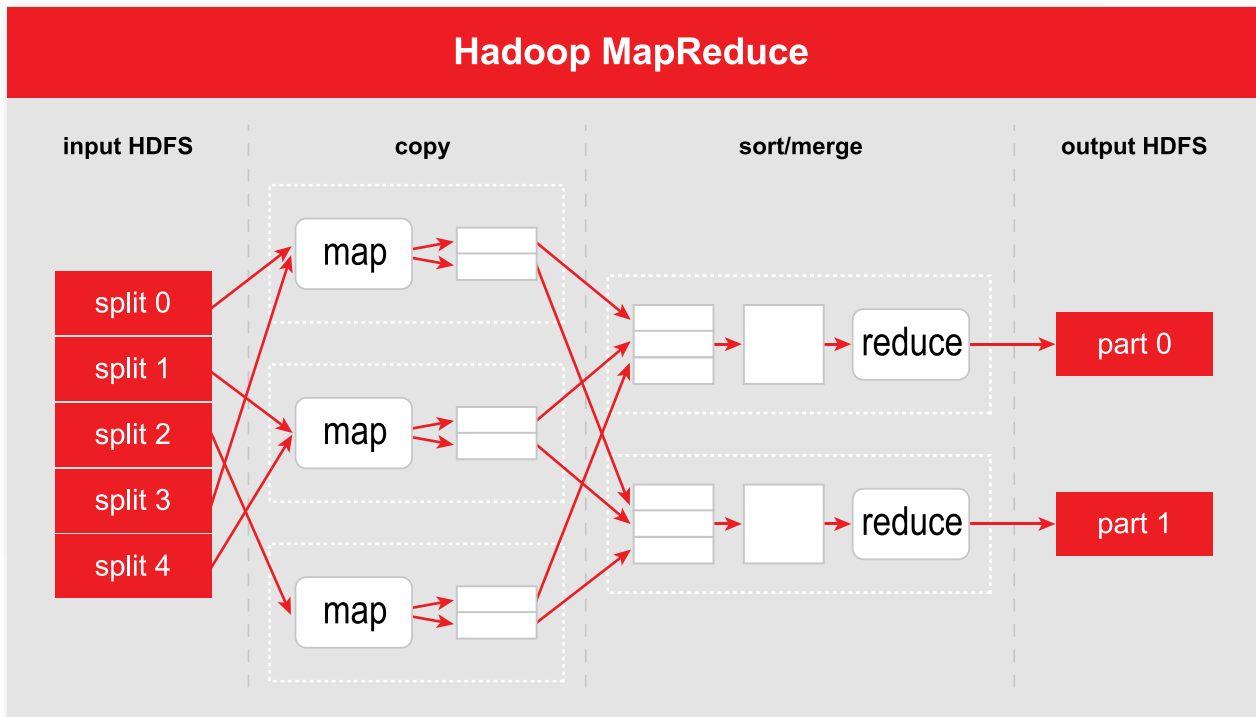


Figure 3

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 [6]. 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. Hadoop includes a distributed file system called HDFS which is analogous to GFS in the Google MapReduce implementation. 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 Hadoop job execution environment supports additional distributed data processing capabilities which are designed to run using the Hadoop MapReduce architecture including the Pig system. Pig includes a 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.

## Sample Pig Latin Program

```
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';
```

Figure 4

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 [6]. Pig programs are automatically translated into sequences of MapReduce programs if needed in the execution environment. An example program is shown in Figure 4 which requires execution of 3 separate MapReduce jobs.

### *Current Limitations of MapReduce.*

Although the MapReduce model and programming abstraction provides basic functionality for many data processing operations, users are limited by its rigid structure and forced to adapt their applications to the model in order to achieve parallelism. This can require implementation of multiple MapReduce sequences for more complex processing requirements that may need to perform multiple sequenced operations or operations such as joining multiple input files which can add substantial job management overhead to the overall processing time, as well as limit opportunities for optimization of the processing with different execution strategies. In addition many data processing operations do not fit naturally into the group-by-aggregation model using single key-value pairs required by the model. Even simple operations such as projection and selection must be fit into this model and users must provide custom Map and Reduce functions for all applications which is more error-prone and limits reusability [2]. Since custom Map and Reduce functions must be provided for each step, the inability to globally optimize the execution of complex data processing sequences can result in significantly degraded performance.

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[22, 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.

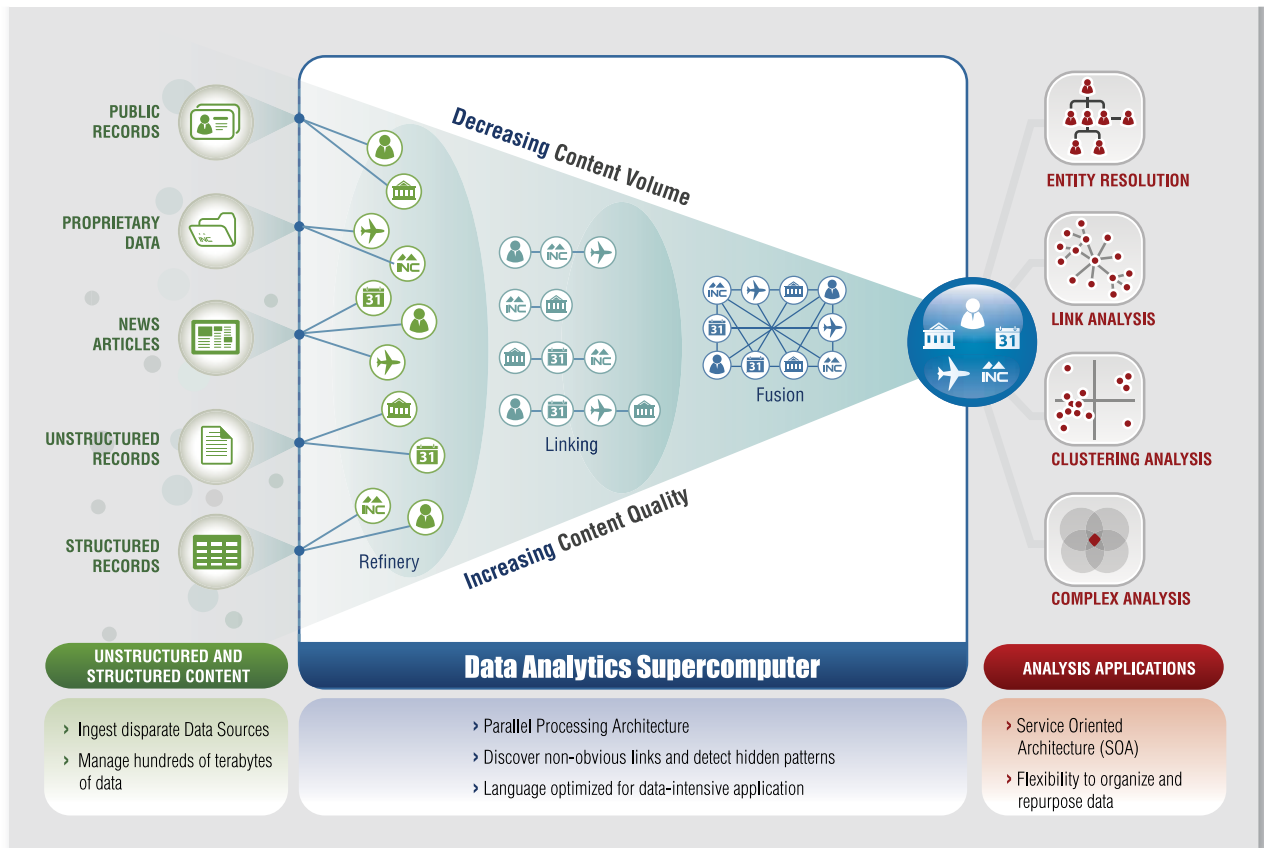
The MapReduce model is designed to operate in a parallel batch processing environment which is useful for performing ETL (Extract, Transform, Load) work on large datasets which must be transformed for some other use such as building inverted indexes. The system is also useful for batch queries performing aggregation operations or complex analytical tasks on large datasets and particularly unstructured data without the need for building indices or loading into a relational DBMS [1]. However, for online efficient querying of large datasets which must support large numbers of users

or provide fast response times with random access to structured data and support data warehouse applications such as that provided by parallel DBMS systems [3], other platforms are required in a MapReduce environment. Google has addressed this requirement by adding BigTable [28], and Hadoop with Hbase and Hive [6]. These operate essentially as bolt-ons to the MapReduce architecture utilizing the underlying file storage systems and MapReduce processing but otherwise operating as independent non-integrated applications. A better approach would be an integrated system environment which excels at both ETL tasks and complex analytics, and at efficient querying of large datasets using a common data-centric parallel processing language. The LexisNexis HPCC system platform was designed exactly for this purpose.

*3) In such a situation, the main strength of the analysis will have been performed during the record selection process.*

## HPCC Platform Overview

LexisNexis, an industry leader in data content, data aggregation, and information services independently developed and implemented a solution for data-intensive computing called 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 Figure 5.



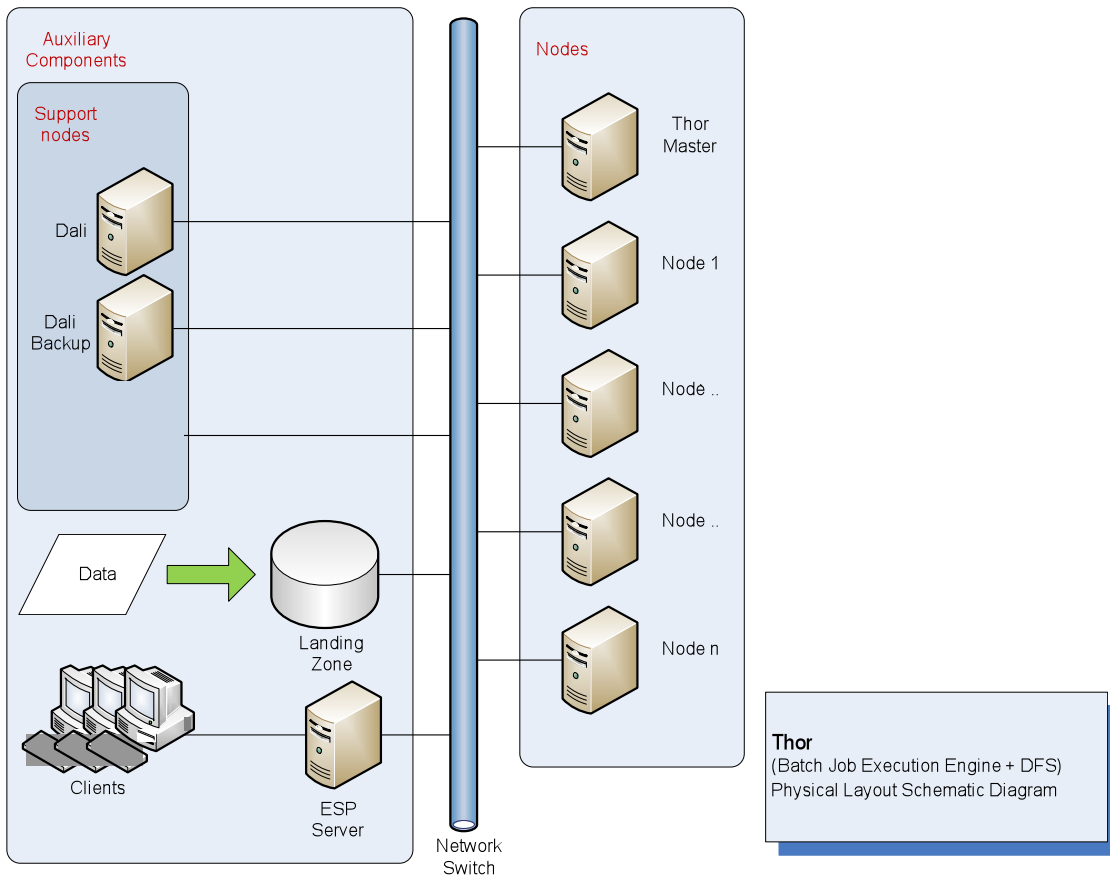
**Figure 5** LexisNexis Vision for a Data Analytics Supercomputer.

The development of this computing platform by the Seisint, Inc. (acquired by LexisNexis in 2004) 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 similar to the cluster depicted in Figure 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 Data Control Language). The power, flexibility, advanced capabilities, speed of development, maturity, and ease of use of the ECL programming language is a primary distinguishing factor between the LexisNexis HPCC platform and other data-intensive computing solutions.

4) Entity extraction being a good example.

## Cluster Types

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 cluster 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, record linking and entity resolution, large-scale ad-hoc complex analytics, 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 cluster is similar in its function, execution environment, filesystem, and capabilities to the Google and Hadoop MapReduce platforms, but offers significantly higher performance in equivalent configurations.

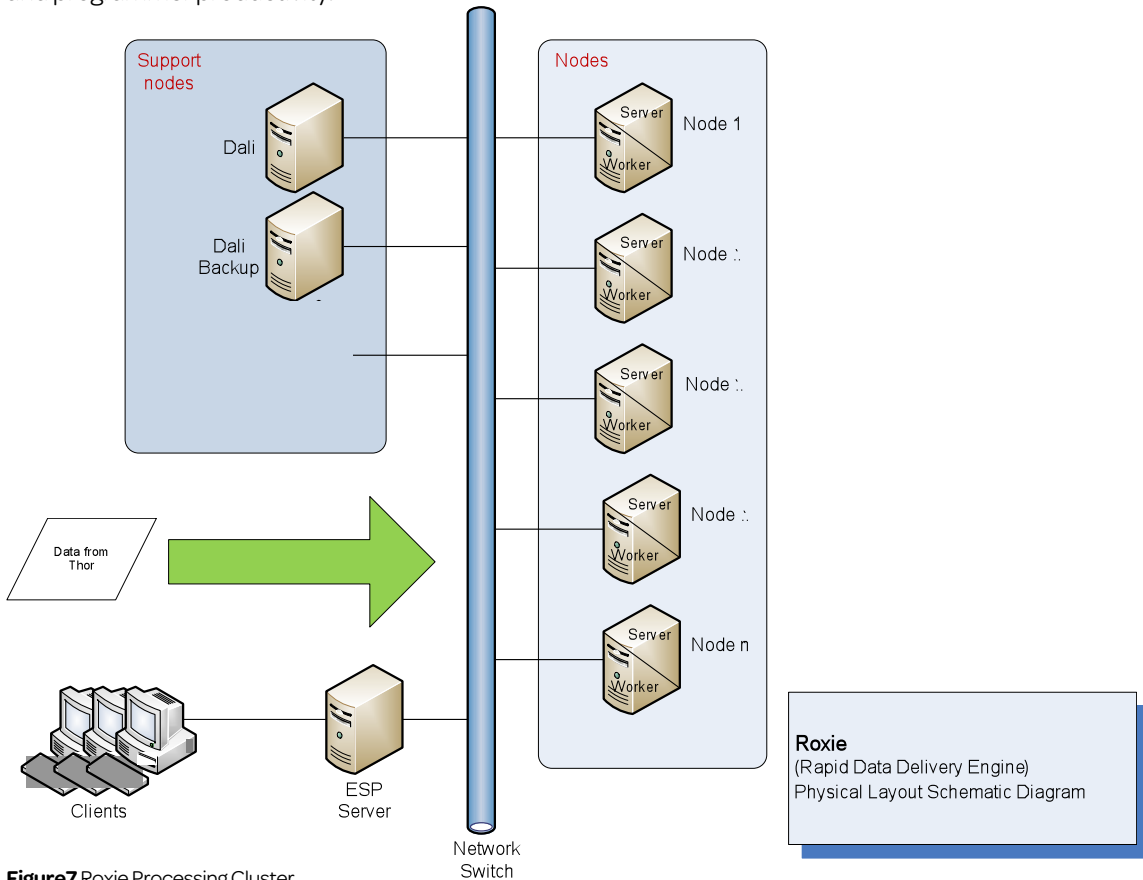


**Figure 6** Thor Processing Cluster.

Figure 6 shows a representation of a physical Thor processing cluster which functions as a batch job execution engine for scalable data-intensive computing applications. 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 Rapid 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. Online applications developed by LexisNexis such as Accurint® utilize both Thor and Roxie platforms. The Rapid 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 cluster 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 clusters utilize the same ECL programming language for implementing applications, increasing continuity and programmer productivity.



**Figure 7** Roxie Processing Cluster.

Figure 7 shows a representation of a physical Roxie processing cluster which functions as an online query execution engine for high-performance query and data warehousing applications. A Roxie cluster includes multiple nodes with server and worker processes for processing queries; an additional auxiliary component called an ESP server which provides interfaces for external client access to the cluster; and additional common components which are shared with a Thor cluster in an HPCC environment. Although a Thor processing cluster can be implemented and used without a Roxie cluster, an HPCC environment which includes a Roxie cluster must also include a Thor cluster. The Thor cluster is required to build the distributed index files used by the Roxie cluster and to develop online queries which will be deployed with the index files to the Roxie cluster. The specific function of the auxiliary and common HPCC components are discussed later in this paper.

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 overlaid 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 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 [29].

The HPCC system architecture incorporates the Thor and Roxie clusters as well as common middleware components, an external communications layer, client interfaces which provide both end-user services and system management tools, and auxiliary components to support monitoring and to facilitate loading and storing of filesystem data from external sources. An HPCC environment can include only Thor clusters, or both Thor and Roxie clusters. Each of these cluster types is described in more detail in the following sections. The overall HPCC system architecture is shown in Figure 8.

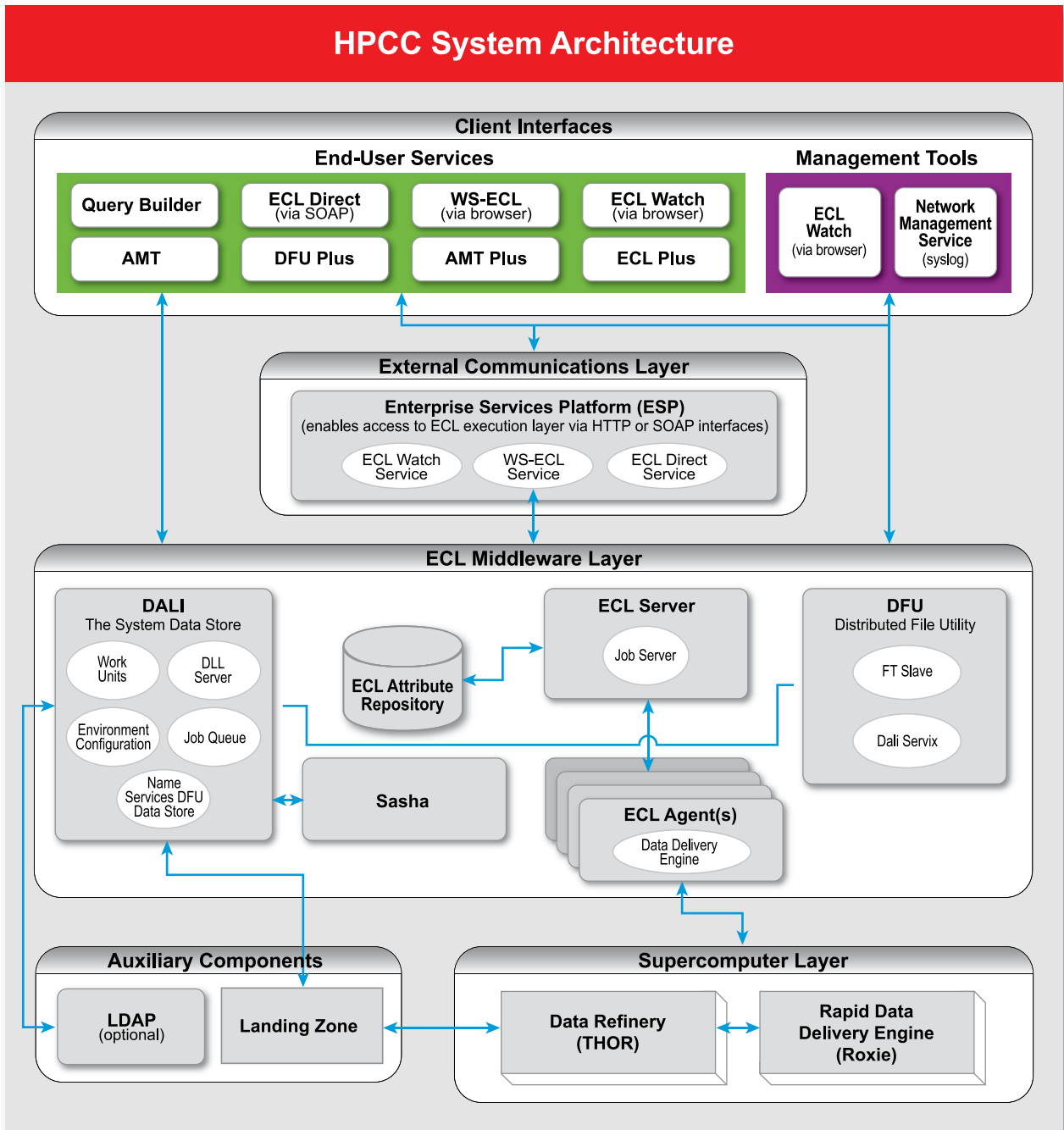


Figure8



### *Data Refinery (Thor)*

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. 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 contains shared, reusable ECL code. 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.

### *Rapid Data Delivery Engine (Roxie)*

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 than Thor clusters.

### *The ECL Programming Language*

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, highly-optimized, data-centric declarative 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++, a 20 times increase in programmer productivity is typical.

ECL is compiled into optimized C++ code for execution on the HPCC system platform, 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 IDE called QueryBuilder similar to Visual C++, Eclipse and other interactive code development environments.

**JOIN**

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

**JOIN**(*setofdatabases*, *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 *setofdatabases* (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.

*setofdatabases*    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 *setofdatabases* and also the output sort order of the result set.

*fields*            A comma-delimited list of fields in the *setofdatabases*, 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.

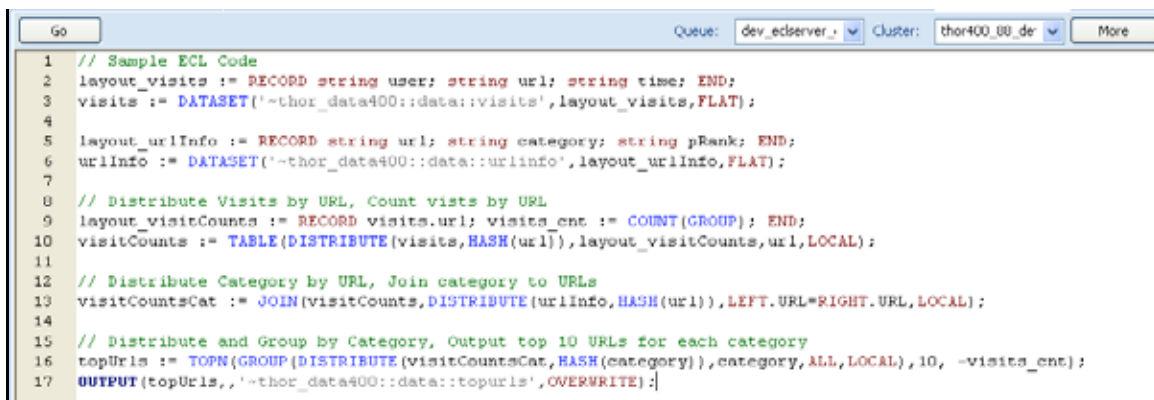
The **JOIN** function produces a result set based on the intersection of two or more datasets or indexes (as determined by the *joincondition*).

**Figure 9** ECL Sample Syntax for JOIN operation.

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 Figure 9.

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 HPC cluster would sort the records by the specified keys 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 10 shows a sample ECL program using the LOCAL mode of operation which is the equivalent of the sample PIG program for Hadoop shown in Figure 4. 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 declarative definitions.



```
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 visits by URL
9 layout_visitCounts := RECORD visits.url; visits.ent := 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 topUris := TOPN(GROUP(DISTRIBUTE(visitCountsCat, HASH(category)), category, ALL, LOCAL), 10, -visits.ent);
17 OUTPUT(topUris, '-thor_data400::data:topuris', OVERWRITE);
```

Figure 10 ECL Code Example.

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 operation. 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 operates 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 is possible to implement parallel processing for information extraction applications across document files including XML-based documents or Web pages.

### *Key Benefits of ECL*

The key benefits of ECL can be summarized as follows:

1. 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.
2. 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. Orders of magnitude performance increases over other approaches are possible.
3. ECL provides a more than 20 times productivity improvement for programmers over languages such as Java and C++. The ECL compiler generates highly optimized C++ for execution.
4. ECL is a powerful, high-level, parallel programming language ideal for implementation of ETL, information retrieval, information extraction, record linking and entity resolution, and many other data-intensive applications.
5. ECL is a mature and proven language but still evolving as new advancements in parallel processing and data-intensive computing occur.

ECL also provides a comprehensive IDE and programming tools that provide a highly-interactive environment for rapid development and implementation of ECL applications.

### *ECL Programming Example*

Analysis of log data collected by Web servers, system servers, and other network devices such as routers and firewalls is an important application for generating statistical information and reports on system and network utilization and other types of analysis such as intrusion detection and misuse of network resources. Log data is usually collected in unstructured text files which must be parsed using NLP to extract key information for reporting and analysis. This is typical of many data processing applications which must process data in a raw form, extracting, transforming, and loading the data for subsequent processing and is commonly referred to as ETL processing. The volume of log data generated by a large network of system and network servers can be enormous and is representative of applications which require a data-intensive computing solution like the LexisNexis HPCC platform.

Since log files from various system servers and networks devices can have varying formats, but a network generally includes multiples of the same types of devices which use common log formats, a useful design approach is to generate a function or macro for each type of device. The ECL programming language includes both functions and macros, and a macro format was selected for this example. A macro in a programming language accepts parameters similar to a function, and substitutes the parameter values to replace parts of the code generated by the macro, generating new inline code each time it is referenced.



```

1 export dataset MAC_Parse_DTSH_Keyval_Format (inlogfile, outlogfile, outerrorfile) := MACRO
2
3 import Text;
4
5 layout_log_seq := record, maxlength(8192)
6 unsigned4 linenum;
7 string line;
8 end;
9
10 log_seq := project(inlogfile,
11                   transform(layout_log_seq,
12                             self.linenum := count,
13                             self := left));
14
15 // Log file special patterns
16 pattern anynoteolon := any not in ':';
17 pattern log_source := anynoteolon*;
18 pattern log_msg_type := anynoteolon*;
19 pattern log_info := any*;
20 pattern logline := Text.date Text.ws Text.ISO_Time Text.ws log_source ':' Text.ws log_msg_type ':' Text.ws log_info;
21
22 // Output record format for parse
23 logfields := record, maxlength(16384)
24 unsigned4 linenum := log_seq.linenum;
25 string date := matchtext(logline/Text.date);
26 string time := matchtext(logline/Text.ISO_Time);
27 string log_source := matchtext(logline/log_source);
28 string log_msg_type := matchtext(logline/log_msg_type);
29 string log_info := matchtext(logline/log_info);
30 end;
31
32 // Parse log info from raw input log file
33 log_init := parse(log_seq, line, logline, logfields, first, maxlength(8192));
34
35 // extract key value pairs from log_info
36 pattern key := (any not in {' '|'\t'|'\r'|'\n'|'\0'|'\1'|'\2'|'\3'|'\4'|'\5'|'\6'|'\7'|'\8'|'\9'|'\a'|'\c'|'\e'|'\f'|'\g'|'\h'|'\i'|'\j'|'\k'|'\l'|'\m'|'\n'|'\o'|'\p'|'\q'|'\r'|'\s'|'\t'|'\u'|'\v'|'\w'|'\x'|'\y'|'\z'|'\{|'\}|'\~'|'\.'|'\:','\|'})+;
37 pattern val := key | ('"' (any not in '"'|'\|')* '"');
38 pattern key_value := key '-' val;
39
40 keyfields := record, maxlength(4096)
41 unsigned4 linenum := log_init.linenum;
42 string key := matchtext(key_value/key);
43 string val := Stringlib.StringFilterOut(matchtext(key_value/val), '"');
44 end;
45
46 keyvals_init := parse(log_init, log_info, key_value, keyfields, many, max, scan, nocase, maxlength(8192));
47
48 // Combine key-value pairs as a child dataset with fixed fields
49 layout_logout := Log_Analysis.Layout_DTSH_Keyval;
50
51 // Initialize output log
52 log_out_init := project(log_init,
53                         transform(layout_logout,
54                                   self := left,
55                                   self := []));
56
57 // Create error log
58 outerrorfile := join(log_seq,
59                     log_out_init,
60                     left.linenum = right.linenum,
61                     transform(recordof(log_seq),
62                               self := left),
63                     left only,
64                     hash);
65
66 // Denormalize key value pairs
67 outlogfile := sort(denormalize(distribute(log_out_init, hash(linenum)),
68                               sort(distribute(keyvals_init, hash(linenum)), linenum, key, local),
69                               left.linenum = right.linenum,
70                               transform(layout_logout,
71                                         self.keyvals := left.keyvals +
72                                                         row((right.key, right.val), Log_Analysis.layout_keyval),
73                                         self := left),
74                               local), linenum);
75

```

Figure 11 ECL Programming Example – Log File Analysis Macro

The example log file data contains lines of text which include a date, time, log source, message type, and additional log information formatted as key value pairs. An ECL macro (MAC\_Parse\_DTSM\_Keyval\_Format) was implemented for this specific type of log file format and is shown in Figure 11. The macro accepts parameters defining the input raw log file, the output formatted log file, and an output error file which will contain lines from the raw log file data which had an invalid format.

The steps used by the ECL macro shown in Figure 11 to process the raw log file data transforming the data to a formatted output file are as follows:

1. The raw input log file (inlogfile) is projected to a new format which adds a sequential line number in a separate field to each log line for reference in macro lines 5-13. Individual ECL statements are terminated by a semicolon character, and whitespace can be used freely to improve readability of the code.
2. NLP patterns are defined using the ECL PATTERN statement to represent the data to be extracted from the raw log lines in macro lines 15-20. Note references to other patterns such as Text.Date and Text.ISO\_Time which are shared pattern definitions stored in the Text module in the ECL repository.
3. The output record format for parsed log lines is shown in macro lines 22-30 and include separate fields for the date, time, log source, message type, and additional log information.
4. Parsing of the raw log data into the format described in step 3 is shown in macro line 33. This parse statement as well as other ECL statements operate on the entire file. Each node in a Thor processing cluster operates on the part of the file locally stored on the node.
5. The log\_info field parsed in the operation described in step 4 includes additional key-value pairs. This information is then parsed into a separate dataset in macro line 46, using pattern statements defined in macro lines 35-38, and the output record definition defined in macro lines 40-44.
6. The final formatted output from the log file is designed to include the fields data, time, log source, and message type, and a child dataset for each log line containing the key-value pairs extracted from the log\_info field. This output record format is defined in macro line 49 which references a separate ECL attribute containing the record definition stored in the ECL repository in the Log\_Analysis module named Layout\_DTSM\_Keyval which is shown in Figure 12.
7. The initially parsed log file from macro line 33 (log\_init) is projected to the output format in lines 51-55. To complete the output file, the key-value pairs for each log line generated in step 5 (keyvals\_init) are added to the initialized output file (log\_out\_init) using the ECL DENORMALIZE statement in macro lines 67-74. Both files are distributed across the available nodes in the cluster by log line number so this operation can be performed locally. The key-value pairs are sorted by the linenum and key fields and the final output is sorted in order by the linenum field.
8. Lines which had invalid formats which failed to parse properly are identified and written to a separate dataset in lines 57-64 using the ECL JOIN operation to join the initial sequenced log file (log\_seq) to the initial log data parse (log\_init) by the log line number (linenum). Lines which appear in the log\_seq file and not in the log\_init file are written to the error dataset. This is facilitated by a unique ECL JOIN option LEFT ONLY which generates records which appear in the left dataset of the join operation and not in the right dataset.

```

1  export Layout_Keyval := record, maxlength(4096)
2  string key;
3  string val;
4  end;
5
6  export Layout_DTSM_Keyval := record, maxlength(16384)
7  unsigned4 linenum;
8  string date;
9  string time;
10 string log_source;
11 string log_msg_type;
12 dataset(layout_keyval) keyvals;
13 end;
14

```

**Figure 12** ECL Programming Example – Log File Output Format

The MAC\_Parse\_DTSM\_Keyval\_Format ECL macro can now be used to process any raw log file with the defined format. An example of using this ECL macro is shown in Figure 13. This code can be executed from the QueryBuilder IDE as an ECL job. The code includes a dataset definition of the raw input log file (lines 1-7), an output statement to display a sample of the raw log data (line 10), a MAC\_Parse\_DTSM\_Keyval\_Format macro call to process the raw log data (line 13), an output statement to display a sample of invalid format raw log lines, and an output statement to display a sample of the processed log data. Figure 14 shows the job execution graph for the example job. Figure 15 shows a sample of the raw log file input data and the formatted log data output for the example job.

```
1 // Raw log data
2 Layout_Messages := record, maxlength(4096)
3 string line:
4 end:
5
6 File_Syslog_Raw := dataset('rhor400_06_dev::logs::sotn04_iptables_iptables syslog',
7     Layout_Messages, CSV(heading(0), separator(','), terminator('\n'), quote(''))):
8
9 //Output sample raw log data
10 output(File_Syslog_Raw):
11
12 // Parse fields from raw Syslog Log Files
13 Log_Analysis.MAC_Parse_DTSM_Keyval_Format(File_Syslog_Raw, log_out, log_errors):
14
15 // Output sample invalid lines if any
16 output(log_errors, named('Syslog_Invalid_Lines')):
17
18 // Output sample of processed log data
19 output(log_out, named('Syslog_Sample_Output')):
20
```

**Figure 13** ECL Programming Example – Log File Analysis Job



# ECL Programming Example — Log File Analysis Graph

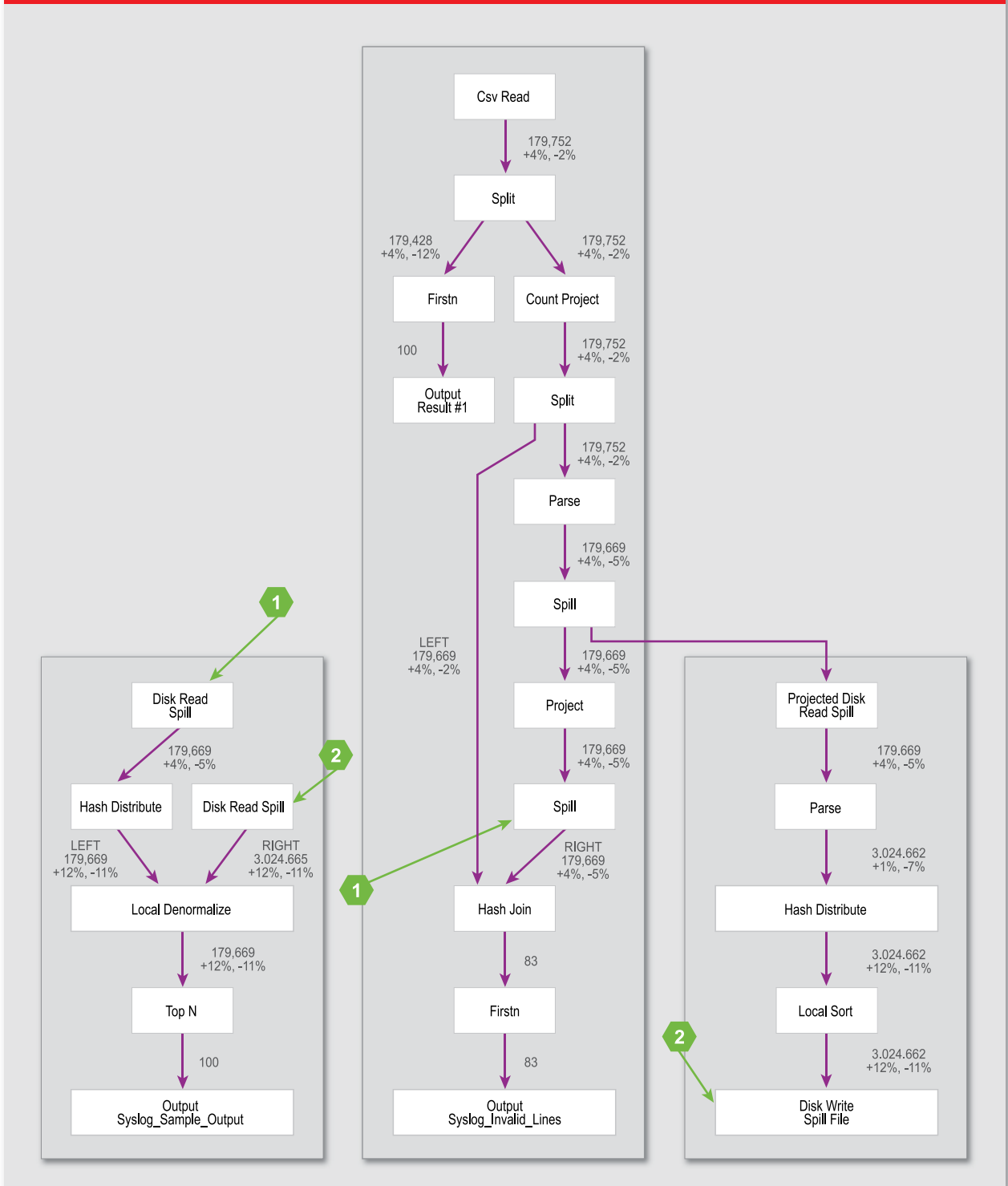


Figure 14

line	
1	Feb 25 12:11:24 bridge kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0 PHYSOUT=eth1 SRC=220.228.136.38 DST=11.11.79.83 LEN=64 TOS=0x00 PREC=0x00 TTL=47 ID=17159 DF PROTO=TCP SPT=1629 DPT=139 WINDOW=44620 RES=0x00 SYN URGP=0
2	Feb 25 12:11:27 bridge kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0 PHYSOUT=eth1 SRC=220.228.136.38 DST=11.11.79.83 LEN=64 TOS=0x00 PREC=0x00 TTL=47 ID=17800 DF PROTO=TCP SPT=1629 DPT=139 WINDOW=44620 RES=0x00 SYN URGP=0

	linenum	date	time	log source	log msg type	keyvals	
						key	val
1	1	Feb 25	12:11:24	bridge kernel	INBOUND TCP	DPT	139
						DST	11.11.79.83
						ID	17159
						IN	br0
						LEN	64
						OUT	br0
						PHYSIN	eth0
						PHYSOUT	eth1
						PREC	0x00
						PROTO	TCP
						RES	0x00
						SPT	1629
						SRC	220.228.136.38
						TOS	0x00
TTL	47						
URGP	0						
WINDOW	44620						
2	2	Feb 25	12:11:27	bridge kernel	INBOUND TCP	DPT	139
						DST	11.11.79.83
						ID	17800
						IN	br0
						LEN	64
						OUT	br0
						PHYSIN	eth0
						PHYSOUT	eth1
						PREC	0x00
						PROTO	TCP
						RES	0x00
						SPT	1629
						SRC	220.228.136.38
						TOS	0x00
TTL	47						
URGP	0						
WINDOW	44620						

Figure 15 ECL Programming Example – Log File Analysis Output

### *HPCC Middleware and System Servers*

An HPCC configuration includes a number of system servers which provide a gateway from the Thor and Roxie clusters to the outside world and also support services within an HPCC environment. These include the ECL Server, Dali Server, Sasha Server, DFU Server, and ESP Server and are referred to as the HPCC middleware components which are shown in Figure 8.

The ECL Server includes the ECL compiler and executable code generator, and functions as the job server for the Thor job execution environment. The ECL compiler translates the source ECL statements into executable C++ code in the form of dynamic link libraries (DLLs) that can be executed on Thor or Roxie clusters. When an ECL job (also referred to as a workunit) is submitted for execution on a Thor cluster, it is first converted to executable code by the ECL Server. For a Roxie cluster, this process occurs when a new ECL query is deployed and stored on a Roxie cluster which allows the query to be compiled once, but then executed multiple times as queries are received. ECL Server is also accessed when a syntax check is performed in the QueryBuilder IDE, and is responsible for starting an ECL Agent process whenever a job is executed. Multiple ECL servers can be configured in an HPCC environment which will automatically be load balanced to increase throughput.

The Dali Server functions as the system data store. It manages workunit data related to job execution, it maintains the logical file directory for the DFS functioning as the nameserver, and provides shared object services for execution of workunits. In addition it is used to configure the HPCC environment, maintain the message queues that implement job execution and scheduling, and enforces the LDAP security restrictions for data files and workunit user scopes.

The Sasha server functions as a companion “housekeeping” server to the Dali server and works independently of all other components and can be restarted without affecting current jobs in flight. Its main function is to reduce the stress and resource utilization on the Dali server whenever possible, and archives job execution workunits and DFU workunits which are stored in a series of folders, can be restored when needed, and can be manually moved to an alternate or off-site location. The Sasha server performs additional housekeeping functions including removal of cached workunits and DFU recovery files.

The DFU Server (distributed file utility) manages and controls the spraying and despraying operations that used to move files to and from the DFS in a Thor cluster. For each DFU operation a workunit, similar to an ECL job workunit, is created for managing and tracking the operation. DFU services can be accessed from the Querybuilder IDE or as part of a ECL job using common service libraries, using the ECL Watch utility program, or the DFU command line interface program.

The ESP Server (Enterprise Service Platform) is a communications server and customizable framework that provides communications interfaces and services to client applications and to the job execution and cluster environment. Protocols supported by the ESP server include HTTP, SOAP, and proprietary protocols. Standard services include WS\_Attribute a SOAP interface to the ECL repository; WS\_ECL which provides a form based Web interface to submit an ECL job on a Thor Cluster or to access a deployed query on the on a Roxie cluster; and ECL\_Watch, a Web-based query execution, monitoring, and file management interface that can be accessed from QueryBuilder or directly from a Web browser. Other ESP-based tools include RoxieConfig which provides a Web-interface for deploying managing deployed queries to a Roxie cluster with the ability to add, delete, suspend, un-suspend, provide alias names for queries, and provide access to statistics on executed queries. ESP can also include custom user-defined authentication, logging, billing, and audit services.

### *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 QueryBuilder, an integrated programming development environment (IDE) 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. Figure 16 shows the Query Builder IDE application.

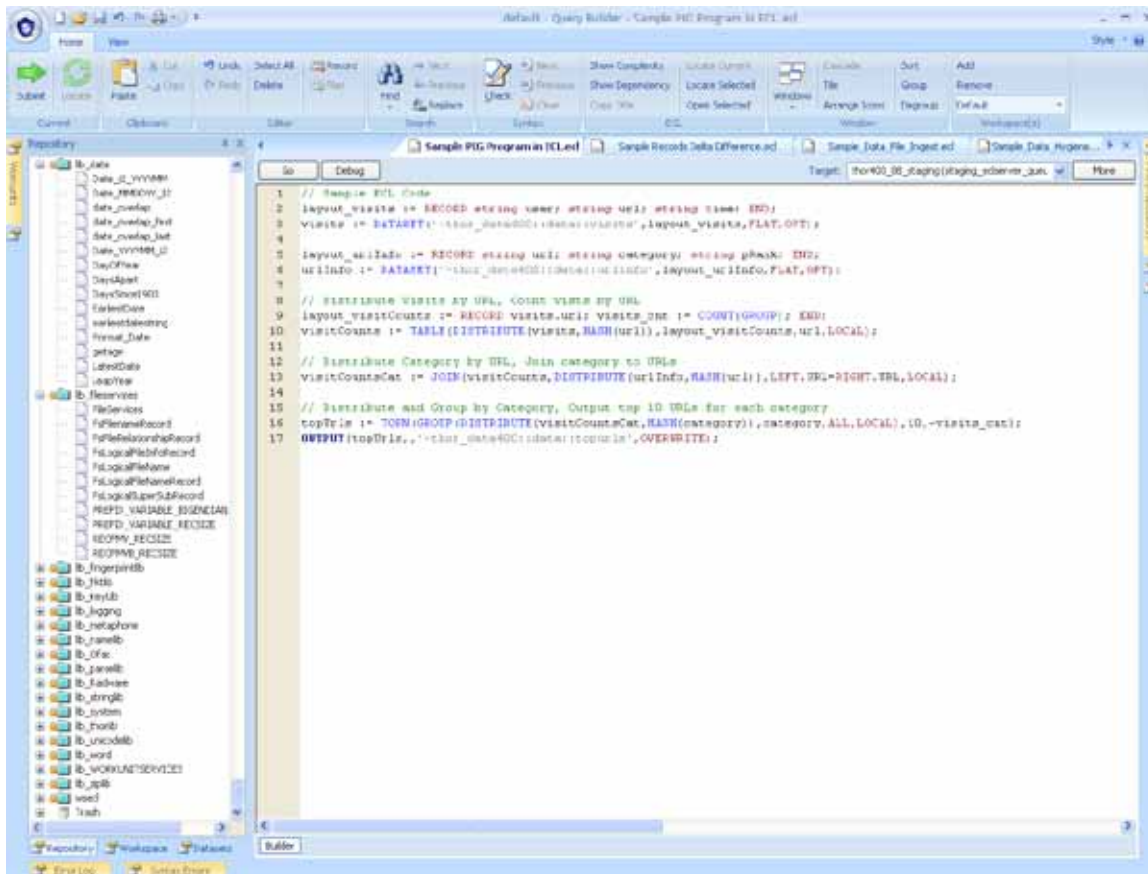


Figure 16 QueryBuilder IDE

QueryBuilder provides a full-featured Windows-based GUI for ECL program development and direct access to the ECL repository source code. QueryBuilder 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. QueryBuilder 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.

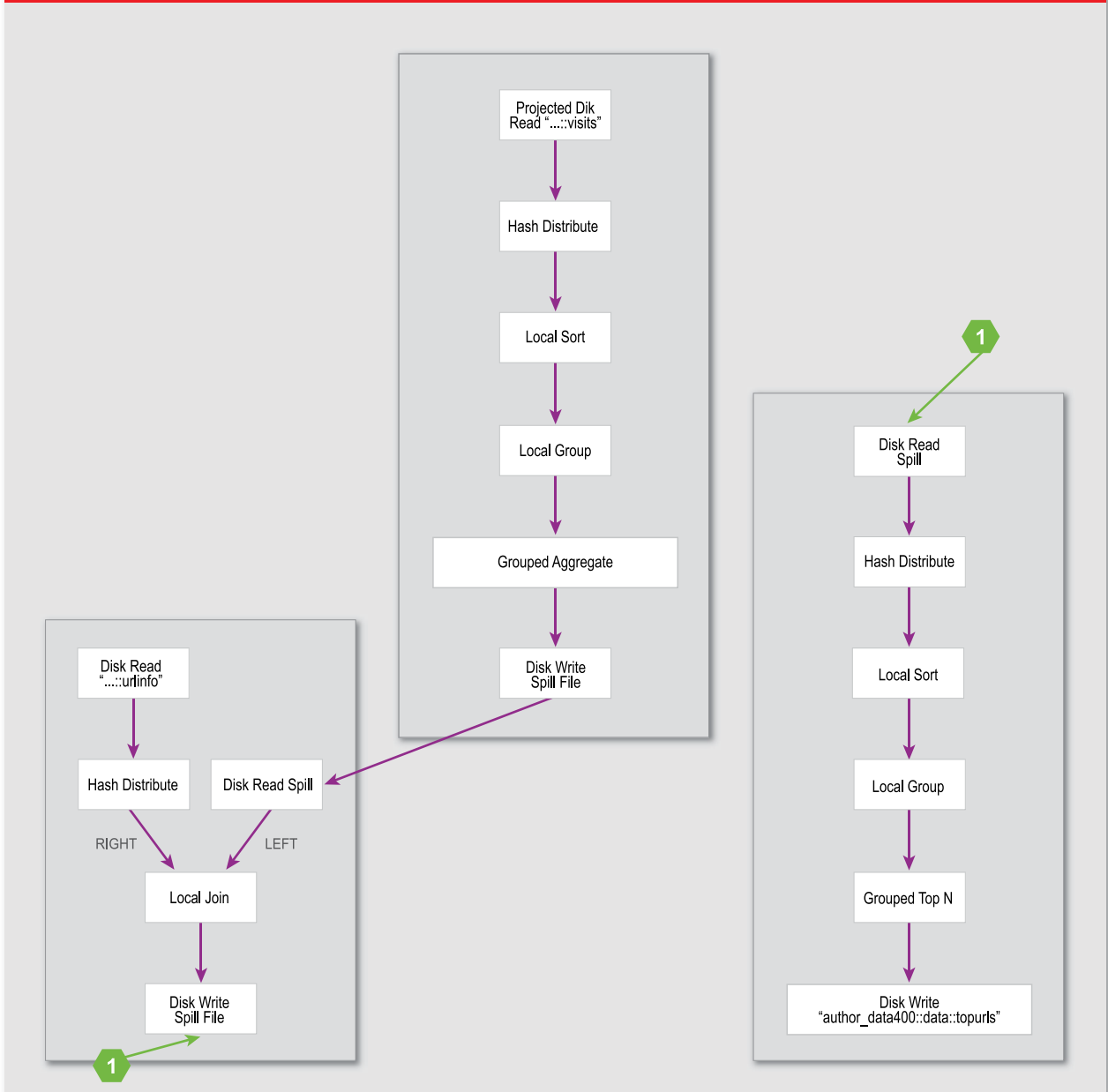
ECLWatch is a Web-based utility which uses the ESP server to provide a set of tools for monitoring and managing HPCC clusters which is shown in Figure 17. ECLWatch allows you see information about workunits including a graph displaying a visual representation of the dataflows for the workunit complete with statistics which are updated as the job progresses. The graph is interactive and you can drill down on nodes and connectors to see more detailed information and statistics. This information is retained in the workunit even after the job has completed so it can be reviewed and analyzed. An example of an ECL execution graph corresponding to the code example in Figure 10 is shown in Figure 18. In addition with ECLWatch, you can monitor cluster activity, browse through or search for previously submitted workunits, use DFU functions to search for files and see information including record counts and layouts and display data from the file, spray and despray files from available landing zones to and from clusters, check the status of all system servers, view log files, change job priorities, and much more. Figure 17 shows an example of the ECLWatch Web-interface for the HPCC environment.



Figure 17 ECLWatch Web-based Utility

The HPCC platform also provides an Attribute Migration Tool (AMT) which allows ECL source code to be copied from one ECL repository to another. For example, in most HPCC configurations there are separate development and production environments. AMT allows newly developed ECL attributes to be migrated from development to production in a controlled manner.

## ECL Watch Job Execution Graph



**Figure 18**

The HPCC platform also includes command line programs which can be used from the Windows command prompt or called from other programs. The ECLPlus application can access the ECL repository and accepts command line parameters to access the ECL repository and initiate ECL job execution. Commands can be typed directly on the command line, read from a batch file, initiated using an INI file, or any combination. The DFUPlus application accepts command line parameters or reads batch files to initiate distributed file utility functions such as spraying or despraying of files to and from clusters. A version of the AMT program is also provided in a command line interface version called AMTPlus. In addition, a command line version of the Roxie configuration utility RoxieConfig provided in the ESP services as a Web interface is available.

### *Using a Thor Cluster*

As described earlier, the Thor cluster is used as 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, record linking and entity resolution, large-scale ad-hoc complex analytics, and creation of keyed data and indexes to support high-performance structured queries and data warehouse applications. A Thor cluster includes a master node, and as many slave nodes as needed to satisfy the processing, data storage, and throughput requirements for a specific HPCC installation. For example, a training cluster might have only a few nodes, a Thor cluster for a small organization might have 20 nodes, and a Thor cluster for large organizations with terabytes of data to be processed might have hundreds of nodes. Multiple Thor clusters can be included in an HPCC environment and share the distributed filesystem storage available on each cluster and job scheduling and processing requirements.

Each Thor cluster also includes a distributed filesystem (DFS<sup>2</sup>), and each node of a Thor cluster is a data node in the file system as well as a job execution node for the parallel processing environment. Data is initially loaded to the Thor DFS using a process called spraying, in which data from a landing zone for external files is copied to the Thor cluster so that each node receives a segment of the file, initially divided as equally among the nodes as possible depending on the logical record structure of the file so that logical records are not split across nodes. Files in the DFS can be redistributed as needed during a processing sequence using the ECL programming language. For example, the logical records in a file can be distributed so that all the records with matching key fields are placed on the same node which insures that subsequent data processing operations such as project, sort, and join for matching records are localized to the node, and no additional inter-node movement of data is needed accomplishing the goal of data parallel operation and maximizing performance.

ETL (extract, transform, load) to process or refine raw data for some other purpose is a typical application performed on a Thor cluster. This type of application can be coded in ECL to execute as a single job on the Thor cluster encompassing may separate processing steps. This allows the ECL compiler to optimize the full processing sequence instead of just a single step. The Extract process may include projecting of source data fields to common record layouts used in the data; splitting or combining multiple source files, records, and fields to match the required layout; cleansing and standardization of data fields which will be used for searching such as name, address, identifiers, dates, etc.; and statistical and other types of analysis of the data to assess quality or to derive new information to be appended to the processed records. The Transform process can include combining multiple records into one (denormalize), or splitting single records into multiple or parent and child records (normalize); translation of codes into descriptions or vice-versa; standardizing names and addresses into separate parts in individual fields; validating and reformatting date fields; resolving and appending internal identifiers to people, businesses, and other entities, in order to link them across datasets; adding new records to an existing dataset, replacing or updating matching existing records; removing duplicates (dedup) or combining information with existing data (rollup); and linking or clustering records to each other if applicable. The Load Process includes building indexes or other data structures for use on a separate system such as a data warehouse or other independent query platform, an online analytical processing (OLAP) system, or a business intelligence system (BIS). In an HPCC environment, indexes are built which are subsequently deployed to a Roxie cluster to support online queries. Roxie indexes can contain both the searchable fields and other data fields from the base data referred to as the payload to improve query performance.

In summary, the steps for a typical process on a Thor cluster from spray to delivery of information are (1) spray the raw data to be processed to the Thor cluster, i.e. load the data to the Thor DFS; (2) perform the ETL process described previously to clean, standardize, and transform the data for its intended purpose updating any internal base files on the Thor cluster and building index files to be used with online queries developed for a Roxie cluster; (3) deploy the transformed data to the delivery system such as a Roxie cluster, or despray the data to a landing zone for transfer to an external system such as a traditional RDBMS, data warehouse, or other system platform. The transformed data can also be left on the Thor cluster to support additional entity resolution or complex ad-hoc analytical processing on large datasets.

*2) Each node in a Thor system serves an important dual-purpose, as a processing node for job execution, and as data storage for the DFS which can be accessed by processing nodes and other clusters.*



### *Using a Roxie Cluster*

As described previously, the Roxie cluster is designed as an online high-performance structured query and analysis engine or data warehouse supporting thousands of simultaneous queries and users with sub-second response times through Web services interfaces. The Roxie cluster uses a distributed indexed file system<sup>3</sup> and structured query programs written in the same ECL programming language used with Thor clusters. Query programs in ECL are pre-compiled and deployed to a Roxie cluster to facilitate high-performance execution, fast response times, and reusability. When a query is deployed to a Roxie cluster, the supporting data and index files are loaded into the Roxie distributed indexed file system which is independent from the DFS on the Thor cluster. Roxie clusters are designed to have high availability, data is redundantly stored on two or more nodes, and Roxie continues to operate seamlessly even if one or more nodes fail. Additional redundancy can be provided by including multiple Roxie clusters in an HPCC environment.

Roxie clusters are typically used for searching and other types of information retrieval and analysis applications using index files previously built on a Thor cluster. Multi-threading is used for efficient parallel multi-user retrieval of data. The Roxie distributed file system supports sophisticated multi-field index structures that can support range of value indices, phonetic keys, compound (multivariate keys), keys with data built from multiple data files, and keys that support full text ranked Boolean searches. Indexes can be specified to be memory-based to further support high-performance lookup or in cases where full scans are required.

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. Queries can access data files and index files referred to in the ECL code. Files can be accessed from a remote location which can be another Thor or Roxie cluster, by copying the files to the nodes of the local Roxie cluster when queries are deployed, and by using a remote copy until the local copy is complete.

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. 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. 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.

### *Thor and Roxie Together: A Complete Solution*

LexisNexis developers recognized that to meet all the requirements of data-intensive computing two distinct computing platforms were needed, one for processing large volumes of raw data which could also support complex ad-hoc analytical applications, and another to function as a high-performance search and structured query processing engine that could support thousands of users with sub-second access to information. LexisNexis also recognized the need for a new data-centric programming language for parallel data processing to significantly enhance programmer productivity and reduce programming complexity for parallel applications. The result was the HPCC platform, which integrates Thor

*3) The Roxie filesystem stores both indexes and data distributed across the nodes of the cluster to facilitate parallel high-performance online query processing.*



and Roxie clusters and the ECL programming language in a powerful, flexible, and easy to implement and use high-performance cluster computing environment. (Note: a Roxie is not required in an HPCC environment which can contain only Thor clusters, however to use a Roxie cluster, a Thor system is required for building and deploying index files to the Roxie cluster).

The high-level of integration in the HPCC platform provides a distinct advantage over competing technology such as Hadoop which utilizes the MapReduce processing approach and bolt-on systems to provide a complete parallel processing solution. This is evident in the significantly better performance of the HPCC platform based on standard benchmark results using the same hardware platform. This performance advantage results because Thor and Roxie clusters and their filesystems are each individually optimized for their specific parallel processing purpose, and ECL batch job execution and online query execution are optimized as a whole process end-to-end, instead of sequencing or chaining individual MapReduce steps. The power, flexibility, advanced capabilities, speed of development, and ease of use of the ECL programming language and seamless integration across HPCC systems is also an important distinguishing factor between the LexisNexis HPCC platform and other data-intensive computing solutions. The HPCC platform provides a complete high-performance integrated parallel processing solution from raw data to useful information.

## HPCC Performance

A standard benchmark available for data-intensive computing platforms is the Terasort benchmark managed by an industry group led by Microsoft and HP. 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 a system. The Terabyte sort has since 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 various systems, it is useful to run benchmarks on identical system hardware configurations. A head-to-head comparison of the HPCC platform to Hadoop using the original Terabyte sort on a 400-node cluster is presented here.

### *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 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 original 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: Daytona (a standard commercial computer system and software with no modifications) and Indy (a custom computer system with any type of modification). No restrictions exist on the size of the system so the sorting benchmark could be conducted on as large a system as desired. The 2009 record holder for the Daytona category is Yahoo! using a Hadoop configuration with 1460 nodes with 8GB Ram per node, 8000 Map tasks, and 2700 Reduce tasks which sorted 1 TB in 62 seconds. 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 with competitive systems on the same hardware configuration. Other factors should also be evaluated such as the amount of code required to perform the benchmark which provides a strong indication of programmer productivity, and is a significant performance factor in the implementation of parallel computing applications.



Figure 19 Hadoop Terabyte Sort Benchmark Results.

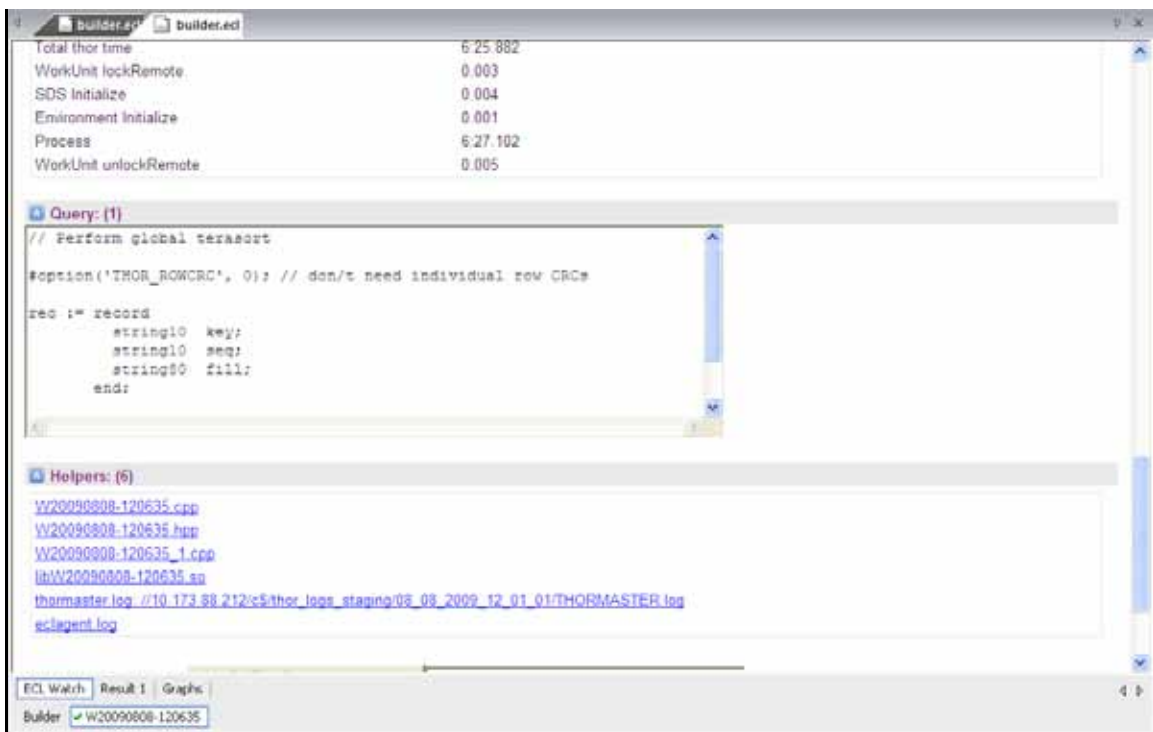


Figure 20 HPCC Terabyte Sort Benchmark Results.

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 300MB SCSI disk drives, Intel Xeon single core processors running at 3.00 GHz, 4GB 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 Figure 19. 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 Figure 20. 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.

## Conclusions

As a result of the continuing information explosion, many organizations are experiencing a significant “data gap” or inability to process this information and use it effectively. High-performance data-intensive computing with commodity computing clusters represents a new approach which can address the data gap and allow government and commercial organizations and research environments to process massive amounts of data and implement new applications previously thought to be impractical or infeasible. Technology solutions such as Hadoop MapReduce and the HPCC platform from LexisNexis are now available which offer data parallel processing capability on low-cost commodity computing clusters.

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 such as Hadoop while others prefer a commercially developed and supported platform by an established industry leader. The Hadoop MapReduce platform is being used successfully at many 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 National Research Laboratories because of its higher-performance cost-effective implementation. Existing HPCC applications include raw data processing, ETL, 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 resolution of unstructured and semi-structured data such as Web documents; statistical analysis of Web logs for security applications such as intrusion detection; online analytical processing to support business intelligence systems; 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 addition to suitability, flexibility, scalability, footprint, and power consumption factors which impact the total cost of ownership (TCO) must be considered.

A performance comparison of the Hadoop MapReduce and the HPCC platform using the Terabyte sort benchmark in this paper reveals a significant performance advantage for the HPCC platform on identical hardware configurations. Other advantages of selecting the LexisNexis HPCC platform for data-intensive computing include: (1) a highly integrated system environment with capabilities from raw data processing to high-performance queries and data analysis using a common language; (2) a cluster approach which provides high performance at a much lower system cost than other system alternatives resulting in significantly lower total cost of ownership (TCO); (3) a stable and reliable processing environment proven in production applications for varied organizations over a 10-year period; (4) an innovative data-centric programming language (ECL) with extensive built-in capabilities for data-parallel processing, significantly increasing programmer productivity for application development, which automatically optimizes execution graphs with hundreds of processing steps into single efficient workunits; (5) a high-level of fault resilience and capabilities which reduce the need for re-processing in case of system failures; (6) suitability for a wide range of data-intensive applications from large volume ETL processing to support databases, data warehouses, and high volume online applications to network security analysis of massive amounts of log information; and (7) available from and supported by a well-known leader in information services and “large data” solutions (LexisNexis) which is part of one of the world’s largest publishers of information – ReedElsevier.

## Glossary

<b>AMT</b>	Attribute Migration Tool. Allows ECL source code to be copied from one ECL repository to another within HPCC system environments.
<b>BORPS</b>	Billions of records per second. A term invented by LexisNexis to describe the processing capabilities of its HPCC platform.
<b>Computing Cluster</b>	A group of shared individual computers, linked by high-speed communications in a local area network topology using technology such as gigabit network switches, and incorporating system software which provides an integrated parallel processing environment for applications with the capability to divide processing among the nodes in the cluster.
<b>COTS</b>	Commodity off the shelf. Used to describe commodity hardware (personal computers, disks, network) that can be purchased from multiple sources.
<b>Dali Server</b>	Functions as the system data store in the HPCC system environment. Manages workunit data related to job execution, maintains the logical file directory for the distributed file system, and provides shared object services for execution of workunits.
<b>DAS</b>	Data Analytic Supercomputer. An alternate name for the HPCC Platform.
<b>Data-Intensive Computing</b>	Used to describe computing applications that are I/O bound or with a need to process large volumes of data. Such applications devote most of their processing time to I/O and movement of data.
<b>Data parallel</b>	A parallel processing approach where computation is 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.
<b>DFU Server</b>	Distributed File Utility. A server in the HPCC system environment that manages and controls the spraying and despraying operations that used to move files to and from the DFS in a Thor cluster and other DFS operations.
<b>ECL</b>	Enterprise Data Control Language. A high-level parallel programming language used on the HPCC platform for data-intensive computing applications.
<b>ECL Server</b>	Includes the ECL compiler and executable code generator, and functions as the job server for Thor job execution in the HPCC system environment. The ECL compiler translates the source ECL statements into executable C++ code in the form of dynamic link libraries (DLLs) that can be executed on Thor or Roxie clusters
<b>ECLWatch</b>	A Web-based utility which uses the ESP server to provide a set of tools for monitoring and managing HPCC clusters. ECLWatch allows you see information about workunits including a graph displaying a visual representation of the dataflows for the workunit complete with statistics which are updated as the job progresses.
<b>ESP Server</b>	Enterprise Service Platform. A communications server and customizable framework in the HPCC system environment that provides communications interfaces and services to client applications and to the job execution and cluster environment. Protocols supported by the ESP server include HTTP, SOAP, and proprietary protocols.
<b>ETL</b>	Extract, transform, load. An industry standard acronym for the process of reading data from an external file, cleansing and converting the data into the form it needs to be, and loading the data into an internal database.

<b>Hadoop</b>	An open source software project initiated to create an open source implementation of the MapReduce architecture
<b>HPC</b>	High-Performance Computing. Describes computing environments which utilize supercomputers and computer clusters to address complex computational requirements, support applications with significant processing time requirements, or require processing of significant amounts of data
<b>HPCC</b>	High-Performance Computing Cluster. The LexisNexis data-intensive computing platform.
<b>MapReduce</b>	A programming model that allows group aggregations in parallel over a cluster of machines. 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.
<b>NLP</b>	Natural Language Processing. Processing of natural language in machine-readable form such as text by a computer system for a wide variety of applications.
<b>QueryBuilder</b>	An interactive development environment (IDE) for the ECL programming language. Provides a full-featured GUI for ECL program development and direct access to the ECL repository source code
<b>ROXIE</b>	The rapid data delivery system for online query processing in the HPCC platform. Acronym for Rapid Online XML Inquiry Engine.
<b>Sasha Server</b>	A companion “housekeeping” server to the Dali server in the HPCC system environment. Archives job execution workunits and DFU workunits which are stored in a series of folders, which can be restored when needed, and can be manually moved to an alternate or off-site location. Provides additional housekeeping functions including removal of cached workunits and DFU recovery files.
<b>Seisint</b>	Refers to Seisint, Inc., the original developer of the HPCC data supercomputer technology which was acquired by LexisNexis in 2004.
<b>THOR</b>	The data refinery system in the HPCC platform. A batch job processing environment used for ETL and other data-intensive computing applications.
<b>Workunit</b>	A job in the HPCC environment. Encapsulates all information related to a job. For ECL job execution, includes input file information, results, timings, graphs, and ECL code and helper files including the C++ code generated and system logs for the job.
<b>XML</b>	Extensible Markup Language. An industry open standard for describing and formatting data. Provides a flexible way to create common information formats and share both the format and the data on the Web.

## References

1. J. Dean, and S. Ghemawat, "MapReduce: A Flexible Data Processing Tool," *Communications of the ACM*, Vol. 53, No. 1, 2010, pp. 72-77.
2. R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," *Proceedings of the VLDB Endowment*, 2008.
3. M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and Parallel DBMSs: Friends or Foes?" *Communications of the ACM*, Vol. 53, No. 1, 2010, pp. 64-71.
4. J. Dean, and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.
5. J. Venner, "Pro Hadoop," Apress, 2009.
6. T. White, "Hadoop: The Definitive Guide," O'Reilly Media Inc., 2009.
7. R. Grossman, and Y. Gu. "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*, 2008.
8. K. Dowd, and C. Severance, "High Performance Computing," O'Reilly and Associates, Inc., 1998.
9. R. Buyya, "High Performance Cluster Computing," Prentice Hall, 1999.
10. L.S. Nyland, J.F. Prins, A. Goldberg, and P.H. Mills, "A Design Methodology for Data-Parallel Applications," *IEEE Transactions on Software Engineering*, Vol. 26, No. 4, 2000, pp. 293-314.
11. D. Ravichandran, P. Pantel, and E. Hovy. "The Terascale Challenge," *Proceedings of the KDD Workshop on Mining for and from the Semantic Web*, 2004.
12. O. O'Malley. "Introduction to Hadoop," 2008, Available from: <http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf>.
13. C. Bookman, "Linux Clustering: Building and Maintaining Linux Clusters," New Riders Publishing, 2003.
14. J.D. Sloan, "High Performance Linux Clusters," O'Reilly Media Inc., 2005.
15. I. Gorton, P. Greenfield, A. Szalay, and R. Williams, "Data-Intensive Computing in the 21st Century," *IEEE Computer*, Vol. 41, No. 4, 2008, pp. 30-32.
16. W.E. Johnston, "High-Speed, Wide Area, Data Intensive Computing: A Ten Year Retrospective," *IEEE Computer Society*, 1998.
17. M. Gokhale, J. Cohen, A. Yoo, and W.M. Miller, "Hardware Technologies for High-Performance Data-Intensive Computing," *IEEE Computer*, Vol. 41, No. 4, 2008, pp. 60-68.
18. R.E. Bryant. "Data Intensive Scalable Computing," 2008, Available from: <http://www.cs.cmu.edu/~bryant/presentations/DISC-concept.ppt>.
19. A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. Dewitt, S. Madden, and M. Stonebraker. "A Comparison of Approaches to Large-Scale Data Analysis," *Proceedings of the 35th SIGMOD international conference on Management of data*, 2009.
20. J.F. Gantz, D. Reinsel, C. Chute, W. Schlichting, J. McArthur, S. Minton, J. Xheneti, A. Toncheva, and A. Manfrediz, "The Expanding Digital Universe," IDC, White Paper, 2007.
21. S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System," *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

22. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," *Scientific Programming Journal*, Vol. 13, No. 4, 2004, pp. 227-298.
23. T. White. "Understanding MapReduce with Hadoop," 2008, Available from: <http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/MapReduce-SPA2008.pdf>.
24. D. Borthakur. "Hadoop Distributed File System," 2008, Available from: [http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/hdfs\\_dhruba.pdf](http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/hdfs_dhruba.pdf).
25. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. "Pig Latin: A Not-So-Foreign Language for Data Processing (Presentation at SIGMOD 2008)," 2008, Available from: <http://i.stanford.edu/~usriv/talks/sigmod08-pig-latin.ppt#283,18,User-Code> as a First-Class Citizen.
26. A.F. Gates, O. Natkovich, S. Chopra, P. Kamath, S.M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. "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)*, 2009.
27. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. "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*, 2008.
28. F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. "Bigtable: A Distributed Storage System for Structured Data," *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.
29. A.M. Middleton, "Data-Intensive Computing Solutions," LexisNexis, Whitepaper, 2009.



**For more information:**

**Website: <http://hpccsystems.com/>**

**Email: [info@hpccsystems.com](mailto:info@hpccsystems.com)**

**US inquiries: 1.877.316.9669**

**International inquiries: 1.678.694.2200**

**About HPCC Systems**

HPCC Systems from LexisNexis® Risk Solutions offers a proven, data-intensive supercomputing platform designed for the enterprise to solve big data problems. As an alternative to Hadoop, HPCC Systems offers a consistent data-centric programming language, two processing platforms and a single architecture for efficient processing. Customers, such as financial institutions, insurance carriers, insurance companies, law enforcement agencies, federal government and other enterprise-class organizations leverage the HPCC Systems technology through LexisNexis® products and services. For more information, visit <http://hpccsystems.com>.

**About LexisNexis Risk Solutions**

LexisNexis® Risk Solutions (<http://lexisnexis.com/risk/>) is a leader in providing essential information that helps customers across all industries and government predict, assess and manage risk. Combining cutting-edge technology, unique data and advanced scoring analytics, Risk Solutions provides products and services that address evolving client needs in the risk sector while upholding the highest standards of security and privacy. LexisNexis Risk Solutions is headquartered in Alpharetta, Georgia, United States.

