Invited Paper

E^3: an Elastic Execution Engine for Scalable Data Processing

Gang Chen[1] Ke Chen[1] Dawei Jiang[2] Beng Chin Ooi[2] Lei Shi[2]
Hoang Tam Vo[2,a)] Sai Wu[2]

Received: July 8, 2011, Accepted: October 17, 2011

Abstract: With the unprecedented growth of data generated by mankind nowadays, it has become critical to develop efficient techniques for processing these massive data sets. To tackle such challenges, analytical data processing systems must be extremely efficient, scalable, and flexible as well as economically effective. Recently, Hadoop, an open-source implementation of MapReduce, has gained interests as a promising big data processing system. Although Hadoop offers the desired flexibility and scalability, its performance has been noted to be suboptimal when it is used to process complex analytical tasks. This paper presents E^3, an elastic and efficient execution engine for scalable data processing. E^3 adopts a "middle" approach between MapReduce and Dryad in that E^3 has a simpler communication model than Dryad yet it can support multi-stages job better than MapReduce. E^3 avoids reprocessing intermediate results by adopting a stage-based evaluation strategy and collocating data and user-defined (map or reduce) functions into independent processing units for parallel execution. Furthermore, E^3 supports block-level indexes, and built-in functions for specifying and optimizing data processing flows. Benchmarking on an in-house cluster shows that E^3 achieves significantly better performance than Hadoop, or put it another way, building an elastically scalable and efficient data processing system is possible.

Keywords: parallel processing, elastic exection engine, cloud computing

1. Introduction

In the "information explosion era" nowadays, the data created by mankind has increased rapidly. While data has become an important commodity in modern business where data analysis facilitates better business strategizing, the decision making in business is currently facing the Big Data challenge: the volume of data is growing at unprecedented pace; the data to analyze are diverse in types such as structured, semi-structured, and unstructured; and analytical applications are becoming more and more complicated. To tackle such challenges, analytical data processing systems must be extremely efficient, scalable, and flexible as well as economically effective.

A traditional method of managing large-scale data is through the use of parallel database management systems (PDMSes), which have their roots from the late of 1980s with pioneer Gamma [5] and Grace [9] projects. The parallel database technology offered by vendors such as Teradata, Netezza and Vertica, is typically a small or medium-size clustered deployment of a database management system that provides an environment for users to perform an analytical query via internal support of parallel query processing.

However, as the company's business grows, it needs to upgrade its hardware capacity on a frequent basis in order to accommodate the increasing workload, which presents many challenges in terms of both technical support and cost. Therefore, the "cloud computing" revolution, in which large clusters of commodity servers are exploited to perform various computing tasks with a "pay-as-you-go" model, becomes a feasible solution that mitigates the pain. Unfortunately, the approaches adopted by most traditional PDMSes cannot be directly applied to cloud data managements mainly due to the elasticity issue of the new environment. In the cloud, a large number of low-end machines, instead of a smaller number of high-end machines, are deployed to process massive datasets, and more importantly, the demand for resources may vary drastically from time to time due to the changes in the application workload. As a consequence, PDMSes may not be able to take full advantage of the cloud since users may desire to elastically allocate resources from the cloud based on the load characteristics while PDMSes are mainly designed and optimized for fairly static or fixed-size clusters.

Hadoop [*1], an open-source implementation of MapReduce [4], has recently become a popular tool for processing massive-scale data analytical tasks. By providing a data-parallel programming model, Hadoop can control the job execution in many advantageous ways: automatic division of job into tasks, automatic placement of computation near data, automatic load balancing, recovery from failures and stragglers, and most importantly, elastic scalability from tens to thousands of machines. Furthermore, users only need to focus on the application logic rather than the complexities of parallel computing.

[1] College of Computer Science, Zhejiang University, China
[2] School of Computing, National University of Singapore, Singapore
[a)] voht@comp.nus.edu.sg

[*1] http://hadoop.apache.org/

Nevertheless, although Hadoop provides the desired flexibility and scalability, its performance is not satisfactory when it is used to process complex analytical tasks since those tasks are evaluated as a MapReduce chain which requires to re-scan and re-process intermediate results during data processing. Furthermore, for some application scenarios, the one-input two-phase data flow of MapReduce is relatively rigid and hard to adapt. Hadoop does not allow for stateful multiple-step processing of records and the opaque nature of the map and reduce functions impedes most of query optimization techniques.

In our recent study [15], we identified the design factors (e.g., I/O modes, indexing utilization, record parsing scheme, grouping algorithm and scheduling policy) that affect the performance of Hadoop and investigated alternative methods for each factor. Based on these findings, this paper presents E^3, an elastic and efficient execution engine for scalable data processing. The key idea of improving performance is that E^3 avoids reprocessing intermediate results by adopting a stage-based evaluation strategy and collocating data and user-defined functions (map or reduce) into independent processing units for parallel execution. One advantage of this approach is the efficient processing of join analytical queries in which E^3 joins multiple datasets in one go and thus avoids frequent checkpointing and shuffling of intermediate results, which is a major performance bottleneck in most of the current MapReduce based systems. In addition, E^3 also utilizes indexes for speeding up query processing and provides built-in support for specifying and optimizing data processing flows. The results of benchmarking our system against Hadoop on an in-house cluster confirm the superior performance of E^3 over Hadoop, or put it another way, building an elastically scalable and efficient processing system is possible.

This paper proceeds as follows. The following section presents related work. Section 3 discusses the desired properties of a scalable data processing system. We describe the design of E^3 in Section 4 and present its performance in Section 5. Section 6 concludes the paper.

2. Related Work

Systems for large-scale data processing can be classified into four groups, including parallel database systems, MapReduce [4] based systems, DAG-based systems, and parallel computing systems.

Parallel Database Systems. The research on parallel databases started in late 1980s [6]. Pioneering research systems include Gamma [5], and Grace [9]. Parallel database systems are mainly designed for processing relational data in a fairly static cluster environment while the design of MapReduce is driven by the increasing demand of processing Big Data with diverse data types in a large-scale dynamic cluster environment. Comparisons between parallel databases and MapReduce are presented in Refs. [19] and [23], which shows that the main differences between the two systems are performance and scalability.

Although parallel database systems can be deployed in cloud environment, they are not able to exploit the built-in elasticity feature which is important for startups, small and medium sized businesses. Parallel database systems are mainly designed and optimized for a cluster with a fixed or fairly static number of nodes and the inflexibility for growing up and shrinking down clusters on the fly based on load characteristics limits their elasticity and suitability for pay-as-you-go model.

Fault tolerance is another issue of parallel database systems in the new environment. Historically, it is assumed that node failures are uncommon in small clusters, and therefore fault tolerance is often provided for transactions only. The entire query must be restarted when a node fails during the query execution. This strategy may cause parallel database systems not being able to process long running queries on clusters with thousands of nodes, since in these clusters hardware failures are common rather than exceptional.

However, many design principles of parallel database systems such as indexing techniques, horizontal data partitioning, partitioned execution, cost-based query processing and declarative query support, could form the foundation for the design and optimization of systems to be deployed in the cloud.

MapReduce-based Systems. MapReduce is first introduced by Dean and Ghemawat [4] for simplifying the construction of web-scale inverted indexes. The framework is then also employed to perform filtering-aggregation data analysis tasks [20]. It is possible to evaluate more complex data analytical tasks as well, by executing a chain of MapReduce jobs [4], [19].

MapReduce systems have considerable advantages over parallel database systems. First, MapReduce is a pure data processing engine and is independent of the underlying storage system. Consequently, MapReduce and the storage system are able to scale independently, which makes this approach go well with the pay-as-you-go model. Second, map tasks and reduce tasks are assigned to available nodes on demand and users can dynamically increase or decrease the size of the cluster without interrupting the running jobs. Third, map tasks and reduce tasks are independently executed from each other, enabling MapReduce to be highly resilient to node failures. When a single node fails during job execution, only map tasks and/or reduce tasks on the failed node need to be restarted, but not the whole job.

Nevertheless, the performance of Hadoop, an open-source implementation of MapReduce, has been noted to be suboptimal in database context [19]. In our recent study [15], we have identified five design factors that affect the performance of Hadoop, including I/O modes, indexing utilization, record parsing, grouping algorithm and scheduling policy, and investigated alternative methods for each factor. In this paper, we introduce E^3 – an elastic and efficient execution engine for scalable data processing – based on these findings.

We note that there are also a number of analytical query processing systems that are built on top of Hadoop such as Pig [18] and Hive [26]. These systems provide a high-level query language and associated optimizer for efficient evaluating complex analytical queries. Compared to these systems, E^3 utilizes indexes for speeding up query processing and provides built-in support for specifying and optimizing data processing flows. Therefore, E^3 can be used as a new building block for these systems to generate an efficient query plan.

DAG-based Systems. When developing applications with

Hadoop, users can only express the application logic in two simple steps – map and reduce. However, to support more complex applications, developers have to manually cascade a sequence of MapReduce steps, which can significantly degrade the performance due to the considerable cost of materializing intermediate results and scanning these data repeatedly.

On the contrary, in Dryad [13], an analytical job is structured as a data flow directed acyclic graph (DAG) in which vertices are computations, i.e., executable programs, while edges are communication channels which could be files, TCP pipes, and shared-memory FIFOs. Each vertex can have several input and output channels. Therefore, Dryad subsumes other computation frameworks, such as MapReduce or the relational algebra. As an execution engine, Dryad runs the job by executing the vertices of the graph on a set of available machines in the cluster. Like Hadoop and E^3, Dryad also handles job creation and management, resource management, job scheduling and monitoring, fault tolerance and re-execution. Especially, the graph can be refined during execution via the just-in-time planning and dynamic optimization techniques.

Overall, Dryad facilitates users more in developing applications than MapReduce does since the mapping from algorithms to implementation is simpler in Dryad. However, the trade-off is that the interface of Dryad is more complex, and building DAGs (especially setting the communication channels between vertices in a large graph) by handed code could be a tedious task.

Clustera [7], an integrated computation and data management system, shares similar goals and designs with Dryad in many ways. Both are targeted toward handling a wide range of applications ranging from single process, computationally intensive jobs to parallel SQL queries. The two systems also adopt the same way of structuring a distributed computation as a DAG. However, Clustera adopts different implementation strategies, i.e., exploiting modern software building blocks such as application servers and relational database systems, in order to provide system performance, scalability and portability.

Parallel Computing Systems. These systems are mainly developed on high-performance computing platforms such as MPI [10], PVM [24] or computing on GPUs [25] to solve large and complex problems, typically for scientific computations. However, most of these models only provide primitives for the key operations such as point-to-point communications, collective operations, process groups, communication contexts, process topologies, and datatype manipulation. These primitives are too low-level and developing programs using these models requires developers to handle the synchronization of parallel executions on their own, which is not as convenient as MapReduce, Dryad or E^3 where all the burdens of synchronization are automatically handled by the system.

E^3 vs. MapReduce vs. Dryad. E^3 shares the same design philosophy with MapReduce and Dryad. These three systems all provide a programming model for users to write data analytical programs that process a subset of the dataset and a communication mechanism for coordinating user specified programs for arbitrary complicated computations. The differences between these systems lie in the flexibility and efficiency of the programming/communication model that each system provides.

Compared to MapReduce and Dryad, E^3's programming model is much richer. The programming model introduces `InputIterator` and `ProcessingUnit` interface to abstract the data access method and the data manipulation program. With proper implementations, E^3 can support various data access methods including sequential scan, B^+-tree, and hash lookup. Furthermore, users can implement multi-pass algorithms in a single `ProcessingUnit`. MapReduce and Dryad, on the other hand, are mainly designed for developing a single pass data processing program (each task only involves a single `map()` and `reduce()` function) by the sequential data access method.

The communication model of E^3 is also richer than MapReduce but less flexible than Dryad. MapReduce has no built-in facilities for users to coordinate their programs as a data flow. E^3 and Dryad, on the other hand, provides such support. Both systems support users to connect their programs as a DAG graph. However, unlike Dryad, E^3 does not allow users to fine-tune the communication channels between tasks. E^3 only supports one communication method, i.e., TCP and does not change the communication graph at runtime.

E^3 as part of epiC Cloud Data Management System. E^3 is part of our bigger system named epiC [*2] – an elastic power-aware data-intensive Cloud computing platform – for providing scalable database services in the cloud. In epiC, two typical workloads inclusive of data intensive analytical jobs (OLAP) and online transactions (OLTP) are supported to simultaneously and interactively run within the same storage and processing system.

The overall architecture of epiC cloud data management system is shown in **Fig. 1**. The system consists of the following main modules: Query Interface, OLAP/OLTP Controller, the Elastic Execution Engine (E^3) and the Elastic Storage System [1], [27] (ES^2). Here we will briefly introduce these modules. The details of how these modules work together in a cohesive system are described elsewhere [2].

The Query Interface provides a SQL-like language for up-level applications and compiles an input SQL query into a series of read and write operations (for OLTP query) or a set of analytical jobs (for OLAP query), which will be handled by the OLTP and OLAP Controller [30] respectively. E^3, a sub-system of epiC that is presented in this paper, is designed to efficiently perform

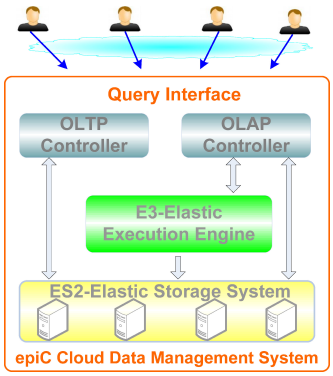


Fig. 1 E^3 as part of epiC cloud data management system.

[*2] http://www.comp.nus.edu.sg/~epiC/

large scale analytical jobs in the cloud. ES^2, the underlying cloud data storage system for supporting both OLAP and OLTP workloads, provides data access interfaces for upper-layer query processing engines, i.e., the OLAP/OLTP Controller. ES^2 develops a generic indexing framework [3] for declaration of various types of distributed indexes (e.g., hash indexes [22], B$^+$-tree-like indexes [29], and R-tree-like indexes [28]) over the cloud data in order to facilitate efficient processing of ad-hoc queries. The main idea of these indexes is to use P2P structured overlays (such as Chord [22], BATON [14] and CAN [21]) as global indexes and combine with local disk-resident indexes at each index node. This strategy is more efficient than the IR-based strategies in integrating distributed independent databases over unstructured network that are proposed in Ref. [17].

3. Desired Properties of a Scalable Data Processing System

This section presents the desired properties of a scalable data processing system which is designed for performing data analysis on Big Data.

Handling Data Diversity One of the major characteristic of Big Data is the diversity of data type. Traditionally, the data that the business produces are transactional data. These kinds of data are structured and often modeled as relational tables. However, the emergence of Web 2.0 has changed the situation significantly. Nowadays, business produces not only structured data but also semi-structured and non-structured data. This is because Web 2.0 enables businesses to build a much closer relationship with their customers than traditional technology (e.g, phone-call or periodical customer meeting) does. By posting product information on social website like Facebook or Twitter, businesses can drive more interactions with their customers and thus get more user feedback. These user-generated data constitute a large part of Big Data.

The ability to analyze data of various types poses a challenge to a data processing system. This "*cross-data*" analytical ability enforces the data processing system to adopt a generic data model instead of the well-known relational data model. As demonstrated by Google's MapReduce system, key-value data model tends to be appropriate for Big Data analysis.

Query Interface A typical data analytical task is performed in two consecutive steps. In the first step, the task retrieves some data from the underline storage systems (e.g., RDBMS or a file system). In the second step, a user-specified program is launched on those data retrieved and completes the analysis. The first step is called data retrieval. Traditionally, the data retrieval step is performed by emitting a SQL query to the underline database through JDBC or ODBC interface. Unfortunately, such an interface is only designed for structured data and particularly for relational databases. The JDBC or ODBC interface along with SQL is not appropriate for Big Data analysis as Big Data is diverse and is not always stored in a relational database. A solution to this problem, adopted by MapReduce, is to introduce a `Reader` interface to retrieve data from storage systems. To be compatible with different storage systems, the `Reader` interface should be generic. However, current `Reader`'s interface only supports sequential data access. Specific data accessing method such as indexing (e.g., B$^+$ tree) is not supported at the interface level, users must use ad hoc ways to utilize those data accessing methods. An ideal data retrieval interface design should cover as much data accessing method as possible without loss of generality.

Data Flow Complicated data analytical tasks are often represented as a data flow where a set of programs or functions (each of them performing part of the computation) communicate with each other to complete the task. For example, a typical relational database system evaluates a SQL query by first building a query tree where each node (called operator) performs certain computation logic, and then processes the data from leave nodes to the top node.

Big Data analysis requires the data processing system to offer a flexible way for the users to build the data processing flow. The system must provide a robust mechanism for the users to write their very own data processing programs and connect those programs as a data flow for automatically parallel execution. The mechanism must be general enough and not be bound to any specific domain. The major drawback of the relational database approach is that the data processing flow produced by the optimizer can only be used to evaluate relational queries and can not be used to perform other kinds of data analysis.

Performance A Big Data analytical system should perform data analytical tasks as efficiently as possible. This requires the data processing system to efficiently invoke user specified programs, efficiently execute data flow, and use the right technique to serialize and de-serialize data between user programs. Unfortunately, recent studies [15], [19] show that high performance is one of the missing features of MapReduce.

Fault Tolerance Big Data analytical tasks are often long-running queries. To complete the data analysis in reasonable time, the task is always performed on a large shared-nothing cluster which consists of hundreds or even thousands of nodes. At this scale, transient or permanent hardware failures during data processing are not uncommon. Thus, the data processing system must provide a reliable solution to efficiently recover from partial failures without restarting the whole analytical job.

4. Design of E^3

In this section, we present E^3, a scalable data processing system that we designed for Big Data analytic. The design of E^3 is largely inspired by two other large-scale data processing systems: MapReduce [4] and Dryad [13]. Following MapReduce, we design the programming interface to be as simple as possible. The burdens of parallelization, fault-tolerance, and data distribution are hidden by the runtime system. Like Dryad, we enable users to directly specify the data processing flow but with a much

simpler way. We also provide some features like indexing which are missing in those systems. In the following sections, we first present the programming model of E^3. Then, we describe several components of E^3 in detail.

4.1 Programming Model

The programming model of E^3 mainly consists of three interfaces: the `ProcessingUnit` interface for writing the analytical programs, the `InputIterator` and `OutputIterator` interfaces for retrieving records from the input and writing results to the output of the `ProcessingUnit`, and the `Splitter` interface for splitting inputs. Since E^3 is developed using Java, we only support Java as the programming language for the time being. Furthermore, users also provide a XML configuration file to connect `ProcessingUnits` as the data processing flow and specify the inputs and outputs of each `ProcessingUnit`. A concrete example of E^3 job configuration for the word count application [4] is presented in the appendix of this paper.

Users write data analytical programs by implementing the `ProcessingUnit` interface. The `ProcessingUnit` can have arbitrary number of inputs and produce arbitrary number of outputs. Each input is associated with a name which is defined in the job configuration file and is used in the `ProcessingUnit` for retrieving records from the corresponding data source. We support three kinds of inputs: files stored in a distributed filesystem (e.g., HDFS), relational tables stored in a DBMS, or outputs from another `ProcessingUnit`. Users can implement specific `InputIterator` interface to support more input types. The system can automatically split inputs of large size into a set of small sized splits and launch a set of `ProcessingUnits` to process those input splits, i.e., one process for each input. Users can fully control the splitting process by providing a specific `Splitter` implementation. Inspired by MapReduce, E^3 treats the input as a sequence, i.e., a list of records. Each record is a triple (pid, key, value) where pid is the partition number indicating which partition the record comes from and the key/value pair encodes the data of interest to analyze. The records in the input sequence can be iterated by a `InputIterator`. The `InputIterator` can be configured to iterate each record of the input by a sequential scan or a subset of records by an index scan. The details of the `InputIterator` will be presented in Section 4.4.

The `ProcessingUnit` writes results to its outputs by `OutputIterator`. Each output is also a sequence and is associated with a name. A special name "jobOutput" is reserved and is used to reference to the final output of the whole E^3 job.

Users specify the data processing flow in the job configuration file. The data processing flow is expressed as a set of stages, each of which consists of a set of `ProcessingUnits`. The stages are executed sequentially according to the declaration order. But the `ProcessingUnits` in each stage are launched by E^3 in parallel. Users link two `ProcessingUnits` defined in different stages by specifying the input of one `ProcessingUnit` as the output of the other one.

4.2 Execution Overview

E^3 is designed to run on a large shared-nothing cluster. Like

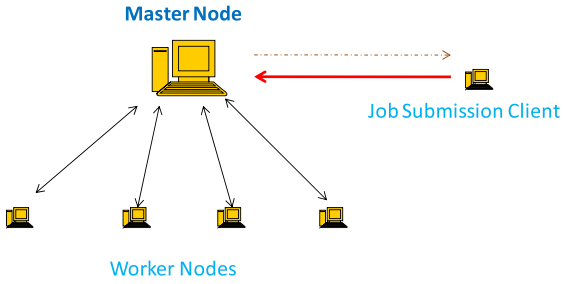


Fig. 2 Architecture of E^3.

MapReduce and Dryad, we employ the master-slaves architecture as shown in **Fig. 2**: one node acts as the single master node and the other nodes are worker nodes.

The master node is responsible for job management and the worker nodes launch processes to run `ProcessingUnits`. The E^3 job is executed as follows.

1. The first step is job initialization step. The master node builds a FIFO queue of processing stages according to the job specification file. For each stage, the master node creates a set of tasks for each `ProcessingUnit` in that stage by splitting the inputs of that `ProcessingUnit` with `Splitter`. For each input split that the `Splitter` produces, we create a single task for that input split.
2. The master node steps through the stage queue and executes those stages one by one. The master node executes a stage by assigning tasks of that stage to available worker nodes to launch. The status information of the launched stage is maintained by an in-memory state machine data structure. The stage is in ready state if it is scheduled by the master node, and in running state if the first of its tasks is assigned to the worker node. The stage is marked as completed if all of its tasks are completed. To overlap data transmission and computation, we allow master to schedule tasks from two consecutive stages simultaneously. The master node also uses a similar data structure to monitor the status of scheduled tasks.
3. When the worker node receives a task from the master node. It firsts check whether the inputs of the task are ready. An input item is ready if it is a data source item (e.g., a file or a database table). If the input item is the output of another `ProcessingUnit`, the input item is ready only if the output of that `ProcessingUnit` is fetched by the worker node.
4. When all inputs are ready, the worker node invokes the `ProcessingUnit` of this task to perform the computation. Unlike MapReduce in which user can only specify a single map or reduce function in each task, in E^3, the user can specify arbitrary number of map, reduce or any other data transformation functions. The user can either use pre-defined sequence processing algorithm or any customized algorithm to drive those functions. The user write intermediate and final results through `OutputIterator`. The intermediate results can be written to the memory of the `ProcessingUnit`. But the final results must be written to the output of the `ProcessingUnit` (i.e., a local file) so that the output can be fetched by other `ProcessingUnits`.

5. The master node uses a web page to display the status information of each submitted job. If all tasks of a job is completed, the job is thus marked as completed. The master node finally updates the job web page to notify the user.

4.3 Fast Startup

Launching tasks on a cluster of machines could be expensive especially for small jobs whose running time is short (typically less than 1 minute). The non-trivial startup overhead is well understood in previous studies [15], [19] that study the performance of Hadoop. In summary, previous studies show that the startup cost comes from two parts: 1) the cost of launching a new process in the worker node and 2) the timed-interrupt (i.e., polling) method for retrieving new tasks from the master node.

In E^3, we adopt two techniques to reduce the startup cost. First, we use the process pool technique, which is widely used in parallel databases [16], to entirely eliminate the cost of launching a new process. The worker node creates a fixed sized pool of processes at start up. After that, the worker node picks up an idle process to run a new task and return that process to the process pool when the task is completed.

Hadoop adopts the polling method for worker nodes to ask for tasks from the master node. To make the master node be scalable, worker nodes can only periodically send task requests to the master node in order to avoid overwhelming the master node. So, the master node can only assign tasks to worker nodes at certain time window determined by the interval of successive task requests. Even though it is scalable, this approach obviously produces long latency during task assignment. E^3 solved this problem by adopting a timed polling approach.

When the master node receives a task request from a worker node, if the master node has no available tasks at hand, instead of responding the worker node with a NULL task at once, the master node suspends the task request for a fixed time window. When the time window has passed and the master node still has no tasks, a NULL task is returned. When the worker node receives a NULL task, it sends the next task request immediately instead of waiting for a fixed period. Therefore, the master node can assign tasks to worker nodes in real time. A problem of the timed-polling approach is that when the master node suspends the task request, the thread for handling that task request is hang. Therefore, the thread pool of the master node will be quickly exhausted when a large number of task requests are pending. E^3 introduces a suspend/resume threading model to solve this problem. When the task request is suspended, the thread for handling that task request is returned to the thread pool instead of waiting. When the time window has passed, the master node re-allocates a thread from the thread pool to process the task request.

4.4 Data Access

The `ProcessingUnit` accesses records from the underlying storage system through the `InputIterator` interface. In E^3, the `InputIterator` interface, inspired by Generic Programming [11], [12], acts as the replacement of JDBC interface for data accessing.

As we described previously, the input of a `ProcessingUnit` is treated as a sequence. The sequence is represented by a pair of `InputIterator`s which points to the beginning and end of the sequence respectively. Users can consider `InputIterator` as a "pointer" which points to a certain record in the input sequence. The user can fetch the record that the `InputIterator` points at and forward `InputIterator` to sequentially iterate each record in the input sequence. The following codes show the definition of `InputIterator` interface. Users invoke `get()` to retrieve the record that the input iterator points to, use `increment()` to forward the input iterator, and test whether the input iterator passes the end of input with `notEquals()`

```
interface InputIterator {
    Triple get()
    InputIterator increment()
    boolean notEquals(InputIterator end)
}
```

The above interface only supports sequentially iterating each record in the input sequence. To efficiently process a subsequence of input through certain indexing structure, we require additional efforts. We first introduce the Dijkstra's notation [8] for representing a subsequence by using a pair of keys. Then we present the implementation of such notation in E^3.

The Dijkstra's notation represents an ordered subsequence of input with the form of a half open range $[k_1, k_2)$ (or $k_1 \leq k < k_2$ equivalently) where k_1 represents the first valid key in the subsequence and k_2 is the first invalid key out of the subsequence. For example, the half range $[1, 5)$ represents the ordered subsequence with 1, 2, 3, and 4 as the key. There are mainly two advantages of this notation. First, an algorithm which requires iterating each key in the subsequence can be expressed in a simple loop as follows:

```
for (k = k1; k != k2; k = next(k)) {
    \\ Do computations with k
}
```

The above algorithm only requires inequality comparison and a `next()` function which returns the next key in the subsequence. Thus, the algorithm is very easy to be implemented using our `InputIterator` interface. We just need to create a pair of iterators which points to the bounding keys and replace the bounding keys with the corresponding iterators as follows:

```
InputIterator begin = new InputIterator(k1)
InputIterator end = new InputIterator(k2)
for (k = begin; k.notEquals(end); k.increment()) {
    \\ Do computations with k
}
```

Therefore, to support index based range scan, we do not need to add additional methods to `InputIterator`. Providing an appropriate `InputIterator` implementation which can utilize the index structure to efficiently point to the bounding key is sufficient.

Second, it was shown that subsequence with three other forms (i.e., $(k_1, k_2]$, (k_1, k_2), $[k_1, k_2]$) can all be converted into the above half open notation by the "offset by one" trick [8]. Furthermore, a sequence with a single key can be represented as $[k, k)$. Thus, our `InputIterator` interface along with the Dijkstra's notation can be used to deal with any kinds of range scan and single point

lookup and compatible with all sorted indexes (e.g., B^+-tree) and hash indexes.

4.5 Data Types

Like MapReduce, the data type of key or value produced by the `InputIterator` is an uninterpreted byte string. However, the user can use any data types such as compound key and multi-attributed value by encoding these complex data types as byte strings and decoding those byte strings at runtime. The encoding process is performed at the data loading phase. Thus, the performance of encoding is less important since it is only performed once. However, one must care the decoding performance because the decoding function is invoked each time the data is processed.

In general, there are two decoding schemes: immutable decoding and mutable decoding. The immutable decoding scheme decodes a byte string by creating a dedicated *immutable* object of desired type in decoding and discarding that object after it is processed. The mutable decoding scheme creates a *"reusable"* object for decoding all byte strings and push that object to user function for processing. Our previous study shows that the mutable decoding is preferred since its performance is faster than the immutable decoding by an order [15]. Even though MapReduce does not restrict the user to employ immutable decoding scheme. Such decoding scheme is very popular in MapReduce domain resulting in a large performance degradation in a variety of data analytical tasks [15], [19], [23]. This may be due to the fact that the mutable decoding introduces more programming efforts than immutable decoding.

To achieve high performance, E^3 only adopts mutable decoding scheme. To reduce the programming effort, E^3 introduces a concept of `Assigner` and decouples the object creation and decoding procedure. To decode the byte string, users just create an object of desired type using standard Java `new Object()` statement and pass that object to the `Assigner`. The `Assigner` automatically reuses the object and handles all related issues.

4.6 Fault Tolerance

In current implementation, the fault tolerance mechanism of E^3 is identical to MapReduce. When a worker node fails, we re-execute all tasks that are assigned to that node. Completed tasks in the final stage are not needed to re-execute.

We have found that this coarse-grained fault tolerance mechanism is not efficient in many cases. We will adopt an exception based fault tolerance mechanism to efficiently recover partial or permanent software/hardware failures in the next version of E^3.

5. Performance Evaluation

In this section, we present the performance of E^3. We note that E^3 is part of the ongoing epiC project. At the present time, we have built the basic execution engine, the `InputIterator`s to access data stored in HDFS files, and `Assigner`s to parse records. However, some advanced features of E^3 are in early implementation phase and not ready for extensive experiments. Therefore, we only report the performance comparisons between $E^3$and Hadoop on the job startup and two simple analytical tasks drawn from a benchmark presented in Ref. [19]. A more comprehensive performance study will be conducted in the near future when the other pieces of the system have been built.

5.1 Benchmarking Environment

The benchmarking is conducted on a 65 nodes in-house cluster (called awan onwards) which includes one master node and 64 worker nodes. Each node in awan is equipped with an Intel Xeon X3430 Quad Core CPU (2.4 Ghz), 8 GB memory, two 450 GB SCSI disks, and 1 Gbps ethernet interface. The buffered reads of each disk is approximately 110 MB/sec. All nodes in awan run a CentOS operation system. We also use Java 1.6.0_16 as the runtime environment for both E^3 and Hadoop.

5.2 System Settings

Hadoop Settings: The benchmark is performed against Hadoop version 0.19.2. We configure the Hadoop system following instructions presented in Ref. [19]. The settings are summarized as follows: 1) The JVM runs in the server mode with maximal 1024 MB heap memory for either the map or reduce task; 2) We ran two concurrent map tasks in each TaskTracker node [*3]; 3) We enable the JVM reuse. 4) The I/O buffer is set to 128 KB; 5) The block size of HDFS is set to 256 MB; 6) The replication factor is set to 1 (no replication). The rest of Hadoop parameters remains their default values.

E^3 Settings: We set the size of process pool of each worker node to be two. We also set the size of thread pool of the master node for handling HTTP requests to be 200.

To deploy E^3 and Hadoop on the awan cluster, we perform an additional setting. For Hadoop, we run the JobTracker and NameNode on a dedicated awan node. Each of the other awan nodes act as both the DataNode and the TaskTracker node. Similarly, for E^3, we run the master node on a dedicated node and the worker node on each of the other nodes. To benchmark the scalability of both systems, we vary the cluster size, i.e., the number of worker nodes, from 8 to 64 nodes. Finally, we run each benchmark task three times and report the average result.

5.3 Datasets

For analytical tasks, we use two datasets (`Grep`, `Rankings`) which are chosen from Ref. [19]. The schemas of the two datasets are as follows.

```
CREATE TABLE Grep(
  key VARCHAR(10) PRIMARY KEY,
  field VARCHAR(90) );

CREATE TABLE Rankings(
  pageURL VARCHAR(100) PRIMARY KEY,
  pageRank INT,
  avgDuration INT);
```

For the Grep task, we generate datasets according to two settings: a small-scale setting and a large-scale setting. For the small-scale setting, we generate 5.6 million records (~535 MB) per node. In the large-scale setting, we generate 10 GB records

[*3] The analytical tasks that we performed do not require the reduce functions. Thus, we leave the setting of concurrent reduce tasks as the default value.

for each node. For the Selection task, we generate 18 million `Rankings` records (~1 GB) for each node. The generated datasets are stored as plain text files in the slave nodes and uploaded to HDFS without further modification.

5.4 Job Startup

The basic service that parallel data processing systems provides is launching the user program in a cluster for parallel execution. This process is called job startup or job stage-in. In general, the job startup process is expected to be as fast as possible so that the startup cost does not offset the benefits of parallelism.

In the first benchmark, we study the performance of job startup of two systems: E^3 and Hadoop. We design a special benchmark task called sleep job to achieve this goal. The idea is to execute a set of sleep tasks, each of which does dummy computations except for sleeping a fixed short period, on the cluster and measure the overall execution time of the whole job. Since the sleep task launched in each node does not perform real computations, the execution time of the whole job is mainly dominated by system overhead introduced by job start.

We run this benchmark in two settings. In the first setting, we vary the cluster size from 8 to 64 and launch two sleep tasks in each node to study the scalability of each benchmarked system. We launch two sleep tasks per node since both system (E^3 and Hadoop) are configured to be able to run two tasks concurrently. In the second setting, we fixed the cluster size to 64 and vary the sleep tasks launched in each node from 2 to 10 and study the performance of each benchmarked system with an increasing workload. In either setting, the sleep period is set to be 200 milliseconds.

For Hadoop, we implement a customized `MapRunner` for the sleep job which sleeps for the specified period and dost not invoke any map functions. We also set the number of reducers to be zero. For E^3, we implement a customized `Splitter` which output an empty input split for each sleep task. The E^3 job consists of a single `ProcessingUnit` which performs the trivial sleep task.

Figure 3 and **Fig. 4** show the results of this experiment. As can be clearly seen from Fig. 3, the startup cost of Hadoop is non-trial. The overall execution time ranges from 7.3 seconds (8 nodes) to 9.2 seconds (64 nodes) while the actual execution time of each task is only 0.2 seconds (200 milliseconds). E^3, on the other hand, is fairly efficient. The execution time ranges from 0.37 seconds (8 nodes) to 1.2 seconds (64 nodes). We attribute the efficiency of E^3 to its adoption of suspend/resume pattern for task assignments. The suspend/resume pattern significantly reduces the delay between successive task polling requests issued from worker nodes. The startup overhead of E^3 is mainly dominated by propagating the user program files to each cluster node. In general, the file propagation cost is proportional to the cluster size.

For the 64 nodes cluster, Fig. 4 shows that even the startup cost of Hadoop is still high, the cost is slightly amortized by the increasing workloads. However, there is still a large performance gap between E^3 and Hadoop.

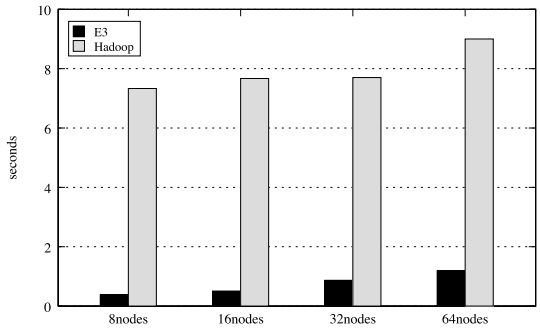


Fig. 3 Sleep task results – varying cluster size.

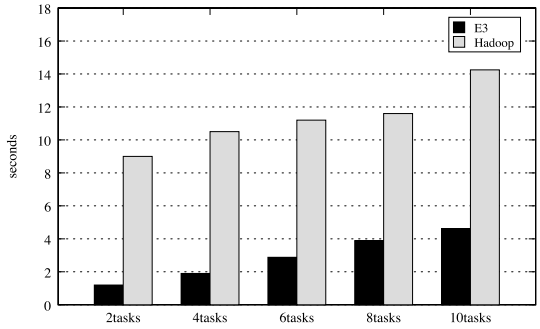


Fig. 4 Sleep task results – varying sleep tasks.

5.5 Grep Task

For the Grep task, both systems (E^3 and Hadoop) are required to scan through the input datasets searching for a three-character pattern "XYZ." The SQL command to perform this task is as follows:

```
SELECT * FROM Grep where field LIKE '%XYZ%';
```

Since the pattern appears once in every 10,000 records approximately, the performance of this work is mainly limited by the job startup/termination and the sequential scan speed of the data processing systems.

The MapReduce program that we used to perform this task is identical to the one described in Ref. [19]. The program consists of a single map function which takes the Grep record as the input key/value pair, performs a string match on the value part and finally outputs the matching record.

The E^3 job that performs the Grep task consists of a single stage which involves a single `ProcessingUnit` implementation called `GrepTask`. The `GrepTask` uses the same function in MapReduce program described above to perform the string matching logic and employs input and output iterators to retrieve records from the input and write matching records to the output.

Figure 5 and **Fig. 6** plot the results of this experiment on two settings (535 MB per node and 10 GB per node). Figure 5 shows that for small-scale dataset (i.e., 535 MB per node), E^3 performs significantly faster than Hadoop by a factor of 3.6. The speedup ratio between E^3 and Hadoop (i.e., 3.6) is larger than the speedup ratio between parallel databases against Hadoop (i.e., 2.5) reported in previous study [19]. This result confirms that E^3 can achieve similar or even better performance than parallel databases in certain analytical tasks such as Grep. We attribute the efficiency of E^3 to its efficient startup method. Launching a large-scale parallel tasks is not expensive in E^3. Low overhead startup enables the system to spend most its time on processing the data.

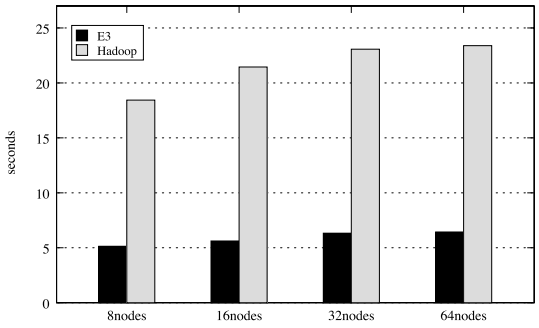


Fig. 5 Grep task results – 535 MB/node data set.

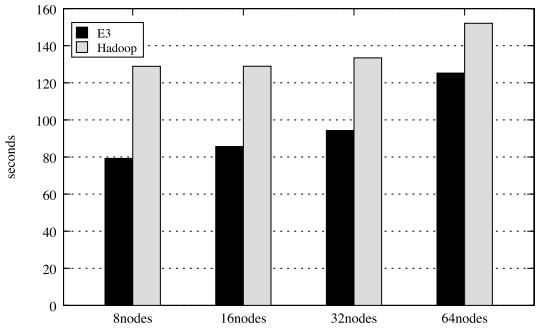


Fig. 6 Grep task results – 10 GB/node data set.

Overall, E^3 can process the data at the speed of 84 MB/sec to 104 MB/sec while Hadoop's data processing speed can approximately reach at most 30 MB/sec. For the large-setting (i.e., 10 GB per node), the performance gap between E^3 and Hadoop is much smaller since the startup cost of Hadoop is amortized by the increasing amount of data processing for this experiment. However, E^3 is still faster than Hadoop by around 30%.

5.6 Selection Task

In the second analytical task, we need to find records in the Rankings dataset (1 GB per node) whose `pageRank` is above a given threshold. The threshold is set to 10, which results in approximately 36,000 records out of 18 millions records in each node. The SQL command of this task is as follows:

```
SELECT pageURL, pageRank
  FROM Rankings where pageRank > 10
```

The MapReduce program that we used to perform this task is also identical to the one presented in Ref. [19]. A map function takes a text line from the source file as its input. The map function adopts a immutable decoding scheme to split the value part into multiple fields and check whether the `pageRank` field is greater than the threshold. Finally, the map function outputs the qualified records.

The E^3 job that we implemented to evaluate this task still consists of a single `ProcessingUnit` defined in a single stage. Like the MapReduce program, the `ProcessingUnit` also takes a text line from the source file as its input. However, the `ProcessingUnit` employs the mutable decoding scheme by creating a `Ranking` record and passing that record to the `Assigner` for decoding.

Figure 7 plots the performance of both system when performing this task. As can be clearly seen from Fig. 7, E^3 outperforms Hadoop by a even larger performance gap (by a factor of four).

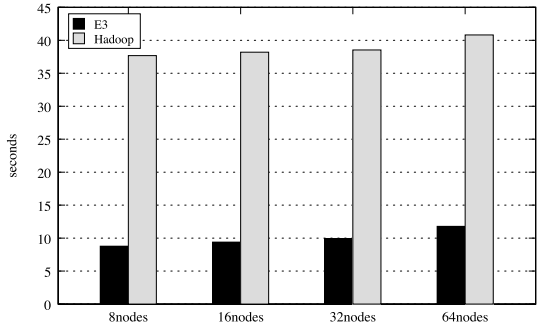


Fig. 7 Selection task results.

Again, the performance gap between E^3 and Hadoop is also larger than the performance gap between DBMS-X over Hadoop (by a factor of three at most) reported in Ref. [19] even DBMS-X adopts a cluster index on the selection column while E^3 does not employ any indexing techniques. The reason of the excellent performance of E^3 is due to its efficient task launching technique and reduction of decoding cost by adopting the mutable decoding scheme. Note that we are not claiming a system equipped with a fast startup and decoding scheme can always outperform a system employing storage optimizing techniques like indexing. In fact, we believe that E^3 can achieve better performance when the storage layer is optimized. However, in order to benchmark the performance of E^3 on indexed datasets, we need to implement appropriate `InputIterator`s. Hence, we leave it as future work. What we emphasized here, based on the experimental results, is that for a large-scale parallel data processing system, the runtime system overhead such as startup and decoding may also dominate the query processing. Therefore, in addition to optimize the storage layer, the system designer should also consider to reduce the runtime overhead at the system level as much as possible.

6. Conclusion

The unprecedented growth of data generated in the "information explosion era" nowadays has intrigued the design and development of MapReduce (and its open-source Hadoop) for massive-scale data processing. However, the performance of current Hadoop implementation has been noted to be unsatisfactory for database applications. We have identified the design factors that affect its performance and developed E^3, an elastic and efficient execution engine for scalable data processing.

E^3 adopts a "middle" communication model for connecting subtasks in the data flow of a job, while MapReduce and Dryad are on the other two extremes, in that E^3 has a simpler communication model than Dryad yet it can support multi-stages job better than MapReduce. The novelty of E^3 is that it avoids reprocessing intermediate results by adopting a stage-based evaluation strategy, and collocating data and user-defined functions into independent processing units for parallel execution.

More importantly, E^3 provides the richest programming model compared to MapReduce and Dryad for its support of block-level indexes, and built-in functions for specifying and optimizing data processing flows. Benchmarking on an in-house cluster shows that E^3 achieves better performance than Hadoop. In terms of future work we are going to develop and benchmark E^3 completely

and examine the ability to execute multiple jobs concurrently and the cooperation between them.

Acknowledgments The work in this paper was in part supported by the Singapore Ministry of Education Grant No. R252-000-454-112 under the project name of epiC. We would also like to thank other team members from National University of Singapore and Zhejiang University for their valuable contributions.

Reference

[1] Cao, Y., Chen, C., Guo, F., Jiang, D., Lin, Y., Ooi, B.C., Vo, H.T., Wu, S. and Xu, Q.: ES^2: A cloud data storage system for supporting both OLTP and OLAP, *International Conference on Data Engineering* (*ICDE'11*), pp.291–302 (2011).

[2] Chen, C., Chen, G., Jiang, D., Ooi, B.C., Vo, H.T., Wu, S. and Xu, Q.: Providing scalable database services on the cloud, *Proc. 11th International Conference on Web Information Systems Engineering* (*WISE'10*), pp.1–19 (2010).

[3] Chen, G., Vo, H.T., Wu, S., Ooi, B.C. and Özsu, M.T.: A Framework for Supporting DBMSlike Indexes in the Cloud, *Proc. VLDB Endow.*, Vol.4, No.11, pp.702–713 (2011).

[4] Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters, *Comm. ACM*, Vol.51, pp.107–113 (2008).

[5] Dewitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.I. and Rasmussen, R.: The Gamma Database Machine Project, *IEEE Trans. Knowl. and Data Eng.*, Vol.2, pp.44–62 (1990).

[6] DeWitt, D.J. and Gray, J.: Parallel Database Systems: The future of high performance database systems, *Comm. ACM*, Vol.35, No.6, pp.85–98 (1992).

[7] DeWitt, D.J., Paulson, E., Robinson, E., Naughton, J., Royalty, J., Shankar, S. and Krioukov, A.: Clustera: An integrated computation and data management system, *Proc. VLDB Endow.*, Vol.1, pp.28–41 (2008).

[8] Dijkstra, E.W.: Why numbering should start at zero (1982). circulated privately.

[9] Fushimi, S., Kitsuregawa, M. and Tanaka, H.: An Overview of The System Software of A Parallel Relational Database Machine GRACE, *Proc. 12th International Conference on Very Large Data Bases* (*VLDB'86*), pp.209–219 (1986).

[10] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L. and Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation, *Proc. 11th European PVM/MPI Users Group Meeting*, pp.97–104 (2004).

[11] Garcia, R., Järvi, J., Lumsdaine, A., Siek, J.G. and Willcock, J.: A comparative study of language support for generic programming, *OOPSLA*, pp.115–134 (2003).

[12] Gregor, D., Järvi, J., Siek, J.G., Stroustrup, B., Reis, G.D. and Lumsdaine, A.: Concepts: Linguistic support for generic programming in C++, *OOPSLA*, pp.291–310 (2006).

[13] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks, *SIGOPS Oper. Syst. Rev.*, Vol.41, pp.59–72 (2007).

[14] Jagadish, H.V., Ooi, B.C. and Vu, Q.H.: BATON: A balanced tree structure for peer-to-peer networks, *Proc. International Conference on Very Large Data Bases* (*VLDB'05*), pp.661–672 (2005).

[15] Jiang, D., Ooi, B.C., Shi, L. and Wu, S.: The performance of MapReduce: An in-depth study, *Proc. VLDB Endow.*, Vol.3, pp.472–483 (2010).

[16] Mehta, M. and DeWitt, D.J.: Managing Intra-operator Parallelism in Parallel Database Systems, *Proc. 21th International Conference on Very Large Data Bases* (*VLDB'95*), pp.382–394 (1995).

[17] Ng, W.S., Ooi, B.C., Tan, K.-L. and Zhou, A.: PeerDB: A P2P-based System for Distributed Data Sharing, *International Conference on Data Engineering* (*ICDE'03*), pp.633–644 (2003).

[18] Olston, C., Reed, B., Srivastava, U., Kumar, R. and Tomkins, A.: Pig latin: A not-so-foreign language for data processing, *Proc. 2008 ACM SIGMOD International Conference on Management of Data* (*SIGMOD'08*), pp.1099–1110 (2008).

[19] Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S. and Stonebraker, M.: A comparison of approaches to large-scale data analysis, *Proc. 35th SIGMOD International Conference on Management of Data* (*SIGMOD'09*), pp.165–178 (2009).

[20] Pike, R., Dorward, S., Griesemer, R. and Quinlan, S.: Interpreting the data: Parallel analysis with Sawzall, *Sci. Program.*, Vol.13, pp.277–298 (2005).

[21] Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Shenker, S.: A scalable content-addressable network, *SIGCOMM*, pp.161–172 (2001).

[22] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F. and Balakrishnan, H.: Chord: A scalable peer-to-peer lookup protocol for internet applications, *IEEE/ACM Trans. Netw.*, Vol.11, No.1, pp.17–32 (2003).

[23] Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A. and Rasin, A.: MapReduce and parallel DBMSs: friends or foes?, *Comm. ACM*, Vol.53, pp.64–71 (2010).

[24] Sunderam, V.S.: PVM: A framework for parallel distributed computing, *Concurrency: Pract. Exper.*, Vol.2, pp.315–339 (1990).

[25] Tarditi, D., Puri, S. and Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses, *SIGOPS Oper. Syst. Rev.*, Vol.40, pp.325–335 (2006).

[26] Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H. and Murthy, R.: Hive - a petabyte scale data warehouse using Hadoop, *International Conference on Data Engineering* (*ICDE'10*), pp.996–1005 (2010).

[27] Vo, H.T., Chen, C. and Ooi, B.C.: Towards elastic transactional cloud storage with range query support, *Proc. VLDB Endow.*, Vol.3, pp.506–514 (2010).

[28] Wang, J., Wu, S., Gao, H., Li, J. and Ooi, B.C.: Indexing multi-dimensional data in a cloud system, *Proc. SIGMOD International Conference on Management of Data* (*SIGMOD'10*), pp.591–602 (2010).

[29] Wu, S., Jiang, D., Ooi, B.C. and Wu, K.-L.: Efficient B-tree Based Indexing for Cloud Data Processing, *Proc. VLDB Endow.*, Vol.3, No.1, pp.1207–1218 (2010).

[30] Wu, S., Li, F., Mehrotra, S. and Ooi, B.C.: Query Optimization for Massively Parallel Data Processing, *Proc. SOCC* (2011). (To appear).

Appendix

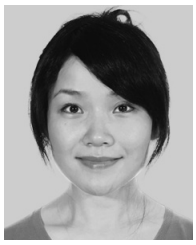
A.1 A Sample E^{3}Job

This section contains a E^{3}job that counts the number of occurrences of each unique word in the input files (i.e., the Word Count program presented in Ref. [4]). We present both the java program and the job configuration file. To save space, some trivial details like tokenizing the input value and Java generic types are intentionally omitted. The job consists of two stages. Each stage contains a single processing unit. The first stage tokenizes the input strings and the second stage produces the final word frequencies. The job configuration file specifies the input of each processing unit as well as the `Splitter` for splitting the input.

```
<?xml version = "1.0"?>
<JobConfig>
  <Output path="/results/WordCount" />

  <!-- First stage: Scan and tokenize the input value
  string -->
  <Stage description="Map phase">
    <ProcessingUnit name="WordCountMapper"
    class="WordCounter">
      <Input splitter="DFSSplitter">
        <Item name="WordInput" type="source"
        path="/test/data" />
      </Input>
    </ProcessingUnit>
  </Stage>

  <!-- Second stage: Compute the word frequencies -->
  <Stage description="Reduce phase">
    <ProcessingUnit name="WordCountReducer"
    class="Adder">
      <!-- PuOutputSplitter for splitting
      ProcessingUnit's output -->
      <Input splitter="PuOutputSplitter">
        <!-- The input is from the output of
```

```
        WordCounterMapper -->
        <Item name="AdderInput" type="inter"
        path="WordCountMapper" />
      </Input>
    </ProcessingUnit>
  </Stage>
</JobConfig>

public class WordCounter implements ProcessingUnit {

  private static class WordMapper implements Mapper {
    public void map(LongWritable key, Text value,
    OutputCollector output) {
      // Tokenize value
      // output.emit(word, 1);
    }
  }

  public void run(ProcessingUnit in,
  ProcessingUnitOutput out) {
    InputIterator begin =
    new LineInputIterator(in.get("WordInput"));
    InputIterator end = new LineInputIterator();
    OutputCollector collector =
    new Collector(out.getOutput());

    CoreAlg.forEach(begin, end, new WordMapper(),
    collector);
  }

}

public class Adder implements ProcessingUnit {

  private static class Adder implements Reducer {
    public void reduce(Text key, Iterator values,
    OutputCollector output) {
      // Sum and output the final value
      // output.emit(word, count);
    }
  }

  public void run(ProcessingUnit in,
  ProcessingUnitOutput out) {
    InputIterator begin =
    new InterInputIterator(in.get("AdderInput"));
    InputIterator end = new InterInputIterator();
    OutputCollector collector =
    new Collector(out.getOutput());

    CoreAlg.forEachGroup(begin, end, new Adder(),
    collector);
  }

}
```

Gang Chen is currently a Professor at the College of Computer Science, Zhejiang University. He received his B.Sc., M.Sc. and Ph.D. in Computer Science and Engineering from Zhejiang University in 1993, 1995 and 1998 respectively. His research interests include database, information retrieval, information security and computer supported cooperative work. Professor Gang is also the executive director of Zhejiang University - Netease Joint Lab on Internet Technology.



Ke Chen is an Associate Professor at the College of Computer Science, Zhejiang University. She earned her Ph.D. degree in Computer Science in 2007, and later became a Post-doctoral Fellow at the School of Aeronautics and Astronautics of Zhejiang University until year 2009. Her research interests include spatial temporal data management, web data mining, and data privacy protection. She is a member of ACM.

Dawei Jiang is currently a Research Fellow at the School of Computing, National University of Singapore. He received both his B.Sc. and Ph.D. in Computer Science from the Southeast University in 2001 and 2008 respectively. His research interests include Cloud computing, database systems and large-scale distributed systems.

Beng Chin Ooi received his B.Sc. (First Class Honors) and Ph.D. degrees from Monash University, Australia, in 1985 and 1989, respectively. He is a Professor of Computer Science at the School of Computing, National University of Singapore. His research interests include database performance issues, indexing techniques, XML, P2P/parallel/distributed computing, embedded system, Internet, and genomic applications. He has served as a PC member for international conferences such as ACM SIGMOD, VLDB, IEEE ICDE, WWW, and SIGKDD, and as Vice PC Chair for ICDE'00, '04, '06, co-PC Chair for SSD'93 and DASFAA'05, PC Chair for ACM SIGMOD'07 and Core DB PC chair for VLDB'08. He serves a PC Chair for IEEE ICDE'12. He was an Editor of VLDB Journal, and IEEE Transactions on Knowledge and Data Engineering. He serves as a co-chair of the ACM SIGMOD Jim Gray Best Thesis Award committee, Editor-in-Chief of IEEE Transactions on Knowledge and Data Engineering (TKDE), an Editor of Distributed and Parallel Databases Journal, an advisory board member of SIGMOD, and a trustee board member and executive of VLDB Endowment. He is the recipient of ACM SIGMOD 2009 Contributions Award. He is a founder of Thothe Technologies (1999), a company providing imaging and digital asset management solutions, the founder of BestPeer (2007), a company providing enterprise quality data processing over P2P corporate networks and P2P data streaming solutions, and a founder of Social@Work (2011).

Lei Shi received his bachelor degree in College of Information Science and Engineering, Northeastern University (NEU), China in 2008. Currently he is a Ph.D. candidate in School of Computing, National University of Singapore. His research interests include cloud computing infrastructure, parallel and distributed database theories and applications.

Hoang Tam Vo received his B.Eng. and M.Eng. degree in Computer Science from Ho Chi Minh City University of Technology, Vietnam, in 2005 and 2007, respectively. He is currently working towards the Ph.D. degree at School of Computing, National University of Singapore (NUS). His research interests include distributed data management techniques especially for P2P, community web and cloud environment.

Sai Wu received his Ph.D. degree from National University of Singapore (NUS) in 2011 and now is a Research Fellow at School of Computing, NUS. He got his bachelor and master degree from Peking University. His research interests include P2P systems, distributed database, cloud systems and indexing techniques. He has served as a Program Committee member for VLDB, ICDE and CIKM.