epiC: an Extensible and Scalable System for Processing Big Data

Dawei Jiang [†], Gang Chen [#], Beng Chin Ooi [†], Kian-Lee Tan [†], Sai Wu [#]
[†] School of Computing, National University of Singapore
[#] College of Computer Science and Technology, Zhejiang University
[†] {jiangdw, ooibc, tankl}@comp.nus.edu.sg
[#] {cg, wusai}@zju.edu.cn

ABSTRACT

The Big Data problem is characterized by the so called 3V features: Volume - a huge amount of data, Velocity - a high data ingestion rate, and Variety - a mix of structured data, semi-structured data, and unstructured data. The state-of-the-art solutions to the Big Data problem are largely based on the MapReduce framework (aka its open source implementation Hadoop). Although Hadoop handles the data volume challenge successfully, it does not deal with the data variety well since MapReduce enforces a key-value data model along with a row-oriented data processing strategy and bundles the data processing model with the underlying concurrent programming model.

This paper presents *epiC*, an extensible system to tackle the Big Data's data variety challenge. *epiC* introduces a general Actor-like concurrent programming model, independent of the data processing models, for specifying parallel computations. Users process multi-structured datasets with appropriate *epiC* extensions, the implementation of a data processing model best suited for the data type and auxiliary code for mapping that data processing model into *epiC*'s concurrent programming model. Like Hadoop, programs written in this way can be automatically parallelized and the runtime system takes care of fault tolerance and inter-machine communications. We present the design and implementation of *epiC*'s concurrent programming model. We also present two customized data processing model, an optimized MapReduce extension and a relational model, on top of *epiC*. Experiments demonstrate the effectiveness and efficiency of our proposed *epiC*.

1. INTRODUCTION

Many of today's enterprises are encountering the Big Data problems. A Big Data problem has three distinct characteristics (so called 3V features): the data volume is huge; the data format is varied and diverse (mixture of structured data, semi-structured data and unstructured data); and the data producing velocity is very high. These 3V features pose a grand challenge to traditional data processing systems since these systems either cannot scale to the huge data volume in a cost effective way or fail to handle data with variety of types [4][8].

Existing solutions tackle the Big Data challenge using the MapReduce programming model and its open source implementation Hadoop [9][1]. In MapReduce, data are structured as key-value pairs and a functional style data processing model is used to process those pairs. The data processing model consists of two functions: `map()` and `reduce()`. Users specify a `map()` function to process key-value pairs and produce a list of intermediate key-value pairs. The `reduce()` function, which is also users specified, merges the intermediate pairs with the same key to produce the final results. The MapReduce system has been shown to be highly scalable and resilient to machine failures. It has been reported that the system can successfully process peta-scale datasets and can be deployed on thousands of machines [9].

However, even though MapReduce successfully handles the Big Data's data volume challenge, the system dose not deal with the data variety challenge well. Analyzing a multi-structured dataset using the MapReduce framework is often inconvenient and inefficient. There are mainly three reasons for this. First, the key-value data model that MapReduce employs works well for unstructured data, but is not appropriate for structured data. Structured data should be best modeled as a relational table. Embedding a table record into a key-value pair is possible but is inconvenient and introduces runtime parsing overhead [16, 4, 21]. Second, MapReduce adopts a row-oriented data processing model which processes a record (i.e., a key-value pair) at a time. This data processing strategy is known to be inefficient for processing structured datasets and certain semi-structured datasets (e.g., RDF data [3]). To process those data, a column-oriented scheme (processing one column of the dataset at a time) has been shown to be more efficient. Third, the MapReduce implementation (i.e., Hadoop) bundles the data processing model with the concurrency model. Even though the programming model itself does not enforce concurrency, users can only write programs consisting of a single map and reduce functions. As such, the runtime system essentially requires complex operations to be expressed as a chain of MapReduce jobs. A lot of research have revealed that this chaining evaluation strategy is inefficient for computations like relational queries [8, 17, 28]. Furthermore, bundling the data processing model with the underlying concurrency model makes extending the programming model hard. Even simply adding an additional merge or join phase to the model requires considerable work to modify the runtime system [17, 28].

This paper presents a new system called *epiC* to tackle the Big Data's data variety challenge. The major contribution of this work is an architectural design that enables users to handle multi-structured data in the most effective and efficient way. The key feature of *epiC* is that it allows users to express an independent computation with the data processing model that is most appropriate for that data

Proceedings of the VLDB Endowment, Vol. X, No. Y

and then automatically parallelizes those independent computations. For example, users can employ the MapReduce data processing model to handle unstructured data and relational data processing to handle structured data; in addition, users can encapsulate or mix those codes in independent computations for parallel execution. To achieve this goal, *epiC* adopts an extensible design. The core abstraction of *epiC* is an Actor-like concurrent programming model. This concurrent programming model is independent of the underlying data processing models and is able to execute any number of independent computations (called units). On top of it, *epiC* provides a set of libraries (called extensions) that allows users to express independent computations in specific data processing models. In our current implementation, *epiC* supports two data processing models, namely MapReduce and relation database model. The decoupling of the concurrent programming model and the data processing models in *epiC* renders it extremely flexible and effective for processing multi-structured datasets.

In addition, *epiC* adopts several novel features. First, it employs a pure shared-nothing design. The underlying storage system (e.g., DFS, key-value store or distributed database) is accessible to all processing units. The unit performs its I/O operations and user-defined jobs independently without communications with others. When the job is done, it sends messages to the master network, which helps disseminate them. Note that the message only contains the control information and metadata of the intermediate results. In *epiC*, there is no shuffling phase, as all units access the distributed storage system directly. Second, *epiC* does not maintain the relationship of the units as a DAG (Directed Acyclic Graph). All units are equivalent, except their roles in the processing. The processing flow is represented as a message flow, which is handled by the master network automatically. In this way, we significantly reduce the programming overhead of the users. They do not need to explicitly express their logics as a DAG, such as in Dryad. Finally, the flexility of *epiC* provides the users more opportunities to customize their implementations for optimal performance. We will illustrate the idea using the equal-join algorithm.

The rest of the paper is organized as follows. Section 2 shows the overview of *epiC* and motivates our design with an example. Section 3 presents the programming abstractions introduced in *epiC*, focusing on the concurrency programming model and the MapReduce extension. Section 4 presents the internals of *epiC*. Section 5 evaluates the performance and scalability of *epiC* based on a selected benchmark of tasks. Section 6 presents related work. Finally, we conclude this paper in Section 7.

2. OVERVIEW OF EPIC

epiC adopts the Actor-like programming model. Each unit applies the user-defined logic to process the data from the underlying storage system independently in an asynchronized way. The only way to communicate with the other units is through message passing. However, unlike existing systems such as Dryad and Pregel [18], units cannot interact directly in *epiC*. All their messages are sent to the master network and then disseminated to the corresponding recipients. The master network is equivalent to the mail servers in the email system. In this way, we avoid maintaining the complex DAG for the units. Figure 1 shows an overview of *epiC*.

2.1 Programming Model

From the point of view of a unit, it works in an isolated way. A unit becomes activated when it receives a message from the master network. Based on the message content, it adaptively loads data from the storage system and applies the user-written codes to consume the data. After completing the process, the unit writes the results back to the storage system and the information of the intermediate results are summarized in a message forwarded to the master network. Then, the unit becomes inactive, waiting for the next message. Units are not aware the existence of each other. The only way of communications is via the master network.

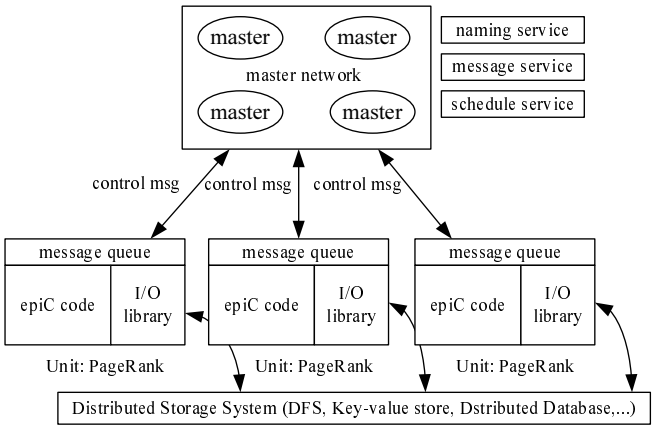

Figure 1: Overview of *epiC*

The master network consists of several synchronized masters, which are responsible for three services: naming service, message service and schedule service. Naming service assigns a unique namespace to each unit. In particular, we maintain a two-level namespace. The first level namespace indicates a group of units running the same user code. For example, in Figure 1, all units share the same first level namespace *PageRank*[20]. The second level namespace distinguishes the unit from the others. *epiC* allows the users to customize the second level namespace. Suppose we want to compute the PageRank values for a graph with 10,000 vertices. We can use the vertex ID range as the second level namespace. Namely, we evenly partition the vertex IDs into small ranges. Each range is assigned to a unit. A possible full namespace may be "[0, 999]@PageRank", where @ is used to concatenate the two namespaces. The master network maintains a mapping relationship between the namespace and the IP address of the corresponding unit process.

Based on the naming service, the master network collects and disseminates the messages to different units. The workload is balanced among the masters and we keep the replicas for the messages for fault tolerance. Note that in *epiC*, the message only contains the meta-information of the data. The units do not transfer the intermediate results via the message channel as in the shuffle phase of MapReduce. Therefore, the message service is a light-weight service with low overhead.

The schedule service of master network monitors the status of the units. If a failed unit is detected, a new unit will be started to take over its job. On the other hand, the schedule service also activates/deactivates units when they receive new messages or complete the processing. When all units become inactive and no more messages are maintained by the master network, the scheduler terminates the job.

Formally, the programming model of *epiC* is defined by a triple $< M, U, S >$, where M is the message set, U is the unit set and S is the dataset. Let $\mathcal{N}$ and $\mathcal{U}$ denote the universes of the namespace and data URIs. For a message $m \in M$, m is expressed as:

$$m := \{(ns, uri) | ns \in \mathcal{N} \wedge uri \in \mathcal{U}\}$$

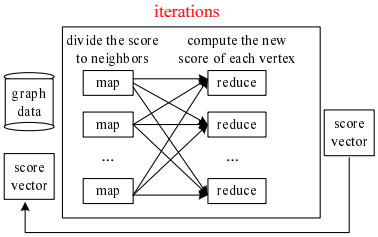

Figure 2: PageRank in MapReduce

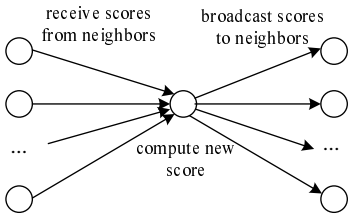

Figure 3: PageRank in Pregel

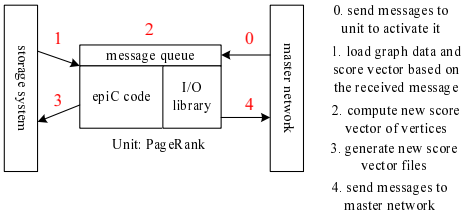

Figure 4: PageRank in *epiC*

We define a projection π function for m as:

$$\pi(m, u) = \{(ns, uri) | (ns, uri) \in m \wedge ns = u.ns\}$$

Namely, π returns the message content sharing the same namespace with u. π can be applied to M to recursively perform the projection. Then, the processing logic of a unit u in *epiC* can be expressed by function g as:

$$g := \pi(M, u) \times u \times S \rightarrow m_{out} \times S'$$

S' denotes the output data and m_{out} is the message to the master network satisfying:

$$\forall s \in S' \Rightarrow \exists (ns, uri) \in m_{out} \wedge \rho(uri) = s$$

where $\rho(uri)$ maps a URI to the data file. After the processing, S is updated as $S \cup S'$. As the behaviors of units running the same code are only affected by their received messages, we use (U, g) to denote a set of units running code g. Finally, the job J of *epiC* is represented as:

$$J := (U, g)^+ \times S_{in} \Rightarrow S_{out}$$

S_{in} is the initial input data, while S_{out} is the result data. The job J does not specify the order of execution of different units, which can be controlled by users for different applications.

2.2 Comparison with Other Systems

To appreciate the workings of *epiC*, we compare the way the PageRank algorithm is implemented in MapReduce (Figure 2), Pregel (Figure 3) and *epiC* (Figure 4). For simplicity, we assume the graph data and score vector are maintained in the DFS. Each line of the graph file represents a vertex and its neighbors. Each line of the score vector records the latest PageRank value of a vertex. The score vector is small enough to be buffered in memory.

To compute the PageRank value, MapReduce requires a set of jobs. Each mapper loads the score vector into memory and scans a chunk of the graph file. For each vertex, the mapper looks up its score from the score vector and then distributes its scores to the neighbors. The intermediate results are key-value pairs, where key is the neighbor ID and value is the score assigned to the neighbor. In the reduce phase, we aggregate the scores of the same vertex and apply the PageRank algorithm to generate the new score, which is written to the DFS as the new score vector. When the current job completes, a new job starts up to repeat the above processing until the PageRank values converge.

Compared to MapReduce, Pregel is more effective in handling iterative processing. The graph file is preloaded in the initial process and the vertices are linked based on their edges. In each super-step, the vertex gets the scores from its incoming neighbors and applies the PageRank algorithm to generate the new score, which is broadcast to the outgoing neighbors. If the score of a vertex converges, it stops the broadcasting. When all vertices stop sending messages, the processing can be terminated.

The processing flow of *epiC* is similar to Pregel. The master network sends messages to the unit to activate it. The message contains the information of partitions of the graph file and the score vectors generated by other units. The unit scans a partition of the graph file based on its namespace to compute the PageRank values. Moreover, it needs to load the score vectors and merge them based on the vertex IDs. As its namespace indicates, only a portion of the score vector needs to be maintained in the computation. The new score of the vertex is written back to the DFS as the new score vector and the unit sends messages about the newly generated vector to the master network. The recipient is specified as "*@PageRank". Namely, the unit asks the master network to broadcast the message to all units under the *PageRank* namespace. Then, the master network can schedule other units to process the messages. Although *epiC* allows the units to run asynchronously, to guarantee the correctness of PageRank value, we can intentionally ask the master network to block the messages, until all units complete their processing. In this way, we simulate the BSM (Block Synchronized Model) as Pregel.

We use the above example to show the design philosophy of *epiC* and why it performs better than the other two.

Flexibility MapReduce is not designed for such iterative jobs. Users have to split their codes into the map and reduce functions. On the other hand, Pregel and *epiC* can express the logic in a more natural way. The unit of *epiC* is analogous to the worker in Pregel. Each unit processes the computation for a set of vertices. However, Pregel requires to explicitly construct and maintain the graph, while *epiC* hides the graph structure by namespace and message passing. We note that maintaining the graph structure, in fact, consumes many system resources, which can be avoided by *epiC*.

Optimization Both MapReduce and *epiC* allow customized optimization. For example, the Haloop system [7] buffers the intermediate files to reduce the I/O cost and the units in *epiC* can maintain their graph partitions to avoid repeated scan. Such customized optimization is difficult to implement in Pregel.

Extensibility In MapReduce and Pregel, the users must follow the pre-defined programming model (e.g., map-reduce model and vertex-centric model), whereas in *epiC*, the users can design their customized programming model. We will show how the MapReduce model and the relational model are implemented in *epiC*. Therefore, *epiC* provides a more general platform for processing parallel jobs.

3. THE EPIC ABSTRACTIONS

epiC distinguishes two kinds of abstractions: a concurrent programming model and a data processing model. A concurrent programming model defines a set of abstractions (i.e., interfaces) for

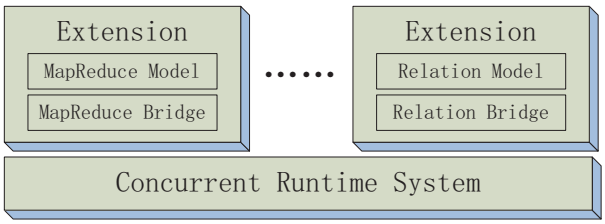

Figure 5: The Programming Stack of *epiC*

users to specify parallel computations consisting of independent computations and dependencies between those computations. A data processing model defines a set of abstractions for users to specify data manipulation operations. Figure 5 shows the programming stack of *epiC*. Users write data processing programs with extensions. Each extension of *epiC* provides a concrete data processing model (e.g., MapReduce extension offers a MapReduce programming interface) and auxiliary code (shown as a bridge in Figure 5) for running the written program on the *epiC*'s common concurrent runtime system.

We point out that the data processing model is problem domain specific. For example, a MapReduce model is best suited for processing unstructured data, a relational model is best suited for structured data and a graph model is best suited for graph data. The common requirement is that programs written with these models are all needed to be parallelized. Since Big Data is inherently multi-structured, we build an Actor-like concurrent programming model for a common runtime framework and offer *epiC* extensions for users to specify domain specific data manipulation operations for each data type. In the previoussection, we have introduced the basic programming model of *epiC*. In this section, we focus on two customized data processing model, the MapReduce model and relational model. We will show how to implement them on top of *epiC*.

3.1 The MapReduce Extension

We first consider the MapReduce framework, and extend it to work with *epiC*'s runtime framework. The MapReduce data processing model consists of two interfaces:

```
map     (k1, v1)          →  list(k2, v2)
reduce  (k2, list(v2))    →  list(v2)
```

Our MapReduce extension reuses Hadoop's implementation of these interfaces and other useful functions such as the `partition`. This section only describes the auxiliary support which enables users to run MapReduce programs on *epiC* and our own optimizations which are not supported in Hadoop.

3.1.1 General Abstractions

Running MapReduce on top of *epiC* is straightforward. We first place the `map()` function in a map unit and the `reduce()` function in a reduce unit. Then, we instantiate M map units and R reduce units. The master network assigns a unique namespace to each map and reduce unit. In the simplest case, the name addresses of the units are like "x@MapUnit" and "y@ReduceUnit", where $0 \leq x < M$ and $0 \leq y < R$.

Based on the namespace, the `MapUnit` loads a partition of input data and applies the customized `map()` function to process it. The results are a set of key-value pairs. Here, a `partition()` function is required to split the key-value pairs into multiple HDFS files. Based on the application's requirement, the `partition()` can choose to sort the data by keys. By default, the `partition()` simply applies the hash function to generate R files and assigns a namespace to each file. The meta-data of the HDFS files are composed into a message, which is sent to the master network. The recipient is specified as all the `ReduceUnit`.

The master network then collects the messages from all `MapUnits` and broadcasts them to the `ReduceUnit`. When a `ReduceUnit` starts up, it loads the HDFS files that share the same namespace with it. A possible merge-sort is required, if the results should be sorted. Then, the customized `reduce()` function is invoked to generate the final results.

```
class Map implements Mapper {
  void map() {
  }
}
class Reduce implements Reducer {
  void reduce() {
  }
}
class MapUnit implements Unit {
  void run(LocalRuntime r, Input i, Output o) {
    Message m = i.getMessage();
    InputSplit s = m[r.getNameAddress()];
    Reader reader = new HdfsReader(s);
    MapRunner map = new MapRunner(reader, Map());
    map.run();
    o.sendMessage("*@ReduceUnit",
        map.getOutputMessage());
  }
}
class ReduceUnit implements Unit {
  void run(LocalRuntime r, Input i, Output o) {
    Message m = i.getMessage();
    InputSplit s = m[r.getNameAddress()];
    Reader in = new MapOutputReader(s);
    ReduceRunner red = new ReduceRunner(in,
        Reduce());
    red.run();
  }
}
```

Here, we highlight the advantage of our design decision to decouple the data processing model and the concurrent programming model. Suppose we want to extend the MapReduce programming model to the Map-Reduce-Merge programming model [28]. All we need to do is to add a new unit `mergeUnit()` and modify the codes in the `ReduceUnit` to send messages to the master network for declaring its output files. Compared to this non-intrusive scheme, Hadoop needs to make dramatic changes to its runtime system to support the same functionality [28] since Hadoop's design bundles data processing model with concurrent programming model.

3.1.2 Optimizations for MapReduce

In addition to the basic MapReduce implementation which is similar to Hadoop, we add an optimization for map unit data processing. We found that the map unit computation is CPU-bound instead of I/O bound. The high CPU cost comes from the final sorting phase.

The Map unit needs to sort the intermediate key-value pairs since MapReduce requires the reduce function to process key-value pairs in an increasing order. Sorting in MapReduce is expensive since 1) the sorting algorithm (i.e., quick sort) itself is CPU intensive and 2) the data de-serialization cost is not negligible. We employ two techniques to improve the map unit sorting performance: 1) order-preserving serialization and 2) high performance string sort (i.e., burst sort).

Definition 1. For a data type T, an order-preserving serialization is an encoding scheme which serializes a variable $x \in T$ to a string

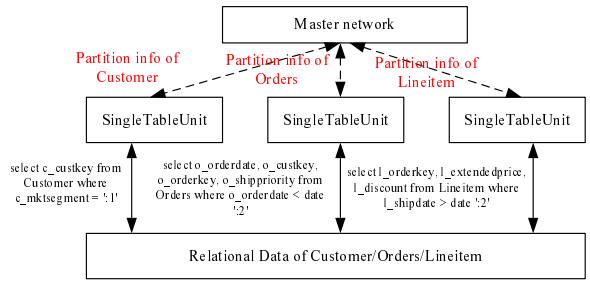

Figure 6: Step 1 of Q3

s_x such that, for any two variables $x \in T$ and $y \in T$, if $x < y$ then $s_x < s_y$ in string lexicographical order.

In other words, the order-preserving serialization scheme serializes keys so that the keys can be ordered by directly sorting their serialized strings (in string lexicographical order) without de-serialization. Note that the order-preserving serialization scheme exists for all Java built-in data types.

We adopt burst sort algorithm to order the serialized strings. We choose burst sort as our sorting technique since it is specially designed for sorting large string collections and has been shown to be significantly faster than other candidates [23]. We briefly outline the algorithm here. Interested readers should refer to [23] for details. The burst sort technique sorts a string collection in two passes. In the first pass, the algorithm processes each input string and stores the pointer of each string into a leaf node (bucket) in a burst trie. The burst trie has a nice property that all leaf nodes (buckets) are ordered. Thus, in the second pass, the algorithm processes each bucket in order, applies a standard sorting technique such as quick sort to sort strings, and produces the final results. The original burst sort requires a lot of additional memory to hold the trie structure and thus does not scale well to a very large string collection. We, thus, developed a memory efficient burst sort implementation which requires only 2 bits additional space for each key entry. We also use the multi-key quick sort algorithm [6] to sort strings resided in the same bucket.

Combining the two techniques (i.e., order-preserving serialization and burst sort), our sorting scheme outperforms Hadoop's quick sort implementation by a factor of three to four.

3.2 Relational Model Extension

As pointed out earlier, for structured data, the relational data processing model is most suited. Like the MapReduce extensions, we can implement the relational model on top of *epiC*.

3.2.1 General Abstractions

Currently, three core units (`SingleTableUnit`, `JoinUnit` and `AggregateUnit`) are defined for the relational model. They are capable of handling non-nested SQL queries. The `SingleTableUnit` processes the queries that involve only a partition of a single table. The `JoinUnit` reads partitions from two tables and merge them into one partition of the join table. Finally, the `AggregateUnit` collects the partitions of different groups and computes the aggregation results for each group. The abstractions of these units are shown below. Currently, we adopt the synchronization model as in MapReduce. Namely, we will start the next types of units, only when all current units complete their processing. We will study the possibility of creating a pipeline model in future work. Due to space limitation, we only show the most important part.

```
class SingleTableQuery implements DBQuery {
  void getQuery() {
  }
}
class JoinQuery implements DBQuery {
  void getQuery() {
  }
}
class AggregateQuery implements DBQuery {
  void getQuery() {
  }
}
class SingleTableUnit implements Unit {
  void run(LocalRuntime r, Input i, Output o) {
    Message m = i.getMessage();
    InputSplit s = m[r.getNameAddress()];
    Reader reader = new TableReader(s);
    EmbededDBEngine  e =
        new EmbededDBEngine(reader, getQuery());
    e.process();
    o.sendMessage(r.getRecipient(),
        e.getOutputMessage());
  }
}
class JoinUnit implements Unit {
  void run(LocalRuntime r, Input i, Output o) {
    Message m = i.getMessage();
    InputSplit s1 = m[r.getNameAddress(LEFT\_TABLE)];
    InputSplit s2 = m[r.getNameAddress(RIGHT\_TABLE)];
    Reader in1 = new MapOutputReader(s1);
    Reader in2 = new MapOutputReader(s2);
    EmbededDBEngine e =
        new EmbededDBEngine(in1, in2, getQuery());
    e.process();
    o.sendMessage(r.getRecipient(),
        e.getOutputMessage());
  }
}
class AggregateUnit implements Unit {
  void run(LocalRuntime r, Input i, Output o) {
    Message m = i.getMessage();
    InputSplit s = m[r.getNameAddress()];
    Reader in = new MapOutputReader(s);
    EmbededDBEngine e =
        new EmbededDBEngine(in, getQuery());
    e.process();
  }
}
```

The abstractions are straightforward and we discard the detailed discussion. In each unit, we embed a customized query engine, which can process single table queries, join queries and aggregations. We have not specified the recipients of each message in the unit abstraction. This must be implemented by users for different queries. However, as discussed later, we provide a query optimizer to automatically fill in the recipients. To show how users can adopt the above relational model to process queries, let us consider the following query (a variant of TPC-H Q3):

SELECT l_orderkey, sum(l_extendedprice(1-l_discount))*
as revenue, o_orderdate, o_shippriority
FROM customer, orders, lineitem
WHERE c_mktsegment = ':1' and c_custkey = o_custkey
and l_orderkey = o_orderkey and o_orderdate
< date ':2' and l_shipdate > date ':2'
Group By o_orderdate, o_shippriority

Figure 6 to Figure 10 illustrate the processing of Q3 in *epiC*. In step 1 (Figure 6), three types of the `SingleTableUnits` are started to process the select/project operators of Lineitem, Orders and Customer respectively. Note that those `SingleTableUnits` run the same code. The only differences are their name addresses

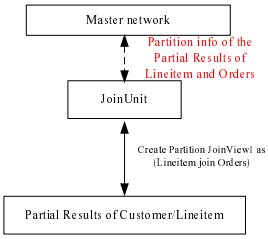

Figure 7: Step 2 of Q3

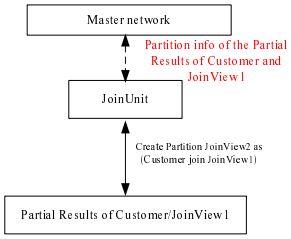

Figure 8: Step 3 of Q3

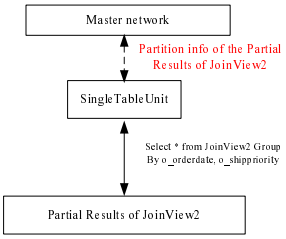

Figure 9: Step 4 of Q3

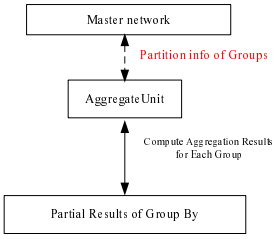

Figure 10: Step 5 of Q3

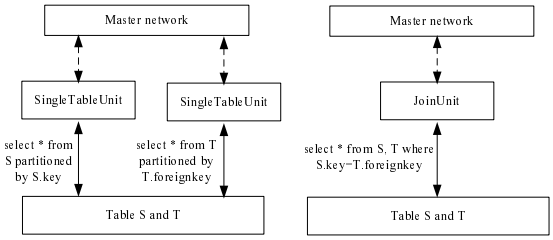

Figure 11: Basic Join Operation

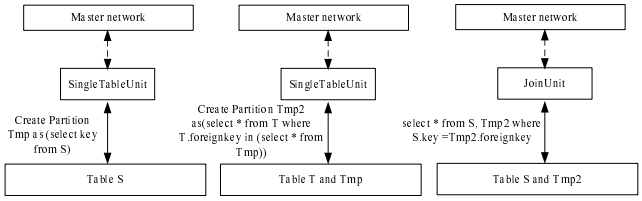

Figure 12: Semi-Join Operation

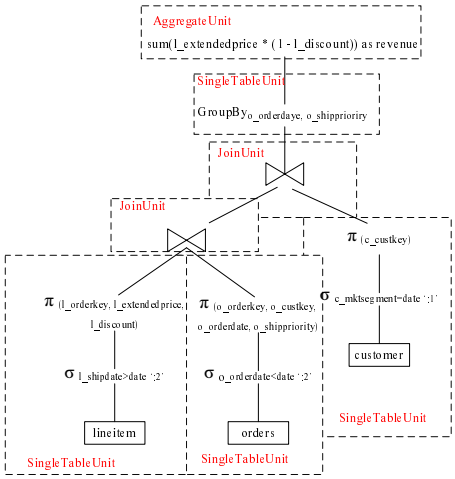

Figure 13: Job Plan of Q3

and processed queries. The results are written back to the storage system (either HDFS or distributed database). The meta-data of the corresponding files are forwarded to the `JoinUnits`.

In step 2 and step 3 (Figure 7 and 8), we apply the hash-join approach to process the data. In previous `SingleTableUnits`, the output data are partitioned by the join keys. So the `JoinUnit` can selectively load the paired partitions to perform the join. We will discuss other possible join implementations in the next section.

Finally, in step 4 (Figure 9), we perform the group operation. However, as the table is not distributed over the units, we actually get a partial group results. Therefore, in step 5 (Figure 10), the `AggregateUnit` needs to load partitions of the same group generated by different `SingleTableUnit` to compute the final aggregation results.

Our relational model simplifies the query processing, as users only need to consider how to partition the tables by the three units. Moreover, it also provides the flexibility of customized optimization.

3.2.2 *Optimizations for Relational Model*

The relational model on *epiC* can be optimized in two layers, the unit layer and the job layer.

In the unit layer, the user can adaptively combine the units to implement different database operations. They can even write their own units, such as `ThetaJoinUnit`, to extend the functionality of our model. In this section, we use the euqi-join as an example to illustrate the flexibility of the model. Figure 11 shows how the basic equi-join ($S \bowtie T$) is implemented in *epiC*. We first use the `SingleTableUnit` to scan the corresponding tables and partition the tables by join keys. Then, the `JoinUnit` loads the corresponding partitions to generate the results. In fact, the same approach is also used in processing Q3. We partition the tables by the keys in step 1 (Figure 6). So the following `JoinUnits` can perform the join correctly.

However, if most of tuples in S do not match tuples of T, semi-join is a better approach to reduce the overhead. Figure 12 illustrates the idea. The first `SingleTableUnit` scans table S and only outputs the keys as the results. The keys are used in the next `SingleTableUnit` to filter the tuples in T that cannot join with S. The intermediate results are joined with S in the last `JoinUnit` to produce the final results. As shown in the example, semi-join can be efficiently implemented using our relational model.

In the job layer, we offer a general query optimizer to translate the SQL queries into an *epiC* job. Users can leverage the optimizer to process their queries, instead of writing the codes for the relational model by themselves. The optimizer works as a conventional database optimizer. It first generates an operator expression tree for the SQL query and then groups the operators into different units. The message flow between units is also generated based on the expression tree. To avoid a bad query plan, the optimizer estimates the cost of the units based on the histograms. Currently, we only consider the I/O costs. The optimizer will iterate all variants of the expression trees and select the one with minimal estimated cost. The corresponding *epiC* job is submitted to the processing engine for execution. Figure 13 shows how the expression tree is partitioned into units for Q3.

The query optimizer acts as the AQUA [27] for MapReduce or PACTs compiler in Nephele [5]. But in *epiC*, the DAG between units are not conducted for data shuffling as in Nephele. Instead,

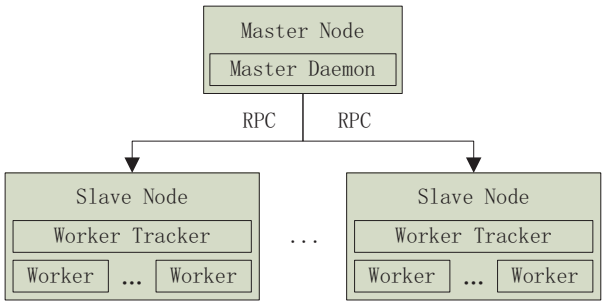

Figure 14: The architecture of an *epiC* cluster

all relationships between units are maintained through the message passing and namespaces. All units fetch their data from the storage system directly. This design follows the core concept of Actor model. The advantage is three-fold: 1) we reduce the overhead of maintaining the DAG; 2) we simplify the model as each unit runs in an isolated way; 3) the model is more flexible to support complex data manipulation jobs (either synchronized or asynchronized).

4. IMPLEMENTATION DETAILS

epiC is written in Java and built from scratch although we reuse some Hadoop codes to implement a MapReduce extension. This section describes the internals of *epiC*.

Like Hadoop, *epiC* is expected to be deployed on a shared-nothing cluster of commodity machines connected with switched Ethernet. It is designed to process data stored in any data sources such as databases or distributed file systems. The *epiC* software mainly consists of three components: master, worker tracker and worker process. The architecture of *epiC* is shown in Figure 14. *epiC* adopts a single master (this master is different from the servers in the master network, which are mainly responsible for routing messages and maintaining namespaces) multi-slaves architecture. There is only one master node in an *epiC* cluster, running a master daemon. The main function of the master is to command the worker trackers to execute jobs. The master is also responsible for managing and monitoring the health of the cluster. The master runs a HTTP server which hosts such status information for human consumption. It communicates with worker trackers and worker processes through remote procedure call (RPC).

Each slave node in an *epiC* cluster runs a worker tracker daemon. The worker tracker manages a worker pool, a fixed number of worker processes, for running units. We run each unit in a single worker process. We adopt this *'pooling'* process model instead of an on-demand process model which launches worker processes on demand for two reasons. First, pre-launching a pool of worker processes reduces the startup latency of job execution since launching a brand new Java process introduces non-trivial startup costs (typically 2$\sim$3 seconds). Second, the latest HotSpot Java Virtual Machine (JVM) employs a Just-In-Time (JIT) compilation technique to incrementally compile the Java byte codes into native machine codes for better performance. To fully unleash the power of HotSpot JVM, one must run a Java program for a long time so that every hot spot (a code segment, performing expensive computations) of the program can be compiled by the JIT compiler. Therefore, a never-ending worker process is the most appropriate one for this purpose.

Here, we will focus on two most important parts of the implementations, the TTL RPC and the failure recovery.

4.1 The TTL RPC

The standard RPC scheme adopts a client-server request-reply scheme to process RPC calls. In this scheme, a client sends a RPC request to the server. The server processes this request and returns its client with results. For example, when a task completes, the worker tracker will perform a RPC call `taskComplete(taskId)` to the master, reporting the completed task identity. The master will perform the call, updating its status, and responds to the worker tracker.

This request-reply scheme is inefficient for client to continuously query information stored at the server. Consider the example of task assignments. To get a new task for execution, the worker tracker must periodically make `getTask()` RPC calls to the master since the master hosts all task information and the worker tracker has no idea of whether there are pending tasks. This periodical-pulling scheme introduces non-negligible delays to the job startup since users may submit jobs at arbitrary time point but the task assignment is only performed at the fixed time points. Suppose the worker tracker queries a new task at time t_0 and the query interval is T, then all tasks of jobs submitted at $t_1 > t_0$ will be delayed to $t_0 + T$ for task assignment.

Since continuously querying server-side information is a common communication pattern in *epiC*, we develop a new RPC scheme to eliminate the pulling interval in successive RPC calls for low latency data processing.

Our approach is called the TTL RPC which is an extension of the standard RPC scheme by associating each RPC call with a user specified Time To Live (TTL) parameter T. The TTL parameter T captures the duration the RPC can live on the server if no results are returned from the server; when the TTL expires, the RPC is considered to have been served. For example, suppose we call `getTask()` with $T = 10s$ (seconds), when there is no task to assign, instead of returning a null task immediately, the master holds the call for at most 10 seconds. During that period, if the master finds any pending tasks (e.g., due to new job submission), the master returns the calling worker tracker with a new task. Otherwise, if 10 seconds passed and there are still no tasks to assign, the master returns a null task to the worker tracker. The standard request-reply RPC can be implemented by setting $T = 0$, namely no live.

We use a double-evaluation scheme to process a TTL-RPC call. When the server receives a TTL-RPC call C, it performs an initial evaluation of C by treating it as a standard RPC call. If this initial evaluation returns nothing, the server puts C into a pending list. The TTL-RPC call will stay in the pending list for at most T time. The server performs a second evaluation of C if either 1) the information that C queries changes or 2) T time has passed. The outcome of the second evaluation is returned as the final result to the client. Using TTL-RPC, the client can continuously make RPC calls to the server in a loop without pulling interval and thus receives server-side information in real time. We found that TTL-RPC significantly improves the performance of small jobs and reduces startup costs.

Even though the TTL-RPC scheme is a simple extension to the standard RPC scheme, the implementation of TTL-RPC poses certain challenges for the threading model that the classical Java network programs adopt. A typical Java network program employs a per-thread per-request threading model. When a network connection is established, the server serves the client by first picking up a thread from a thread pool, then reading data from the socket, and finally performing the appropriate computations and writing result back to the socket. The serving thread is returned to the thread pool after the client is served. This per-thread per-request threading model works well with the standard RPC communication. But it is not appropriate for our TTL RPC scheme since TTL

RPC request will stay at the server for a long time (We typically set $T = 20 \sim 30$ seconds). When multiple worker trackers make TTL RPC calls to the master, the per-thread per-request threading model produces a large number of hanging threads, quickly exhausting the thread pool, and thus makes the master unable to respond.

We develop a pipeline threading model to fix the above problems. The pipeline threading model uses a dedicated thread to perform the network I/O (i.e., reading request from and writing results to the socket) and a thread pool to perform the RPC calls. When the network I/O thread receives a TTL RPC request, it notifies the server and keeps the established connection to be opened. The server then picks up a serving thread from the thread pool and performs the initial evaluation. The serving thread will return to the thread pool after the initial evaluation no matter whether the initial evaluation produces the results or not. The server will re-pickup a thread from the thread pool for the second evaluation, if necessary, and notify the network I/O thread to complete the client request by sending out the results of the second evaluation. Using the pipeline threading model, no thread (serving threads or network I/O thread) will be hang during the processing of TTL RPC call. Thus the threading model is scalable to thousands of concurrent TTL RPC calls.

4.2 Fault Tolerance

Like all single master cluster architecture, *epiC* is designed be resilient to a large-scale slave machines failures. *epiC* treats a slave machine failure as a network partition from that slave machine to the master. To detect such a failure, the master communicates with worker trackers running on the slave machines by heartbeat RPCs. If the master cannot receive heartbeat messages from a worker tracker many times, it marks that worker tracker as dead and the machine where that worker tracker runs on as "failed".

When a worker tracker is marked as failed, the master will determine weather the tasks that the worker tracker processed need to be recovered. We assume that users persist the output of an *epiC* job into a reliable storage system like HDFS or databases. Therefore, all completed terminal tasks (i.e., tasks hosting units in the terminal group) need not to be recovered. We only recover in-progress terminal tasks and all non-terminal tasks (no matter completed or in-progress).

We adopt task re-execution as the main technique for task recovering and employ an asynchronous output backup scheme to speedup the recovering process. The task re-execution strategy is conceptually simple. However, to make it work, we need to make some refinements to the basic design. The problem is that, in some cases, the system may not find idle worker processes for re-running the failed tasks.

For example, let us consider a user job that consists of three unit groups: a map unit group M with two reduce groups R_1 and R_2. The output of M is processed by R_1 and the output of R_1 is further processed by R_2, the terminal unit group for producing the final output. *epiC* evaluates this job by placing three unit groups M, R_1 and R_2, in three stages S_1, S_2 and S_3 respectively. The system first launches tasks in S_1 and S_2. When the tasks in S_1 complete, the system will launch tasks in S_3, and at the same time, shuffle data from S_1's units to S_2's units.

Suppose at this time, a work tracker failure causes a task m's ($m \in M$) output to be lost, the master will fail to find an idle worker process for re-executing that failed task. This is because all worker processes are running tasks in S_2 and S_3 and the data lost introduced by m causes all tasks in S_2 to be stalled. Therefore, no worker process can complete and go back to the idle state.

We introduce a preemption scheduling scheme to solve the above deadlock problem. If a task A fails to fetch data produced by task B, the task A will notify the master and update its state to $in - stick$. If the master cannot find idle worker processes for recovering failed tasks for a given period of time, it will kill $in-stick$ tasks by sending `killTask()` RPCs to the corresponding worker trackers. The worker trackers then kill the $in - stick$ tasks and release corresponding worker processes. Finally, the master marks the killed $in - stick$ tasks as failed and adds them to the failed task list for scheduling. The preemption scheduling scheme solves the deadlock problem since *epiC* executes tasks based on the stage order. The released worker processes will first execute predecessor failed tasks and then the killed $in - stick$ tasks.

Algorithm 1 Generate the list of completed tasks to backup

Input: the worker tracker list W
Output: the list of tasks L to backup
1: **for** each worker tracker $w \in W$ **do**
2: $T \leftarrow$ the list of completed tasks performed by w
3: **for** each completed task $t \in T$ **do**
4: **if** $E_B(t) < E_R(t)$ **then**
5: $L \leftarrow L \cup \{t\}$
6: **end if**
7: **end for**
8: **end for**

Re-execution is the only approach for recovering in-progress tasks. For completed tasks, we also adopt a task output backup strategy for recovering. This scheme works as follows. Periodically, the master notifies the worker trackers to upload the output of completed tasks to HDFS. When the master detects a worker tracker W_i fails, it first commands another live worker tracker W_j to download W_i's completed tasks' output and then notifies all in-progress tasks that W_j will server W_i's completed tasks' output.

Backing up data to HDFS consumes network bandwidth. So, the master decides to backup a completed task's output only if the output backup can yield better performance than task re-execution recovery. To make such a decision, for a completed task t, the master estimates two expected execution time E_R and E_B of t where E_R is the expected execution time when the task re-execution scheme is adopted and E_B is the expected execution time when the output backup strategy is chosen. E_R and E_B are computed as follows

$$E_R = T_t \times P + 2T_t \times (1 - P) \quad (1)$$

$$E_B = (T_t + T_u) \times P + T_d \times (1 - P) \quad (2)$$

where P is the probability that the worker track is available during the job execution; T_t is the execution time of t; T_u is the elapsed time for uploading output to HDFS; and T_d is the elapsed time for downloading output from HDFS. The three parameters T_t, T_u and T_d are easily collected or estimated. The parameter P is estimated by the availability of a worker tracker in one day, namely we assume that each job can be completed in 24 hours.

The master uses Alg. 1 for determining which completed tasks should be backed up. The master iterates each worker tracker (line 1). For each worker tracker, the master retrieves its completed task list (line 2). Then, for each task in the completed task list, the master computes E_B and E_R and adds the task t into the result list L if $E_B < E_R$ (line 4 to line 5).

5. EXPERIMENTS

We evaluate the performance of *epiC* on different kinds of data processing tasks, including unstructured data processing, relational data processing and graph processing. We benchmark *epiC* against

Hadoop, an open source implementation of MapReduce for processing unstructured data and relational data and GPS [22], an open source implementation of Pregel [18] for graph processing, respectively. For all experiments, the results are reported by averaging six runs.

5.1 Benchmark Environment

The experimental study is conducted on an in-house cluster, consisting of 72 nodes hosted on two racks. The nodes within each rack are connected by a 1 Gbps switch. The two racks are connected by a 10 Gbps cluster switch. Each cluster node is equipped with a quad-core Intel Xeon 2.4GHz CPU, 8GB memory and two 500 GB SCSI disks. The `hdparm` utility reports that the buffered read throughput of the disk is roughly 110 MB/sec. However, due to the JVM costs, our tested Java program can only read local files at $70 \sim 80$ MB/sec.

We choose 65 nodes out of the 72 nodes for our benchmark. For the 65-node cluster, one node acts as the master for running Hadoop's NameNode, JobTracker daemons, GPS's server node and *epiC*'s master daemon. For scalability benchmark, we vary the number of slave nodes from 1, 4, 16, to 64.

5.2 System Settings

In our experiments, we configure the three systems as follows:

1. The Hadoop settings consist of two parts: HDFS settings and MapReduce settings. In HDFS settings, we set the block size to be 512 MB. As indicated in [16], this setting can significantly reduce Hadoop's cost for scheduling MapReduce tasks. We also set the I/O buffer size to 128 KB and the replication factor of HDFS to one (i.e., no replication). In MapReduce settings, each slave is configured to run two concurrent map and reduce tasks. The JVM runs in the server mode with maximal 1.5 GB heap memory. The size of map task's sort buffer is 512 MB. We set the merge factor to be 500 and turn off speculation scheduling. Finally, we set the JVM reuse number to -1.

2. For each worker tracker in *epiC*, we set the size of the worker pool to be four. In the worker pool, two workers are current workers (running current units) and the remaining two workers are appending workers. Similar to Hadoop's setting, each worker process has 1.5 GB memory. For the MapReduce extension, we set the bucket size of burst sort to be 8192 keys (string pointers).

3. For GPS, we employ the default settings of the system without further tuning.

5.3 Benchmark Tasks and Datasets

5.3.1 Benchmark Tasks

The benchmark consists of four tasks: Grep, TeraSort, TPC-H Q3, and PageRank. The Grep task and TeraSort task are presented in the original MapReduce paper for demonstrating the scalability and the efficiency of using MapReduce for processing unstructured data (i.e., plain text data). The Grep task requires us to check each record (i.e., a line of text string) of the input dataset and output all records containing a specific pattern string. The TeraSort task requires the system to arrange the input records in an ascending order. The TPC-H Q3 task is a standard benchmark query in TPC-H benchmark and is presented in Section 3.2. The PageRank algorithm [20] is an iterative graph processing algorithm. We refer the readers to the original paper [20] for the details of the algorithm.

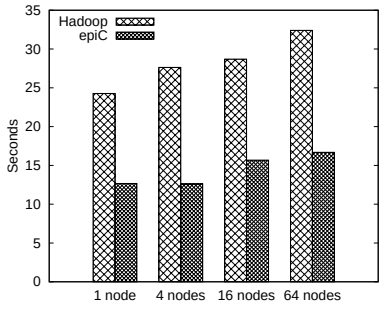

Figure 15: Results of Grep task with cold file system cache

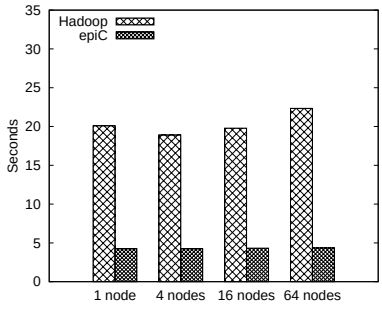

Figure 16: Results of Grep task with warm file system cache

5.3.2 Datasets

We generate the Grep and TeraSort datasets according to the original MapReduce paper published by Google. The generated datasets consists of N fixed length records. Each record is a string and occupies a line in the input file with the first 10 bytes as a key and the remaining 90 bytes as a value. In the Grep task, we are required to search the pattern in the value part and in the TeraSort task, we need to order the input records according to their keys. Google generates the datasets using 512 MB data per-node setting. We, however, adopt 1 GB data per-node setting since our HDFS block size is 512 MB. Therefore, for the 1, 4, 16, 64 nodes cluster, we, for each task (Grep and TeraSort), generate four datasets: 1 GB, 4 GB, 16 GB and 64 GB, one for each cluster setting.

We generate the TPC-H dataset using the `dbgen` tool shipped with TPC-H benchmark. We follow the benchmark guide of Hive, a SQL engine built on top of Hadoop, and generate 10GB data per node.

For the PageRank task, we use a real dataset from Twitter[1]. The user profiles were crawled from July 6th to July 31st 2009. For our experiments, we select 8 million vertices and their edges to construct a graph.

5.4 The Grep Task

Figure 15 and Figure 16 present the performance of employing *epiC* and Hadoop for performing Grep task with the cold file system cache and the warm file system cache settings, respectively.

In the cold file system cache setting (Figure 15), *epiC* runs twice faster than Hadoop in all cluster settings. The performance gap between *epiC* and Hadoop is mainly due to the startup costs. The heavy startup cost of Hadoop comes from two factors. First, for each new MapReduce job, Hadoop must launch brand new java

[1] http://an.kaist.ac.kr/traces/WWW2010.html

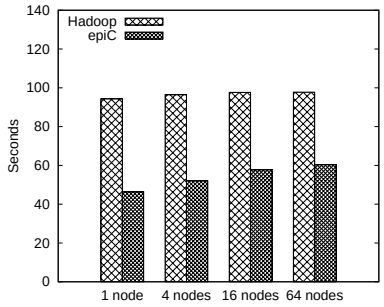

Figure 17: Results of TeraSort task with cold file system cache

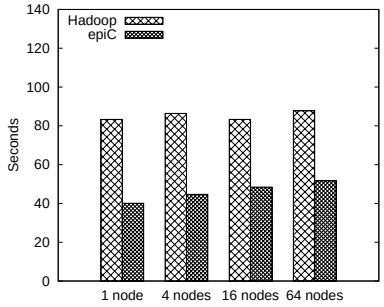

Figure 18: Results of TeraSort task with warm file system cache

processes for running the map tasks and reduce tasks [2]. The second, which is also the most important factor, is the inefficient pulling mechanism introduced by the RPC that Hadoop employed. In a 64-node cluster, the pulling RPC takes about 10∼15 seconds for Hadoop to assign tasks to all free map slots. *epiC*, however, uses the worker pool technique to avoid launching java processes for performing new jobs and employs TTL RPC scheme to assign tasks in real time. We are aware that Google has recently also adopted the worker pool technique to reduce the startup latency of MapReduce [8]. However, from the analysis of this task, clearly, in addition to the pooling technique, efficient RPC is also important.

In the warm file system cache setting (Figure 16), the performance gap between *epiC* and Hadoop is even larger, up to a factor of 4.5. We found that the performance of Hadoop cannot benefit from warm file system cache. Even, in the warm cache setting, the data is read from fast cache memory instead of slow disks, the performance of Hadoop is only improved by 10%. The reason of this problem is again due to the inefficient task assignments caused by RPC. *epiC*, on the other hand, only takes about 4 seconds to complete the Grep task in this setting, three times faster than performing the same Grep task in cold cache setting. This is because the bottleneck of *epiC* in performing the Grep task is I/O. In the warm cache setting, the *epiC* Grep job can read data from memory rather than disk. Thus, the performance is approaching optimality.

5.5 The TeraSort Task

Figure 17 and Figure 18 show the performance of the two systems (*epiC* and Hadoop) for performing TeraSort task in two settings (i.e., warm and cold cache). *epiC* beats Hadoop in terms of performance by a factor of two. There are two reasons for the performance gap. First, the map task of Hadoop is CPU bound. On average, a map task takes about 7 seconds to read off data from disk and then takes about 10 seconds to sort the intermediate data. Finally, another 8 seconds are required to write the intermediate data to local disks. Sorting approximately occupies 50% of the map execution time. Second, due to the poor pulling RPC performance, the notifications of map tasks cannot be propagated to the reduce tasks in a timely manner. Therefore, there is a noticeable gap between map completion and reduce shuffling.

epiC, however, has no such bottleneck. Equipped with order-preserving encoding and burst sort technique, *epiC*, on average, is able to sort the intermediate data at about 2.1 seconds, roughly five times faster than Hadoop. Also, *epiC*'s TTL RPC scheme enables reduce units to receive map completion notifications in real time. On average, *epiC* is able to start shuffling 5∼8 seconds earlier than Hadoop.

Compared to the performance of cold cache setting (Figure 17), both *epiC* and Hadoop do not run much faster in the warm cache setting (Figure 18); there is a 10% improvement at most. This is because scanning data from disks is not the bottleneck of performing the TeraSort task. For Hadoop, the bottleneck is the map-side sorting and data shuffling. For *epiC*, the bottleneck of the map unit is in persisting intermediate data to disks and the bottleneck of the reduce unit is in shuffling which is network bound. We are planning to eliminate the map unit data persisting cost by building an in-memory file system for holding and shuffling intermediate data.

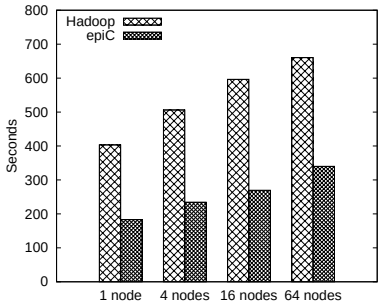

Figure 19: Results of TPC-H Q3

5.6 The TPC-H Q3 Task

Figure 19 presents the results of employing *epiC* and Hadoop to perform TPC-H Q3 under cold file system cache [3]. For Hadoop, we first use Hive to generate the query plan. Then, according to the generated query plan, we manually wrote MapReduce programs to perform this task. Our manually coded MapReduce program runs 30% faster than Hive's native interpreter based evaluation scheme. The MapReduce programs consist of five jobs. The first job joins `customer` and `orders` and produces the join results I_1. The second job joins I_1 with `lineitem`, followed by aggregating, sorting, and limiting top ten results performed by the remaining three jobs. The query plan and unit implementation of *epiC* is presented in Section 3.2.

Figure 19 shows that *epiC* runs about 2.5 times faster than Hadoop. This is because *epiC* uses fewer operations to evaluate the query (5 units vs. 5 maps and 5 reduces) than Hadoop and employs the asynchronous mechanism for running units. In Hadoop, the five jobs run sequentially. Thus, the down stream mappers must wait for the

[2]Hadoop can reuse java process for map and reduce tasks within the same job but cannot perform the reusing for tasks belonging to different jobs.

[3]For TPC-H Q3 task and PageRank task, the three systems cannot get a significant performance improvement from cache. Therefore, we remove warm cache results to save space.

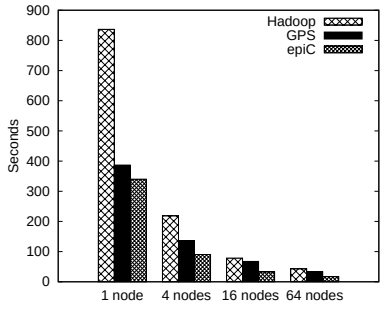

Figure 20: Results of PageRank

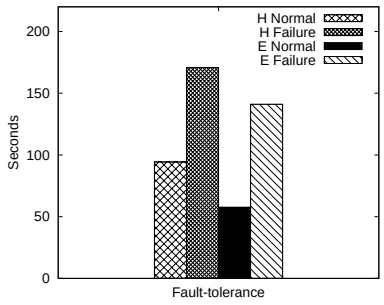

Figure 21: Fault tolerance experiment on a 16 node cluster

completion of all up stream reducers to start. In *epiC*, however, down stream units can start without waiting for the completion of up stream units.

5.7 The PageRank Task

This experiment compares three systems in performing the PageRank task. The GPS implementation of PageRank algorithm is identical to [18]. The *epiC* implementation of PageRank algorithm consists of a single unit. The details are discussed in Section 2.2. The Hadoop implementation includes a series of iterative jobs. Each job reads the output of the previous job to compute the new PageRank values. Similar to the unit of *epiC*, each *mapper* and *reducer* in Hadoop will process a batch of vertices. In all experiments, the PageRank algorithm terminates after 20 iterations. Figure 20 presents the results of the experiment. We find that all systems can provide a scalable performance. However, among the three, *epiC* has a better speedup. This is because *epiC* adopts an asynchronous communication pattern based on message passing, whereas GPS needs to synchronize the processing nodes and the Hadoop repeatedly creates new *mappers* and *reducers* for each job.

5.8 Fault Tolerance

The final experiment studies the ability of *epiC* for handling machine failures. In this experiment, both *epiC* and Hadoop are employed for performing the TeraSort task. During the data processing, we simulate slave machine failures by killing all daemon processes (TaskTracker, DataNode and worker tracker) running on those machines. The replication factor of HDFS is set to three, so that input data can be resilient to DataNode lost. Both systems (*epiC* and Hadoop) adopt heartbeating for failure detection. The failure timeout threshold is set to 1 minute. We configure *epiC* to use task re-execution scheme for recovering. The experiment is launched at a 16 node cluster. We simulate 4 machine failures at 50% job completion.

Figure 21 presents the results of this experiment. It can be seen that machine failures slow down the data processing. Both *epiC* and Hadoop experience 2X slowdown when 25% of the nodes fail (H-Normal and E-Normal respectively denotes the normal execution time of Hadoop and *epiC*, while H-Failure and E-Failure respectively denotes the execution time when machine failures occur).

6. RELATED WORK

As more and more enterprises are facing BigData problems, a number of systems have been built for large-scale distributed data processing in recent years. These systems can be classified into the following categories: 1) Parallel Databases, 2) MapReduce based systems, 3) DAG based data processing systems and 4) Actor-like systems.

The research on parallel databases started in the late 1980s [11]. Pioneering research systems include Gamma [10], and Grace [12]. Parallel databases are mainly designed for processing structured data sets where each data (called a record) strictly forms a table structure. Parallel databases employ data partitioning and partitioned execution techniques for high performance query processing. Recent parallel database systems also employ the column-oriented processing strategy to even improve the performance of analytical workloads such as OLAP queries [24]. Parallel databases have been shown to scale to at least peta-byte dataset but with a relatively high cost on hardware and software [4]. The main drawback of parallel databases is that those system cannot effectively process unstructured data. However, there are recent proposals trying to integrate Hadoop into database systems to mitigate the problem [25]. Our *epiC*, on the other hand, has been designed and built from scratch to provide the scalability, efficiency and flexibility found in both platforms.

MapReduce was proposed by Dean and Ghemawat in [9]. The system was originally developed as a tool for building inverted index for large web corpus. However, the ability of using MapReduce as a general data analysis tool for processing both structured data and unstructured data was quickly recognized [28] [26]. MapReduce gains popularity due to its simplicity and flexibility. Even though the programming model is relatively simple (only consists of two functions), users, however, can specify any kinds of computations in the `map()` and `reduce()` implementations. MapReduce is also extremely scalable and resilient to slave failures. The main drawback of MapReduce is its inefficiency for processing structured (relational) data and graph data. Many research work have been proposed to improve the performance of MapReduce on relational data processing [4][17]. The most recent work shows that, in order to achieve better performance of relational processing, one must relax the MapReduce programming model and make non-trivial modifications to the runtime system [8]. Our work is in parallel to these work. Instead of using a one-size-fit-all solution, we propose to use different data processing models to process different data and employ a common concurrent programming model to parallelize all those data processing.

Dryad is an ongoing research project at Microsoft [14] [15]. This work is close to ours since Dryad is intended for a general purpose data parallel programming framework. Dryad's DAG based computation model is also similar to our concurrent programming model. The difference between our work with Dryad is that our concurrent programming model is entirely independent of data processing model while Dryad's graph computation model assumes that the data passed to each vertex (a user specified computation) needs to be a structured item and each vertex must process one such data item at a time. This row oriented processing strategy and the implicit data model may degrade the performance if the data is best

suited for column oriented processing.

The the concept of Actor was originally proposed for simplifying concurrent programming [13]. Recently, systems like Storm [2] and S4 [19] implement the Actor abstraction for streaming data processing. Our concurrent programming model is also inspired by the Actor model. However, different from Storm and S4, our system is designed for batch data processing. A job of *epiC* will complete eventually. However, jobs of Storm and S4 may never end. This difference influenced us in choosing quite different design decisions from Storm and S4.

7. CONCLUSIONS

This paper presents *epiC*, a scalable and extensible system for processing BigData. *epiC* solves BigData's data volume challenge by parallelization and tackles the data variety challenge by the design of decoupling the concurrent programming model and the data processing model. To handle a multi-structured data, users process each data type with the most appropriate data processing model and wrap those computations in a simple unit interface. Programs written in this way can be automatically executed in parallel by *epiC*'s concurrent runtime system. In addition to the simple yet effective interface design for handling multi-structured data, *epiC* also introduces several optimizations in its Actor-like programming model. We use MapReduce extension and relational extension as two examples to show the power of *epiC*. The benchmarking of *epiC* against Hadoop and GPS confirms the efficiency advantage of *epiC*.

8. REFERENCES

[1] The hadoop offical website. http://hadoop.apache.org/.

[2] The storm project offical website. http://storm-project.net/.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[4] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *VLDB*, 2(1):922–933, Aug. 2009.

[5] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.

[6] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA*, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *VLDB*, 3(1-2):285–296, Sept. 2010.

[8] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, M. Wong, and M. Wong. Tenzing a sql implementation on the mapreduce framework. pages 1318–1327, 2011.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[10] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *VLDB*, pages 228–237, 1986.

[11] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[12] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine grace. In *VLDB*, pages 209–219, 1986.

[13] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, Mar. 2007.

[15] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD Conference*, pages 987–994, New York, NY, USA, 2009. ACM.

[16] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: an in-depth study. *VLDB*, 3(1-2):472–483, Sept. 2010.

[17] D. Jiang, A. K. H. Tung, and G. Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. *IEEE Trans. on Knowl. and Data Eng.*, 23(9):1299–1311, Sept. 2011.

[18] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[19] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[20] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[21] 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. In *SIGMOD Conference*, pages 165–178, 2009.

[22] S. Salihoglu and J. Widom. Gps: A graph processing system. In *SSDBM (Technical Report)*, 2013.

[23] R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics*, 9, Dec. 2004.

[24] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[25] X. Su and G. Swart. Oracle in-database hadoop: when mapreduce meets rdbms. In *SIGMOD Conference*, pages 779–790, 2012.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB*, 2(2):1626–1629, Aug. 2009.

[27] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *SoCC*, page 12, 2011.

[28] H. Yang, A. Dasdan, R. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD Conference*, pages 1029–1040, New York, NY, USA, 2007. ACM.