Noname manuscript No.
(will be inserted by the editor)

epiC: an extensible and scalable system for processing Big Data

Dawei Jiang · Sai Wu · Gang Chen · Beng Chin Ooi · Kian-Lee Tan · Jun Xu

Received: date / Accepted: date

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 the programming interfaces and its associated data processing model is inconvenient and inefficient for handling structured data and graph data.

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 models, an optimized MapReduce extension and a relational model, on top of *epiC*. We show how users can leverage *epiC* to process heterogeneous data by linking different types of operators together. To improve the performance of complex analytic jobs, *epiC* supports a partition-based optimization technique where data are streamed between the operators to avoid the high I/O overheads. Experiments demonstrate the effectiveness and efficiency of our proposed *epiC*.

Keywords Parallel Processing · MapReduce · Pregel · Hadoop

Dawei Jiang, Beng Chin Ooi and Kian-Lee Tan
School of Computing, National University of Singapore, Singapore, Singapore
E-mail: {jiangdw, ooibc, tankl}@comp.nus.edu.sg

Gang Chen and Sai Wu
College of Computer Science and Technology, Zhejiang University, Hangzhou, China
E-mail: {cg, wusai}@zju.edu.cn

Jun Xu
School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
E-mail: junxu1991@gmail.com

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 type is 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 [3][7].

A popular approach to process Big Data is to use the MapReduce programming model and its open source implementation Hadoop [8][1]. The advantage of MapReduce is that the system tackles the data volume challenge successfully and is resilient to machine failures [8]. Unfortunately, the MapReduce programming model does not handle the data variety problem well - while it manages certain unstructured data (e.g., plain text data) effectively, the programming model and its associated data processing scheme is inconvenient and inefficient for processing structured data and graph data that require DAG (Directed Acyclic Graph) like computation and iterative computation [21,3,29,26,36]. Thus,

systems like Dryad [18] and Pregel [26] are built to process those kinds of analytical tasks.

As a result, to handle the data variety challenge, the state-of-the-art approach favors a hybrid architecture [3, 11]. The approach employs a hybrid system to process multi-structured datasets (i.e., datasets containing a variety of data types: structured data, text, graph). The multi-structured dataset is stored in a variety of systems based on types (e.g., structured data are stored in a database, unstructured data are stored in Hadoop). Then, a split execution scheme is employed to process those data. The scheme splits the whole data analytical job into sub-jobs and choose the appropriate systems to perform those sub-jobs based on the data types. For example, the scheme may choose MapReduce to process text data, database systems to process relational data, and Pregel to process graph data. Finally, the output of those sub-jobs will be loaded into a single system (Hadoop or database) with proper data formation to produce the final results. Even though the hybrid approach is able to employ the right data processing system to process the right type of data, it introduces complexity in maintaining several clusters (i.e., Hadoop cluster, Pregel cluster, database cluster) and the overhead of frequent data formation and data loading for merging output of sub-jobs during data processing.

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 within a single system. We found that although different systems (Hadoop, Dryad, Database, Pregrel) are designed for different types of data, they all share the same shared-nothing architecture and decompose the whole computation into independent computations for parallelization. The differences between them are the types of independent computation that these systems allow and the computation patterns (intermediate data transmission) that they employ to coordinate those independent computations. For example, MapReduce only allows two kinds of independent computations (i.e., map and reduce) and only allows transmitting intermediate data between mappers and reducers. DAG systems like Dryad allow arbitrary number of independent computations and DAG-like data transmission. Graph processing systems like Pregel employ recursive /iterative data transmission. Therefore, if we can decompose the computation and communication pattern by building a common runtime system for running independent computations and developing plug-ins for implementing specific communication patterns, we are able to run all those kinds of computations in a single system. To achieve this goal, *epiC* adopts an extensible design. The core abstraction of *epiC* is an Actor-like concurrent programming model which is able to execute any number of independent computations (called units). On top of it, *epiC* provides a set of extensions that enable users to process different types of data with different types of data processing models (MapReduce, DAG or Graph). In our current implementation, *epiC* supports two data processing models, namely MapReduce and relation database model.

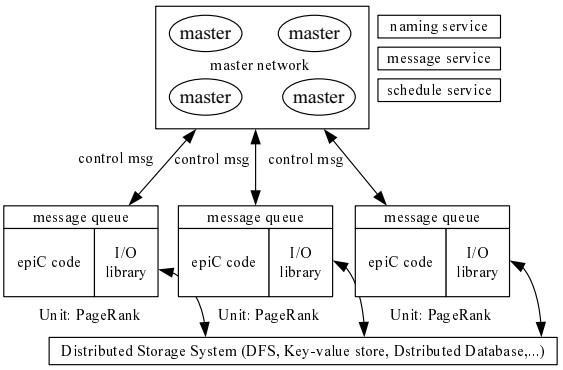


Fig. 1 Overview of *epiC*

The concrete design of *epiC* is summarized as follows. The system 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 enforce units to communicate with each other through a DAG (Directed Acyclic Graph). All units are equivalent, except for their roles in processing. The processing flow is represented as a message flow, which is handled by the master network. This flexible design provides the users more opportunities to customize their implementations for optimal performance. The equal-join algorithm will be used to illustrate this point.

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. In Section 4, we show how *epiC* can be extended to support the processing of heterogeneous data, and in Section 5, we present a novel optimization technique adopted in *epiC*. Section 6 presents the internals of *epiC*. Section 7 evaluates the performance and scalability of *epiC* based on a selected benchmark of tasks. Section 8 presents related work. Finally, we conclude this paper in Section 9.

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 [26], 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 similar to the mail servers in the email system. 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 of the existence of each other. The only way of communications is via the master network.

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* [28]. 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:

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

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

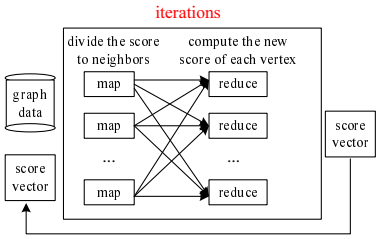


Fig. 2 PageRank in MapReduce

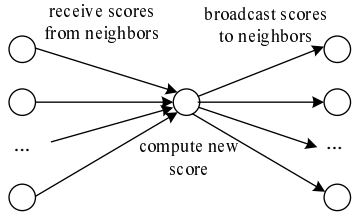


Fig. 3 PageRank in Pregel

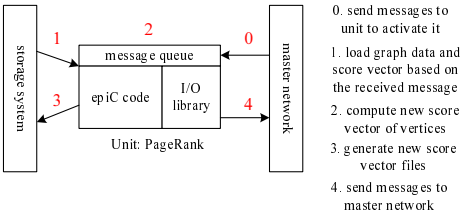


Fig. 4 PageRank in *epiC*

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 BSP (Bulk Synchronous Parallel 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 [6] 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 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

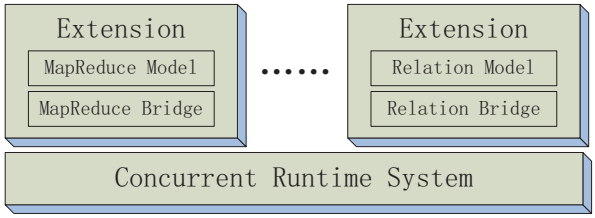


Fig. 5 The Programming Stack of *epiC*

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 previous section, 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 [36]. 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 [36] 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 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 [31]. We briefly outline the algorithm here. Interested readers should refer to [31] 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 of additional space for each key entry. We also use the multi-key quick sort algorithm [5] 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*.

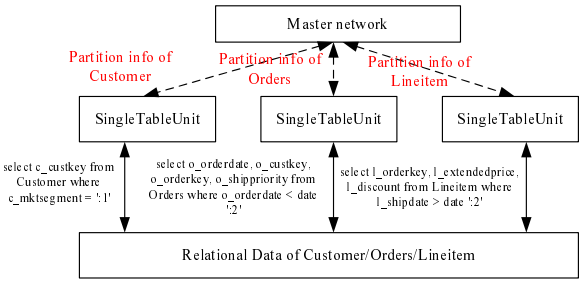


Fig. 6 Step 1 of Q3

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 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);
```

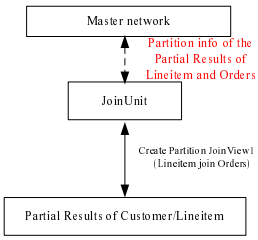


Fig. 7 Step 2 of Q3

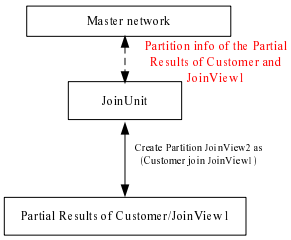


Fig. 8 Step 3 of Q3

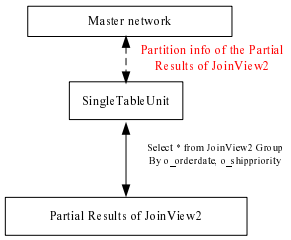


Fig. 9 Step 4 of Q3

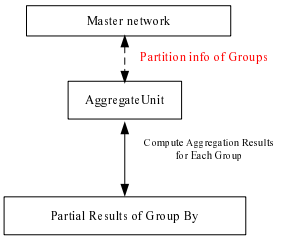


Fig. 10 Step 5 of Q3

```
    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 TPC-H Q3 in *epiC*. In step 1 (Figure 6), three different 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 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 (Figures 7 and 8), we apply the hash-join approach to process the data. In previous step, 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 for two attributes. As the join results are partitioned into multiple chunks, one `SingleTableUnit` can only generate the grouping results for its own chunk. To produce the complete grouping results, we merge groups generated by different `SingleTableUnits`. Therefore, in step 5 (Figure 10), one `AggregateUnit` needs to load the partitions generated by all `SingleTableUnits` for the same group 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 the 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

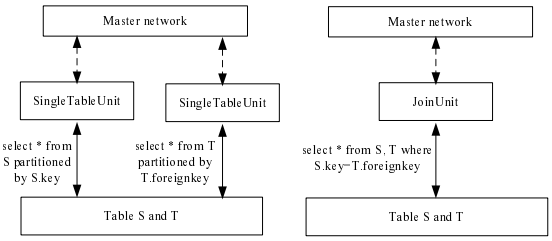


Fig. 11 Basic Join Operation

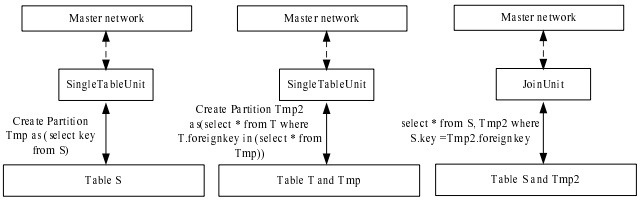


Fig. 12 Semi-Join Operation

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 over all variants of the expression trees and select the one with the 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 [35] for MapReduce or PACTs compiler in Nephele [4]. But in *epiC*, the DAG between units are not used for data shuffling as in Nephele. Instead, 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).

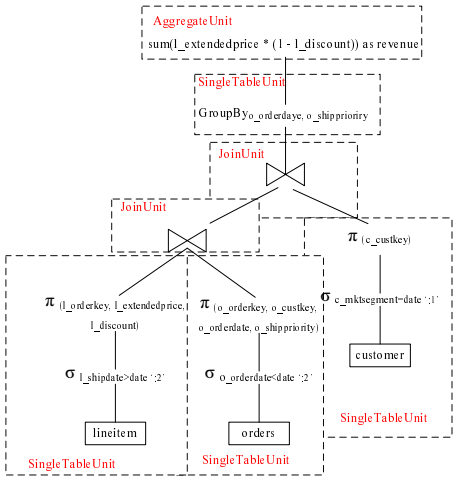


Fig. 13 Job Plan of Q3

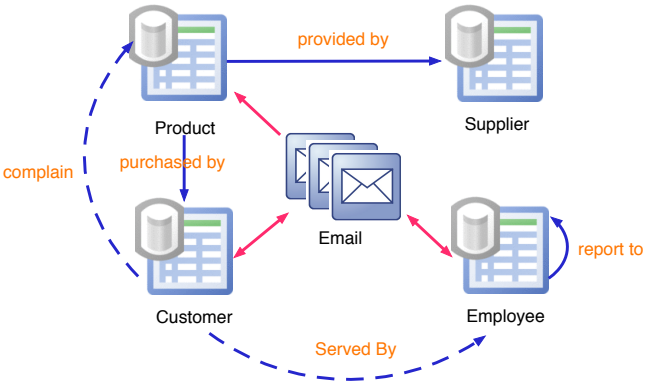


Fig. 14 Data Analysis on Heterogeneous Data

4 Processing Heterogeneous data

Data is expected to grow by 800% over the next five years, and 80% of it will be unstructured[1]. Most business information are maintained in the unstructured form, such as e-mail, phone record, PowerPoint slides and Word documents. Compared to the structured data, which are well managed and exploited, the similar analytical tool to process the unstructured data is missing. Even though data mining algorithms and machine learning models have been developed for analyzing unstructured data, the main problem of correlating the two kinds of datasets (i.e., structured dataset and unstructured dataset) is still not solved. This is because the unstructured data are handled separately from the structured data in the database. To produce more insights, it is a must to perform the data analytical job on top of the heterogeneous dataset.

Figure 14 illustrates an example. This example demonstrates a business with four entities: suppliers, products, em-

[1] http://www.computerworld.com/s/article/352399/XP_Deadline_Haunts_IT?source=CTWNLE_nlt_msft_2010-10-25

Table 1 Customer

UID	Name	City	Age	Gender	Income($)
U10001	Jason Mraz	New York	23	M	38,218
U10002	Tyler Perry	Atlanta	30	M	22,308
U10003	Emily Lee	Chicago	46	F	10,483
U10004	John Lewis	L.A.	48	M	17,287
U10005	Lucy Chen	New York	25	F	10,057

Table 2 Email

DocID	Content
100001	Hi, I am Tyler Perry. You guys have built a great product...
100002	Dear Ms Lucy Chen, We are sorry for the disruption of our service...
100003	The service is not well designed, while my friend Emily said... regards Jason Mraz
100004	I am Emily Lee. Thanks for providing such a good product, but...
100005	Dear Mr Jasonn Marz, regarding to your problem, we...

ployees and customers. Suppliers provide products. Employees (our customer service representatives) report to their managers and serve customers. Customers communicate with customer service via emails for asking the support of specific products. The customer emails provide the detail information of which customer complains about which product and how the employees serve the customers. In the database, we maintain the relational tables for customers, products, suppliers and employees. . Emails are stored a separate file system (outside the database). Table 1 and 2 show sample data of customer table and emails. The customer opinions about products are hidden in the emails which must be retrieved using NLP techniques.

We consider the following two queries:

- Find the supplier whose products received the most complains.
- Evaluate the customer satisfaction rate for each manager's group.

To answer such queries, we need NLP (National Language Processing) technique to process the unstructured data (emails) and database technique to handle the relational data. It is not a trivial task to integrate these two techniques seamlessly into one system. However, due to the flexibility of *epiC*, we can effectively support such applications by enhancing our relational model presented in Section 3.2.

4.1 NLP Operators

Let $\mathcal{D}$ denote the document set for processing. For each document d, we represent it as a set of keywords $d = \{k_0, ..., k_n\}$. To support text mining jobs, we implement NLP operators as *epiC* units which can be classified as two categories. The first type of operators are used to search the document set $\mathcal{D}$ and locate the position of a specific keyword k which are listed as below:

1. search(f, q, ϵ): Given a distant function f, return a document set $\mathcal{D}' \in \mathcal{D}$ which satisfies that
$$\forall d_i \in \mathcal{D}' \rightarrow \exists k_j \in d_i \wedge f(k_j, q) < \epsilon$$
If f is edit distance and $\epsilon = 0$, this is exactly the keyword search.
2. pos(id, f, q, ϵ): Given a document identified by its ID, return all positions of the keywords that match q under the distance function f and threshold ϵ.
3. get(id, p, offset): Given a document identified by its ID, return a text string starting at the position $p - offset$ to $p + offset$.

The second type of operators are used to perform the complex model-based analysis. Currently, we implement two operators:

1. entity(id, $type$) : Given a document identified by its ID, return all entities of a specific type (e.g., location or person).
2. sentiment(s) : Given a text string, return its semantic sentiment (e.g., positive, negative or neutral).

To use model-based operators, we must first train the corresponding models. In particular, we use CRF model [13] for entity recognition and feature-based [17] model for sentiment analysis. We adopt the two models, as they are easy to compute in parallel and also incur less storage overheads, so every *epiC* unit can buffer those models in its memory.

4.2 Hybrid Join

To answer the example queries, we need to join the unstructured data with relational tables. We introduce the hybrid join operator, $\bar{\bowtie}$ to accomplish the requirement. The hybrid join is defined as:

Definition 2 Hybrid Join
$T \bar{\bowtie}_{search(f,c_i,\epsilon)} \mathcal{D}$ is a hybrid join between table T and document set $\mathcal{D}$ on condition $|search(f, c_i, \epsilon)| > 0$, where c_i is

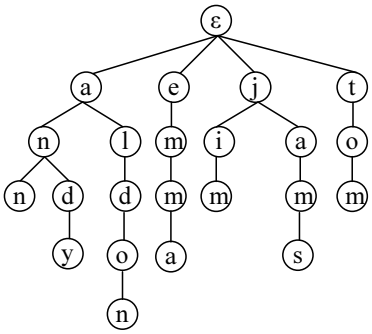


Fig. 15 Trie Index

a column of T. For a tuple t_j of T, its join result follows the format of (t_j, d_x), where $d_x \in search(f, t_j.c_i, \epsilon)$.

After integrating NLP operators into the relational model, we can write a hybrid join as:

SELECT count(*), name FROM $customer$, $email$
WHERE $|email.search(editdis, name, 0)| > 0$
GROUP BY $name$

The above query returns the number of emails which mention the name of a specific customer.

4.2.1 Trie Index

Hybrid join is similar to the string similarity join [12][20][24]. However, we are now comparing two huge string sets (strings in a column and all words appear in the document set). It is very expensive to compare each string from one set to all the strings in the other set. Therefore, we adopt an index-based approach. Motivated by the idea of trie-join [12], we can compute the edit distance between the strings with the help of trie tree index, which is widely used in the string comparison. Trie tree can be also used to compute the jaccard distance and hamming distance between strings. In the following discussion, we will use the edit distance as our example to illustrate the idea. Figure 15 shows the trie tree for string set {*ann, andy, aldon, emma, jim, jams, tom*}. To speed up the evaluation of NLP operators, the leaf node maintains an inverted index for the corresponding keyword, recording the documents containing the keyword and its positions in those documents.

Trie tree index can be efficiently built via *epiC* units. Figure 16 shows how the trie tree of Figure 15 is built. We first create some scan units which iterate all keywords in the documents. Then, keywords sharing the same prefix (e.g., {*an, al, em, ji, ja, to*}) is shuffled to the same merge unit where a local trie tree is built for each prefix. Finally, the roots of local trie trees are forwarded to one merge unit which will generate a global tree denoting all keywords in the document set.

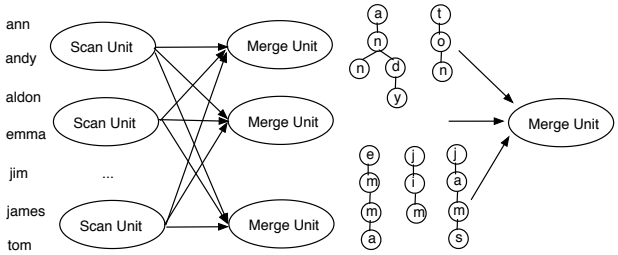


Fig. 16 Index Construction

Algorithm 1 `HybridJoin(Table` T`, int` c_idx`, DocumentSet` $\mathcal{D}$`)`

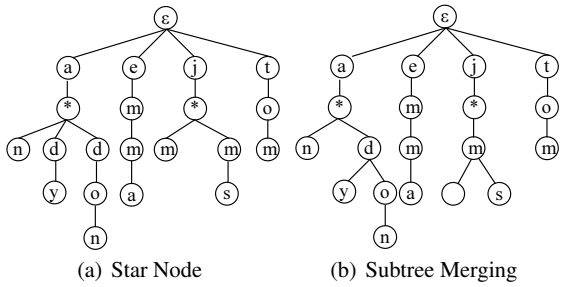
```
1: Set result = ∅
2: for ∀t_i ∈ T do
3:     TrieNode n = getTrieRoot(D)
4:     Set S_index = ∅
5:     IndexSearch(n, t_i[c_idx], S_index)
6:     for IndexEntry e : S_index do
7:         result.add(new Tuple(t_i, e.docID))
8: return result
```

Algorithm 2 `IndexSearch(TrieNode` n`, String` s`, Set` S_{index}`)`

```
1: if n.isLeaf and editdis(n, s) < ϵ then
2:     S_index.add(n.index)
3: else
4:     for TrieNodeIter n_i : n.Children do
5:         if !pruneViaLength(n_i, s.length) then
6:             for i=1 to s.length do
7:                 string s'=s.substring(0, i)
8:                 if dist(n_i, s') < ϵ then
9:                     IndexSearch(n_i, s, S_index)
10:                    break;
```

Algorithm 1 and 2 show how we progressively search the trie index to process the hybrid join. In Algorithm 1, we compare each tuple to the trie tree. If the returned inverted index entries are not empty, we will assemble the join results correspondingly. Algorithm 2 illustrates how we exploit the trie index to search the document set. Starting from the root node, we evaluate whether a child node is an active node of the query to prune the subtrees (line 7-10). If the child node is not a leaf node, it will recursively explore the tree in the same way. Otherwise, it returns the corresponding indexes as the matching results. We also record the minimal and maximal string lengths in each subtree and apply the length filter to prune the subtrees that obviously cannot generate any result (line 5).

Example 1 Suppose the query string is "ams" and we match it again the trie tree in Figure 15. We set ϵ to 2. The first level nodes {"a", "e", "j", "t"} are active nodes for "ams", as their distances to the prefix "a" are at most 1. Because "an" and "ja" are still active nodes to prefix "a", the search will continue in the corresponding two subtrees. "em" is the



(a) Star Node (b) Subtree Merging

Fig. 17 Compression Strategy

active node of prefix "am" and hence, we forward the query to its subtree. On the contrary, node "ji" and "to" are pruned. In the next step, all subtrees are pruned, except "jam" which is an active node of prefix "am". Finally, we get the only result "jams" when the search ends in the leaf level.

Ideally, if the trie tree is small, all *epiC* units can maintain a full copy of the index in memory. In that case, we can directly apply Algorithm 1 to process the join. However, in most cases, the size of trie tree is proportional to the size of document set. So we adopt two techniques to reduce the storage overhead, partitioning and compression. The idea of partitioning is straightforward and similar to the one shown in Figure 16. We use scan unit to retrieve all values in a column and then apply the hash function for the prefix to shuffle data to the join unit. Based on the prefix, each join unit only needs to load a subtree of the global tree into its memory to perform the string comparison. Hence, the memory overhead is reduced.

The idea of compression is to transform the trie tree into an approximate index. We show the details of compression algorithm as below.

4.2.2 Index Compression

To compress the trie tree, we introduce a specific node, the star node. The star node can match any character. For example, in Figure 17(a), the subtree of node *a* represents the string set {*a*n, a*dy, a*don*}. By introducing the star node, we reduce the size of trie tree in two ways:

1. Given a node *n*, we can replace all its child nodes with the star node. In this way, there is only one child of *n*, the star node. The number of nodes in trie tree is reduced by $g(n) - 1$, where $g(n)$ is the number of child nodes of *n*.
2. All subtrees of *n*'s child nodes are inherited by the new star node. Some subtrees share the same structure and hence, can be merged to reduce the storage cost.

In Figure 17(a), the subtrees of node *an* and *al* are migrated to the star node, *a**. As the subtree *dy* and *don* share the the same prefix *d*, we combine the subtrees in Figure 17(b). The same rule is applied to node *j**. However, to identify that there is a leaf node *j*m*, we add an empty node to the merged tree.

The star node can effectively reduce the size of trie tree. In an extreme case, we only need an L-length string **...* to denote the tree. L is the height of the tree. However, that will make the index useless, as it cannot be used to estimate the edit distance between strings. In other words, introducing the star nodes will generate false positives for the string matching using the trie tree index. The effect of adding the star node is described by the following theorem.

Theorem 1 *Given a node n, let S_L denote the strings in n's subtree (the leaf nodes of n). For a string s, we use $eidtdis(s, s_i)$ to represent the edit distance between s and a string $s_i \in S_L$. If we use a star node to replace n's child nodes, s_i is changed to s'_i. Let $dist(s, s'_i)$ be the new distance, we have $eidtdis(s, s_i) - eidtdis(s, s'_i) \leq 1$.*

Proof Let $s_{prefix}[c]s_{suffix}$ represent s_i , where s_{prefix} and s_{suffix} are substrings of s_i and c is a character between the two substrings. s_i is changed to $s_{prefix}[*]s_{suffix}$, after adding the star node. Based on how $eidtdis(s, s_i)$ is computed, we have three cases:

1. If c is removed in computing $eidtdis(s, s_i)$, then we can similarly remove the star character. Hence, $eidtdis(s, s_i) = eidtdis(s, s'_i)$.
2. If c is matched to a character of s, the star character also matches. We have $eidtdis(s, s_i) = eidtdis(s, s'_i)$.
3. If c is replaced with another character to match with s, we do not need to modify s', as star character can match any character. We have $eidtdis(s, s_i) = eidtdis(s, s'_i)+ 1$.

Using the compression techniques, we can significantly reduce the size of trie tree. However, the trade-off is the computation cost for verifying the false positvies. To design a good compression algorithm, we design a model to search for the optimal compression ratio. This is beyond the scope of this paper and hence, is discarded for space limitation.

4.3 Integrating with Relational Model

Using hybrid join, we can answer analytic queries for heterogeneous data. In particular, we implement our NLP operators and the hybrid join operator as the user-defined functions in *epiC*'s relational model. As a result, our first example query can be written as:

SELECT $max(count(*))$, $supplier.name$
FROM $product$, $supplier$, $email$
WHERE $sentiment(get(id, p, 100)) =' Negative'$ and
$supplier.id = product.supplier$
WITH id in $email.search(editdis, product.name, 2)$ and
p in $pos(id, editdis, product.name, 2)$
GROUP BY $supplier.name$

The idea is to perform the hybrid join between product names and emails to locate where a specific product is mentioned in the emails. We then retrieve the nearby text and apply the sentiment analysis to get the users' opinions. Finally, we join the supplier table with the product table to find the supplier with most complains.

Similarly, the second query is transformed into:

SELECT $\frac{count(*)}{|email.search(editdis,E.name,2)|}, M.name$
FROM $employee$ E, $employee$ M, $email$
WHERE $sentiment(get(id, p, 100)) =' Positive'$ and $E.manager = M.id$
WITH id in $email.search(editdis, E.name, 2)$ and p in $pos(id, editdis, E.name, 2)$
GROUP BY $M.name$

We apply the sentiment analysis for each email if it mentions our customer service. Then the satisfaction rate is estimated as the percentage of positive emails.

When users issue the above queries, we will parse them into a set of epiC units and process them one by one. More NLP operators are being implemented and this model allows *epiC* to perform more complex mining jobs.

5 Partitioning-Based Optimization

In basic *epiC* framework, each unit is considered as an actor, interacting with other actors via "emails", a predefined message type between actors. The unit loads data from the DFS based on the partitioning information specified in the email. After it completes its processing, the intermediate results are flushed back to the DFS which may be used by the other units as input. This design is very flexible, as the storage layer is completely transparent to the processing layer. However, it may also incur high I/O overheads, because units repeatedly read/write data from/into the DFS. To address this problem, we adopt an optimization technique by grouping a set of *Partitioning-Free* units together.

5.1 Partitioning-Free Unit

A unit is a partitioning-free unit, if its result is not affected by how data are partitioned among nodes. More formally,

Definition 3 Suppose unit U is deployed on N cluster nodes to process dataset $\mathcal{D}$. Let p denote the partitioning function and we use $U(p(\mathcal{D}, i))$ to represent the results of applying unit U to the partition of node i. Unit U is a partition-free unit, if for any two random partitioning functions p_1 and p_2,

$$\bigcup_{i=0}^{N-1} U(p_1(\mathcal{D}, i)) = \bigcup_{i=0}^{N-1} U(p_2(\mathcal{D}, i))$$

One property of partitioning-free units is that consecutive partitioning-free units can be grouped together. So instead of processing them one by one, we can link them together and apply the batch processing technique.

Theorem 2 *Suppose U_1 and U_2 are two partitioning-free units. For any two partitioning functions p_1 and p_2, we have*

$$\bigcup_{i=0}^{N-1} U_1(U_2(p_1(\mathcal{D}, i))) = \bigcup_{i=0}^{N-1} U_1(U_2((p_2(\mathcal{D}, i)))$$

Proof Based on the definition, we have

$$\bigcup_{i=0}^{N-1} U_2(p_1(\mathcal{D}, i)) = \bigcup_{i=0}^{N-1} U_2(p_2(\mathcal{D}, i))$$

So

$$U_1(\bigcup_{i=0}^{N-1} U_2(p_1(\mathcal{D}, i))) = U_1(\bigcup_{i=0}^{N-1} U_2(p_2(\mathcal{D}, i)))$$

We define two new partitioning functions. The first partitions the data into $\bigcup_{i=0}^{N-1} U_2(p_1(\mathcal{D}, i))$, $\emptyset$,...,$\emptyset$. And the second partitions the data into $U_2(p_1(\mathcal{D}, 0))$, ..., $U_2(p_1(\mathcal{D}, N-1))$. Applying the definition again, we have

$$U_1(\bigcup_{i=0}^{N-1} U_2(p_1(\mathcal{D}, i))) = \bigcup_{i=0}^{N-1} U_1(U_2(p_1(\mathcal{D}, i)))$$

Similarly,

$$U_1(\bigcup_{i=0}^{N-1} U_2(p_2(\mathcal{D}, i))) = \bigcup_{i=0}^{N-1} U_1(U_2(p_2(\mathcal{D}, i)))$$

So the theorem is correct.

For example, in relational model, *select* unit and *project* unit are partitioning-free units, while *aggregate* unit is not, because different partition strategies will generate different aggregation results. In *epiC*, users can explicitly define a unit as partitioning-free units. So the scheduler can apply more aggressive plans. In particular, suppose U_1, U_2, ..., U_k are consecutive partitioning-free units. Namely, U_i is processed before U_{i+1}. Instead of scheduling each unit to a specific node, we group all units together as $\bar{U}$ which is used as the scheduling task by merging all the source codes of U_1 to U_k. The result of U_i is directly streamed to U_{i+1} for processing. We avoid the cost of writing back the internal results to the DFS.

5.2 Function Transformation

Most units are not partitioning-free units. However, we can apply the function decomposition technique to transform a unit into partitioning-free unit. Let f denote the processing logic defined in the *run* function of a unit. We use x_1,..,x_n to represent its input (the partition that assigns to the node).

We first show that if f is a symmetric rational function[2], it can be represented by a set of symmetric polynomial functions. Then, all symmetric polynomial functions can be decomposed into some elementary functions, which can be transformed into partitioning-free functions.

Theorem 3 *The symmetric rational function $f(x_1, ..., x_n)$ can be represented as a fraction of two symmetric polynomial functions.*

Proof If function $P(x_1, ..., x_n)$ is a symmetric polynomial function, function $Q(x_1, ..., x_n)$ must be a symmetric polynomial function too and vice versa. Therefore, we consider the case where both functions are not symmetric polynomial functions. For all n variables, there are $N = n!$ permutations. let Q^i denote function Q in the ith permutation. We have

$$f(x_1, ..., x_n) = \frac{P(x_1, ..., x_n)Q^1(x_1, ..., x_n)...Q^{N-1}(x_1, ..., x_n)}{Q^1(x_1, ..., x_n)Q^2(x_1, ..., x_n)...Q^N(x_1, ..., x_n)}$$

As the denominator covers all possible permutations, it is, in fact, a symmetric polynomial function. Therefore, the numerator must be a symmetric polynomial function, because f is a symmetric rational function. In this way, f is represented as the fraction of two symmetric polynomial functions.

One most important type of symmetric polynomial functions are elementary symmetric polynomial functions. For a symmetric polynomial function f with n variables, we define $n + 1$ elementary symmetric polynomial functions. In particular, the ith function is represented as:

$$e_i(x_1, ..., x_n) = \sum_{1 \le j_1 < j_2 < ... < j_i \le n} x_{j_1} x_{j_2} ... x_{j_i}$$

j_1 to j_i is an i-permutation for the variables. As an example,

$$e_2(x_1, ..., x_n) = \sum_{1 \le j_1 < j_2 \le n} x_{j_1} x_{j_2}$$

To handle the special case, we also define $e_0(x_1, ..., x_n) = 1$ and $e_j(x_1, ..., x_n) = 0$ for $j < 0$.

It was proved that all symmetric polynomial functions can be transformed into the polynomial expressions with elementary functions. The complete proof can be found in [25]. We give a brief overview, as our function decomposition algorithm adopts the same technique in the proof.

Let $(i_1, i_2, ..., i_n)$ and $(j_1, j_2, ..., j_n)$ denote two permutations of the variables. We say $(i_1, i_2, ..., i_n) > (j_1, j_2, ..., j_n)$, if the first non-zero difference in $i_1 - j_1$,...,$i_n - j_n$ is positive. We first define the degree of the polynomial function as follows.

Definition 4 Degree of Polynomial Function
The degree of symmetric polynomial function $f(x_1, ..., x_n)$ ($deg(f)$) is defined as the largest permutation $(i_1, i_2, ..., i_n)$, satisfying that the co-efficient of $x_1^{i_1} x_2^{i_2} ... x_n^{i_n}$ is non-zero.

For example, $deg(e_1) = (1, 0, ..., 0)$, $deg(e_2) = (1, 1, ..., 0)$ and $deg(e_n) = (1, 1, ..., 1)$. It is easy to verify that

$$deg(e_1 \times e_2) = deg(e_1) + deg(e_2)$$

The proof of symmetric polynomial function, in fact, tries to show that we can always reduce the degree of the function by decomposing it into some elementary functions.

Theorem 4 *The symmetric polynomial function can be transformed into the polynomial expressions via elementary functions.*

Proof Suppose the degree of a symmetric polynomial function $f(x_1, ..., x_n)$ is $(i_1, ..., i_n)$. By the definition of degree, we can find a term in f like $ax_1^{i_1}...x_n^{i_n}$ ($a \neq 0$). As f is symmetric, we can find all permutations of that term, which are represented as a symmetric function p.

$$p(x_1, ..., x_n) = ae_1^{i_1 - i_2} e_2^{i_2 - i_3} ... e_{n-1}^{i_{n-1} - i_n} e_n^{i_n}$$

The degree of p is computed as:

$$\begin{aligned} deg(p) &= (i_1 - i_2)deg(e_1) + (i_2 - i_3)deg(e_2) + ... + i_n deg(e_n) \\ &= (i_1 - i_2, ..., 0) + (i_2 - i_3, i_2 - i_3, ..., 0) + ... + (i_n, ..., i_n) \\ &= (i_1, ..., i_n) \end{aligned}$$

Namely, p has the same degree as f. We can now transform f into $\bar{f}$ as:

$$\bar{f}(x_1, ..., x_n) = f(x_1, ..., x_n) - p(x_1, ..., x_n)$$

As both f and p are symmetric polynomial functions, f' is also a symmetric polynomial function. Moreover, because p's degree is equal to f's degree, $\bar{f}'s$ degree should be smaller than f. In this way, we reduce the degree of a symmetric function. If we recursively apply the above technique to $\bar{f}$, we can further reduce the degree. When the degree reaching $(0, 0, ..., 0)$, we have decomposed function f into a polynomial expressions via the elementary symmetric polynomial functions.

We now show that all elementary functions can be decomposed. Recall that given a function f, we want to find a set of functions satisfying

$$f(S_0, S_1) = h_0(h_1(S_0), h_2(S_0), ..., h_n(S_0), S_1)$$

$S_0 = \{x_1, ..., x_k\}$ and $S_1 = \{x_{k+1}, ..., x_n\}$ denote the two random partitions, respectively. To simplify the problem, we

[2] http://en.wikipedia.org/wiki/Symmetric_polynomial

study a special case, where S_1 only contains one tuple x_n and all the rest are in S_0.

If we consider the elementary function, the problem is transformed into computing $e_i(x_1, ..., x_n)$ using functions of $x_1, ..., x_{n-1}$. In fact, $e_i(x_1, ..., x_n)$ can be divided into two parts, one with tuple x_n and one without tuple x_n.

$$\begin{aligned} e_i(x_1, ..., x_n) &= \sum_{1 \le j_1 < j_2 < ... < j_i \le n} x_{j_1} x_{j_2} ... x_{j_i} \\ &= \sum_{1 \le j_1 < j_2 < ... < j_{i-1} < n, j_i = n} x_{j_1} x_{j_2} ... x_{j_i} + \\ &\quad \sum_{1 \le j_1 < j_2 < ... < j_i \le n-1} x_{j_1} x_{j_2} ... x_{j_i} \\ &= x_n e_{i-1}(x_1, ..., x_{n-1}) + e_i(x_1, ..., x_{n-1}) \end{aligned}$$

Therefore, we create two function h_1 and h_2 as:

$$h_1(x_1, ..., x_{n-1}) = e_{i-1}(x_1, ..., x_{n-1})$$

$$h_2(x_1, ..., x_{n-1}) == e_i(x_1, ..., x_{n-1})$$

Correspondingly, let $S_0 = \{x_1, ..., x_{n-1}\}$. h_0 function is defined as:

$$h_0(h_1(S_0), h_2(S_0), \{x_n\}) = x_n h_1(S_0) + h_2(S_0)$$

In this way, given two random partitions S_0 and S_1 (S_1 only contains one element), we can successfully compute the final results. So the elementary functions are partitioned-free functions, if one partition only contains one element.

Now, let us consider the more complex case, where we partition the data into two random sets with size $n - k$ and k. We have the following theorem:

Theorem 5 *All elementary functions can be transformed into their decomposed forms for two random partitions.*

Proof Given an elementary function e_i, suppose input data is partitioned into $\{x_1, ..., x_{n-k}\}$ and $\{x_{n-k+1}, ..., x_n\}$. We can decompose e_i as:

$$\begin{aligned} e_i(x_1, ..., x_n) &= x_n e_{i-1}(x_1, ..., x_{n-1}) + e_i(x_1, ..., x_{n-1}) \\ &= x_n x_{n-1} e_{i-2}(x_1, ..., x_{n-2}) + (x_n + x_{n-1}) \\ &\quad \times e_{i-1}(x_1, ..., x_{n-2}) + e_i(x_1, ..., x_{n-2}) \\ &= x_n x_{n-1} e_{i-2}(x_1, ..., x_{n-2}) + \\ &\quad e_1(x_{n-1}, x_n) e_{i-1}(x_1, ..., x_{n-2}) + \\ &\quad e_0(x_{n-1}, x_n) e_i(x_1, ..., x_{n-2}) \\ &= \Pi_{j=n-k+1}^{n} x_j e_{i-k}(x_1, ..., x_{n-k}) + \\ &\quad \sum_{j=0}^{k-1} e_{i-j}(x_1, ..., x_{n-k}) e_j(x_{n-k+1}, ..., x_n) \end{aligned}$$

To transform the function into the recursive form, we create $k + 1$ functions. The jth function is defined as:

$$h_j = e_{i-j+1}(x_1, ..., x_{n-k})$$

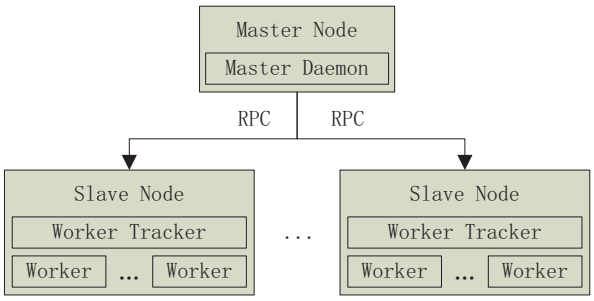


Fig. 18 The architecture of an *epiC* cluster

The combining function $h_0(h_1, ..., h_{k+1}, \{x_{n-k+1}, ..., x_n\})$ is constructed as an expression of elementary functions:

$$\Pi_{j=n-k+1}^{n} x_j h_{k+1} + \sum_{j=1}^{k} h_j e_j(x_{n-k+1}, ..., x_n)$$

In this way, we can generate the decomposed form for all elementary functions.

Currently, we cannot automatically detect whether the *run* function of a unit is a symmetric rational function. Instead, users are required to extend their units to implement the symmetric rational function interface. Then, the system can decompose the *run* function into elementary functions and consider the unit as partitioning-free unit, where the optimized scheduling algorithm is applied.

6 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 a 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 18. *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∼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.

6.1 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 eval-

uation 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 hanged during the processing of TTL RPC call. Thus the threading model is scalable to thousands of concurrent TTL RPC calls.

6.2 Fault Tolerance

Like all single master cluster architecture, *epiC* is designed to 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 whether 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 recovery 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 the 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.

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

$$E_R = T_t \times P + 2T_t \times (1 - P) \tag{1}$$

$$E_B = (T_t + T_u) \times P + T_d \times (1 - P) \tag{2}$$

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. 3 for determining which completed tasks should be backed up. The master iterates over 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).

Algorithm 3 Generate the list of completed tasks to backup

Require: the worker tracker list W
Ensure: 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: $L \leftarrow L \cup \{t\}$

7 Experiments

We evaluate the performance of *epiC* on different kinds of data processing tasks, including unstructured data processing, relational data processing, graph processing and heterogenous data processing. We benchmark *epiC* against Hadoop, AsterixDB and SparkSQL [3] for processing unstructured data (i.e., text data), relational data. We also benchmark *epiC* with Hadoop for processing heterogenous data (i.e., mixture of text data and relational data) and GPS [30], an open source implementation of Pregel [26] for graph processing, respectively. For all experiments, the results are reported by averaging six runs.

7.1 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 node for all systems (i.e., Hadoop, Spark, AsterixDB, GPS and epiC). The rest of other nodes act as slave/worker nodes. For scalability benchmark, we vary the number of slave nodes from 1, 4, 16, to 64.

7.2 System Settings

In our experiments, we configure benchmark 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 [21], 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 enable compression of Map output and set the JVM reuse number to -1.
2. For SparkSQL, we configure Spark (i.e., the underline execution engine) as follows. Each slave node runs a single worker instance with 4 executors. The memory of each executor is 5GB. We turn on compression for shuffle, broadcast and RDD with lz4 compression algorithm.
3. For AsterixDB, we setup the system according to the instructions presented in its official website without further tuning.
4. 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).
5. For GPS, we employ the default settings of the system without further tuning.

7.3 Benchmark Tasks and Datasets

7.3.1 Benchmark Tasks

The benchmark consists of six tasks: Grep, TeraSort, TPC-H Q3 and Q5, PageRank and hybrid join. 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 and Q5 task is a standard benchmark query in TPC-H benchmark and is presented in Section 3.2. The PageRank algorithm [28] is an iterative graph processing algorithm. We refer the readers to the original paper [28] for the details of the algorithm. The hybrid join task joins a tweet dataset with a Item table (i.e., relational table) using the techniques presented in Section 4.2. The join task calculates the number of tweets containing negative comments for each brand in Item table. For all benchmark tasks, we generate data in HDFS in plain text format and

[3] SparkSQL is Spark's module for performing SQL queries with Spark execution engine.

configure all systems to process those data in-place. For AsterixDB and SparkSQL, this in-place processing is achieved by using `create external table` statement. Unfortunately, we cannot perform all benchmark tasks on every system. For AsterixDB, the built-in `create external table` statement cannot create tables for TeraSort and Grep [4] datasets. For SparkSQL, it fails to perform Q5 since the query produces very large intermediate results which cannot fit into memory. We further noticed, in a recent blog, that the underline engine (i.e., Spark) is specially tuned and improved for TeraSort benchmark [5]. However, those improvements are not available in the Spark's current stable release for the time being. Therefore, we intentionally remove the results of aforementioned benchmark tasks for AsterixDB and SparkSQL.

7.3.2 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 sort the input records based on their keys. We perform both scale up and speed up benchmark for these two tasks. For scale up benchmark, we generate two datasets: a small dataset of 1GB data per-node and a large dataset of 10GB data per-node for each cluster size (i.e., 1, 4, 16, 64). For speed up benchmark, we generate a 64GB dataset for the whole cluster.

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[6]. 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.

We use the data generator shipped with BigFrame benchmark [7] to generate the heterogenous dataset for hybrid join. The data generator can generates both relational data and text data. The relational data is compliant to the standard TPC-DS benchmark dataset and the text data are syntactically generated tweets. We configure the data generator to produce 10GB data per node. Among those data, 80% data are tweets (i.e., text data) and 20% data are relational data.

[4] This is because AsterixDB's create table statement requires that the input record contains a delimiter character to separate fields. However, all valid delimiter character may appear in the key field of Grep and TeraSort.

[5] http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

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

[7] https://github.com/bigframeteam/BigFrame

7.4 The Grep Task

Figure 19 and Figure 20 present the performance of employing *epiC*, SparkSQL and Hadoop for performing Grep task with the cold file system cache and the warm file system cache settings in 1GB per-node datasets, respectively.

In the cold file system cache setting (Figure 19), the performance of *epiC* is similar to SparkSQL and is 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 processes for running the map tasks and reduce tasks. 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$\sim$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 [7].

In the warm file system cache setting (Figure 20), 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. We also found *epiC* is about 1.7 faster than SparkSQL. We again attribute the performance improvements of *epiC* to its efficient task assignment scheme.

Figure 21 shows the performance of employing three systems to perform Grep task on 10GB data per-node settings. The performance is similar for three systems. This is because, in this setting, the startup cost is amortized by processing large dataset. Figure 22 presents the results of speed up benchmark for Grep task. In this benchmark, the size of dataset is fixed to 64GB and we vary the number of processing nodes to process the dataset. It can be seen from Figure 22 that all three systems can achieve near perfect linear speed up in this task.

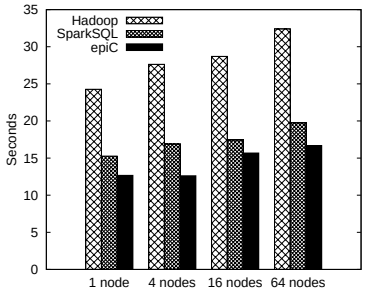


Fig. 19 Grep task with cold file system cache – 1GB per-node

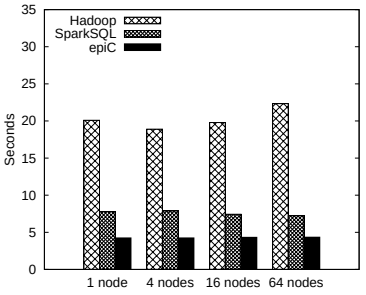


Fig. 20 Grep task with warm file system cache – 1GB per-node

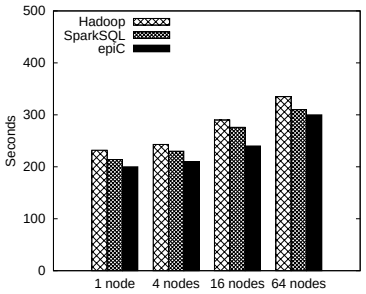


Fig. 21 Grep task – 10GB per-node

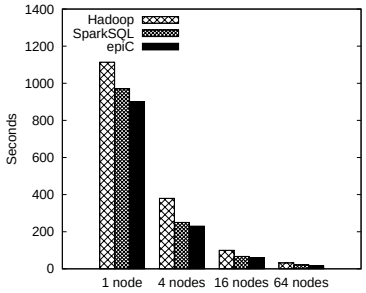


Fig. 22 Grep task – 64GB

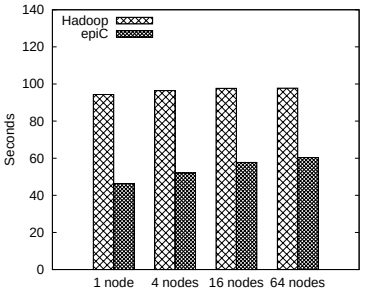


Fig. 23 TeraSort task with cold file system cache – 1GB per-node

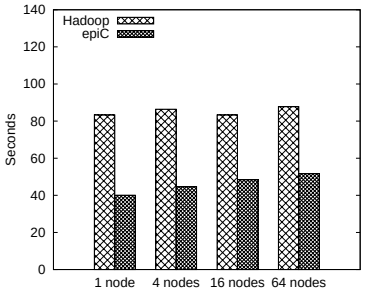


Fig. 24 TeraSort task with warm file system cache – 1GB per-node

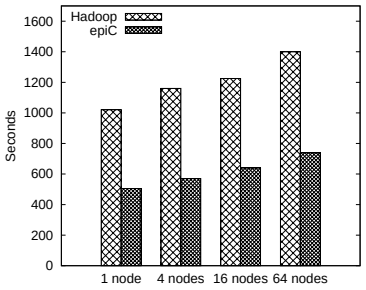


Fig. 25 TeraSort task – 10GB per-node

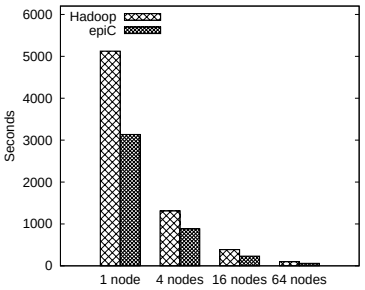


Fig. 26 TeraSort task – 64GB

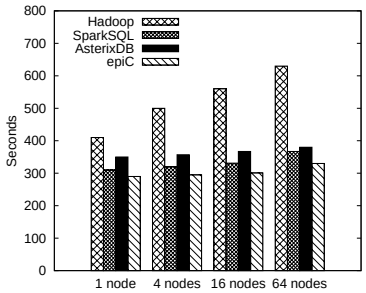


Fig. 27 Results of TPC-H Q3

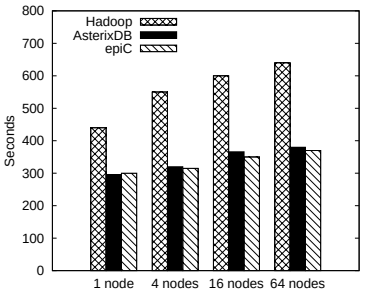


Fig. 28 Results of TPC-H Q5

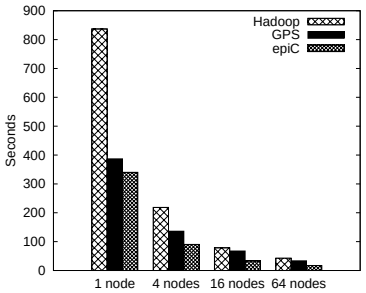


Fig. 29 Results of PageRank

7.5 The TeraSort Task

Figure 23 and Figure 24 show the performance of the two systems (*epiC* and Hadoop) for performing TeraSort task on 1GB data per-node settings. Figure 25 presents the results of employing the two systems on 10GB data per-node setting where intermediate data cannot fit into memory. Figure 26 presents the results for speed up benchmark where the two systems are employed to sort 64GB data with different number of processing nodes. Overall, *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. Sort-

ing 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. *epiC* is able to start shuffling 5∼8 seconds earlier than Hadoop.

Compared to the performance of cold cache setting (Figure 23), both *epiC* and Hadoop do not run much faster in the warm cache setting (Figure 24); 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.

7.6 The TPC-H Q3 Task

Figure 27 presents the results of employing *epiC*, AsterixDB, SparkSQL and Hadoop to perform TPC-H Q3 under cold file system cache [8]. 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 27 shows the results. The performance of *epiC* is similar to AsterixDB and SparkSQL and is 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 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.

[8] For TPC-H Q3 and Q5 task, PageRank task and the Hrbrid join task, all systems cannot get a significant performance improvement from cache. Therefore, we remove warm cache results to save space.

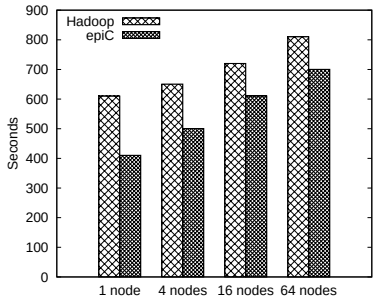


Fig. 30 Results of Hybrid join

7.7 The TPC-H Q5 Task

Figure 28 presents the results of applying *epiC*, AsterixDB and Hadoop to perform TPC-H Q5 task. SparkSQL fails to complete this task since the intermediate results produced by the query are too large to fit into memory. We found that SparkSQL repeatedly swaps RDD from memory into disk and eventually fails after a number of retries. Disk-based systems (i.e., *epiC*, AsterixDB and Hadoop) complete the task since their scalability is not constrained by available memory. The performance of *epiC* and AsterixDB are very similar. In certain settings, AsterixDB is slightly faster. We attribute the excellent performance of AsterixDB to its highly optimized query plan. Both systems (*epiC* and AsterixDB) run about twice faster than Hadoop.

7.8 The PageRank Task

This experiment compares three systems in performing the PageRank task. The GPS implementation of PageRank algorithm is identical to [26]. 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 29 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 Hadoop repeatedly creates new *mappers* and *reducers* for each job.

7.9 The Hybrid Join Task

This experiment evaluates the performance of employing *epiC* and Hadoop to perform hybrid join. The hybrid join

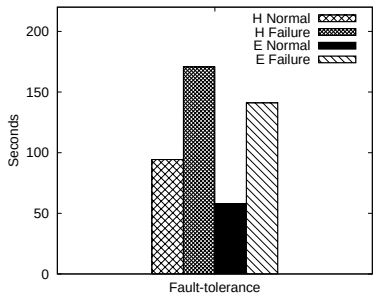


Fig. 31 Fault tolerance experiment on a 16 node cluster

task calculates the number of tweets containing negative comments for each brand in Item table. The SQL-like command to illustrate the hybrid join task is as follows:

SELECT $I.brand, count(T.tweetID)$
FROM $Tweet$ T, $Item$ I
WHERE $sentiment(T.text) =' Negative'$ and
$T.search(editdis, I.product\_name, 0)$
GROUP BY $I.brand$

The *epiC* job for this task consists of three units. The first unit scan T and I, filtering positive and neutral tweets. The second unit performs the join and the final unit conducts the aggregation. We also implement the same evaluation scheme using two MapReduce jobs on Hadoop.

Figure 30 presents the results of this experiment. The performance gap between *epiC* and Hadoop is small, about $40\% \sim 50\%$. This is because the hybrid join task is both CPU intensive and I/O intensive. We found that the CPU cost of this task is dominated by evaluating sentiment of tweets and joining tweets with Item records. Therefore, even *epiC* is more efficient in scheduling and shuffling than Hadoop, the overall performance of the two systems is similar.

7.10 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 recovery. The experiment is launched at a 16 node cluster. We simulate 4 machine failures at 50% job completion.

Figure 31 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).

8 Related Work

Big Data processing systems can be classified into the following categories: 1) Parallel Databases, 2) MapReduce based systems, 3) DAG based data processing systems, 4) Actor-like systems and 5) hybrid systems. A comprehensive survey could be found in [23], and a new benchmark called BigBench [15], was also recently proposed to evaluate and compare the performance of different big data processing systems.

The research on parallel databases started in the late 1980s [10]. Pioneering research systems include Gamma [9], and Grace [14]. 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 [32]. Parallel databases have been shown to scale to at least peta-byte dataset but with a relatively high cost on hardware and software [3]. 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 [33]. 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 [8]. 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 [36] [34]. 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 [3][22]. 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 [7]. 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 [18] [19]. This work is close to ours since Dryad is intended for a general purpose data parallel programming framework. Our work is different from that of Dryad - our concurrent programming model is entirely independent of communication patterns while Dryad enforces processing units to transfer data through DAG.

The the concept of Actor was originally proposed for simplifying concurrent programming [16]. Recently, systems like Storm [2] and S4 [27] 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.

HadoopDB [3] and PolyBase [11] are new systems for handling the the variety challenge of Big Data. The difference between these systems and ours is that the two systems adopt a hybrid architecture and use a combination of a relational database system and Hadoop to process Big Data. Based on the data type, a split execution strategy is employed to split the whole data analytical job into database and Hadoop for processing. Our system, on the other hand, do not employ the split execution strategy and use a single system to process all types of data.

9 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 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*. We also show how users can leverage *epiC* to process heterogeneous data, and we discuss a novel partition-based optimization technique adopted in *epiC*. The benchmarking of *epiC* against Hadoop and GPS confirms its efficiency.

10 Acknowledgments

This work was supported by the National Research Foundation, Prime Ministers Office, Singapore under Grant No. NRF-CRP8-2011-08.

References

1. The hadoop offical website. http://hadoop.apache.org/.
2. The storm project offical website. http://storm-project.net/.
3. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1), Aug. 2009.
4. 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*, 2010.
5. J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA*, 1997.
6. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *VLDB*, 3(1-2), Sept. 2010.
7. 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. 2011.
8. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), Jan. 2008.
9. 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*, 1986.
10. D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6), 1992.
11. D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. In *SIGMOD Conference*, 2013.
12. J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.
13. J. R. Finkel, T. Grenager, and C. D. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *ACL 2005, 43rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, 25-30 June 2005, University of Michigan, USA*, 2005.
14. S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine grace. In *VLDB*, 1986.
15. A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *SIGMOD Conference*, 2013.
16. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, 1973.
17. M. Hu and B. Liu. Mining and summarizing customer reviews. In *SIGKDD*, pages 168–177, 2004.
18. 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), Mar. 2007.
19. M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD Conference*, 2009.
20. J. Jestes, F. Li, Z. Yan, and K. Yi. Probabilistic string similarity joins. In *SIGMOD*, pages 327–338, 2010.
21. D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: an in-depth study. *PVLDB*, 3(1-2), Sept. 2010.
22. 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), Sept. 2011.

23. F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Comput. Surv. (to appear)*, 46(3), 2014.
24. G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: a partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264, Nov. 2011.
25. I. Macdonald. *Symmetric Functions and Hall Polynomials, second ed.* Oxford: Clarendon Press, 1998.
26. 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*, 2010.
27. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177, 2010.
28. L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, November 1999.
29. 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*, 2009.
30. S. Salihoglu and J. Widom. Gps: A graph processing system. In *SSDBM*, 2013.
31. R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics*, 9, Dec. 2004.
32. 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*, 2005.
33. X. Su and G. Swart. Oracle in-database hadoop: when mapreduce meets rdbms. In *SIGMOD Conference*, 2012.
34. 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), Aug. 2009.
35. S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *SoCC*, 2011.
36. H. Yang, A. Dasdan, R. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD Conference*, 2007.