Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters

Hung-chih Yang, Ali Dasdan
Yahoo!
Sunnyvale, CA, USA
{hcyang,dasdan}@yahoo-inc.com

Ruey-Lung Hsiao, D. Stott Parker
Computer Science Department, UCLA
Los Angeles, CA, USA
{rlhsiao,stott}@cs.ucla.edu

ABSTRACT

Map-Reduce is a programming model that enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines. Through a simple interface with two functions, map and reduce, this model facilitates parallel implementation of many real-world tasks such as data processing for search engines and machine learning.

However, this model does not directly support processing multiple related heterogeneous datasets. While processing relational data is a common need, this limitation causes difficulties and/or inefficiency when Map-Reduce is applied on relational operations like joins.

We improve Map-Reduce into a new model called Map-Reduce-Merge. It adds to Map-Reduce a Merge phase that can efficiently merge data already partitioned and sorted (or hashed) by map and reduce modules. We also demonstrate that this new model can express relational algebra operators as well as implement several join algorithms.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; H.2.4 [**Database Management**]: Systems—*Parallel databases; Relational databases*

General Terms

Design, Languages, Management, Performance, Reliability

Keywords

Cluster, Data Processing, Distributed, Join, Map-Reduce, Map-Reduce-Merge, Parallel, Relational, Search Engine

1. INTRODUCTION

Search engines process and manage a vast amount of data collected from the entire World Wide Web. To do this task efficiently at reasonable cost, instead of relying on generic DBMS, they are usually built as customized parallel data processing systems and deployed on large clusters of shared-nothing commodity nodes. In [3], based on his experience as Inktomi (now part of Yahoo!) co-founder, Eric Brewer advocated that building novel data-intensive systems (e.g., search engines) should "apply the principles of databases, rather than the artifacts." It was because DBMS are usually overly generalized with many features that some can be unnecessary overhead for specific applications like search engine. Hence, search engine companies have developed and operated on "simplified" distributed storage and parallel programming infrastructures. These include Google's File System (GFS) [10], Map-Reduce [6], BigTable [4]; Ask.com's Neptune (using the Data Aggregation Call (DAC) framework) [5]; and Microsoft's Dryad [13]. Yahoo! also has similar infrastructures. These infrastructures adopt only a selected subset of database principles, hence are "simplified," but they are sufficiently generic and effective that they can be easily adapted to data processing in search engines, machine learning, and bioinformatics. Following these useful but proprietary (non-publicly released) infrastructures, Hadoop[1] is an open-source implementation, which is reminiscent of GFS and Map-Reduce, and is released under the umbrella of the Apache Software Foundation.

Common to these infrastructures is the refactoring of data processing into two primitives: (a) a *map* function to process input key/value pairs and generate intermediate key/values, and (b) a *reduce* function to merge all intermediate pairs associated with the same key and then generate outputs. The DAC framework has similar primitives, called *local* and *reduce*. These primitives allow users to develop and run parallel data processing tasks without worrying about the nuisance details of coordinating parallel sub-tasks and managing distributed file storage. This abstraction can greatly increase user productivity [6].

Though sufficiently generic to perform many real world tasks, the Map-Reduce framework is best at handling homogeneous datasets. As indicated in [15], *joining* multiple heterogeneous datasets does not quite fit into the Map-Reduce framework, although it still can be done with extra Map-Reduce steps. For example, users can map and reduce one dataset and read data from other datasets on the fly. In short, processing data relationships, which is what RDBMS excel at, is perhaps not Map-Reduce's strong suit.

For a search engine, many data processing problems can be easily solved using the Map-Reduce framework, but there are some tasks that are best modeled as joins. For ex-

SIGMOD'07, June 12–14, 2007, Beijing, China.

ample, a search engine usually stores crawled URLs with their contents in a *crawler* database, inverted indexes in an *index* database, click or execution logs in a variety of *log* databases, and URL linkages along with miscellaneous URL properties in a *webgraph* database. These databases are gigantic and distributed over a large cluster of nodes. Moreover, their creation takes data from multiple sources: *index* database needs both *crawler* and *webgraph* databases; a *webgraph* database needs both a *crawler* and a previous version of the *webgraph* database.

To handle these tasks in the Map-Reduce framework, developers might end up writing awkward map/reduce code that processes one database while accessing others on the fly. Alternatively they might treat these databases as homogeneous inputs to a Map-Reduce process but encode heterogeneity with an additional *data-source* attribute in the data and extra conditions in the code.

Processing data relationships is ubiquitous, especially in enterprise information systems. One major focus of the extremely popular relational algebra and RDBMS is to model and manage data relationships efficiently. Besides search engine tasks, another scenario of applying a *join-enabled* Map-Reduce framework is to join large databases across application, company, or even industry boundaries. For example, both airliners and hotel chains have huge databases. Joining these databases can permit data miners to extract more comprehensive rules than they could individually. While many traditional (shared- or shared-nothing, cluster-based or mass parallel) RDBMS have been deployed in enterprise OLAP systems, a join-enabled Map-Reduce system can provide a highly parallel yet cost effective alternative.

Based on these observations, we believe that one important improvement for the Map-Reduce framework is to include *relational algebra* in the subset of the database principles it upholds. That is, it should be further extended to support relational algebra primitives without sacrificing its existing generality and simplicity. The chief focus and contribution of this paper is this extension. We extend the Map-Reduce framework (shown in Fig. 1) to the Map-Reduce-Merge framework (shown in Fig. 2). This new framework introduces a naming and configuring scheme that extends Map-Reduce to processing heterogeneous datasets simultaneously. It also adds a new Merge phase that can join reduced outputs.

To recap, the contributions of this paper are as follows:

- Abiding by Map-Reduce's "simplified" design philosophy, we augment the Map-Reduce framework by adding a Merge phase, so that it is more efficient and easier to process data relationships among heterogeneous datasets.

  Note that, while Map-Reduce tasks are usually stacked to form a linear user-managed workflow, adding a new Merge primitive can introduce a variety of hierarchical workflows for one data processing task. A Map-Reduce-Merge workflow is comparable to a RDBMS execution plan, but developers can embed programming logic in it and it is designed specifically for parallel data processing.

- In a parallel setting, relational operators can be modeled using various combinations of the three functional-programming-based primitives: *map*, *reduce*, and *merge*. With proper configurations, these three primitives can be used to implement the parallel versions of several join algorithms: *sort-merge*, *hash*, and *block nested-loop*.

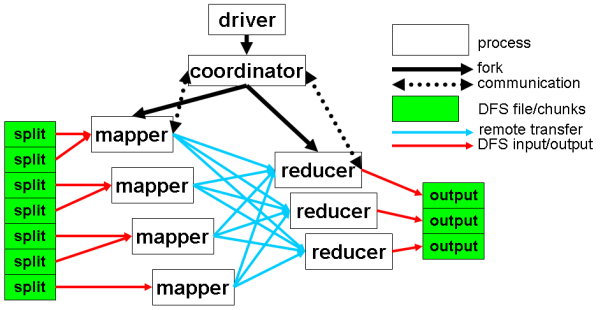

Figure 1: Data and control flow for Google's Map-Reduce framework. A driver program initiates a coordinator process. It remotely forks many mappers, then reducers. Each mapper reads file splits from GFS, applies user-defined logic, and creates several output partitions, one for each reducer. A reducer reads remotely from every mapper, sorts, groups the data, applies user-defined logic, and sends outputs to GFS.

In [12], Jim Gray et al. emphasized that there must be a "synthesis of database systems and file systems," as "file systems grow to petabyte-scale archives with billions of files." This vision not only applies to scientific data management, the focus of [12], but also applies to any data-intensive system such as a search engine. As stated in [12], Google's Map-Reduce framework not only abstracts parallel programming from data processing tasks, but it also abstracts files as just "containers for data" through its set-oriented model. This "synthesis" vision echoes Brewer's "principle" idea as Map-Reduce/GFS provides both views a great example of *database-oriented* data processing. Jim Gray et al. also envisioned that simplified data/programming models like Google's Map-Reduce could evolve into more general ones in the coming decade. Our Map-Reduce-Merge proposal is a step towards that goal.

2. MAP-REDUCE

Google's Map-Reduce programming model and its underlying Google File System (GFS) focus mainly to support search-engine-related data processing. It has a simple programming interface, and, though seemingly restricted, it is actually quite versatile and generic. It can extend to data processing tasks beyond the search-engine domain. According to [6], it has also been heavily applied within Google for data-intensive applications such as machine learning.

2.1 Features and Principles

Contrary to traditional data processing and management systems, Map-Reduce and GFS are based on several unorthodox assumptions and counter-intuitive design principles:

- **Low-Cost Unreliable Commodity Hardware**: Instead of using expensive, high-performance, and reliable symmetric multiprocessing (SMP) or massively

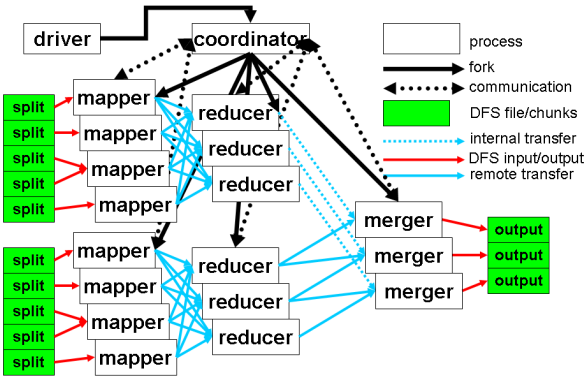

Figure 2: Data and control flow for the Map-Reduce-Merge framework. The coordinator manages two sets of mappers and reducers. After these tasks are done, it launches a set of mergers that read outputs from selected reducers and merge them with user-defined logic.

parallel processing (MPP) machines equipped with high-end network and storage subsystems, most search engines run on large clusters of commodity hardware. This hardware is managed and powered by open-source operating systems and utilities, so that the cost is low.

- **Extremely Scalable RAIN Cluster**: Instead of using centralized RAID-based SAN or NAS storage systems, every Map-Reduce node has its own local off-the-shelf hard drives. These nodes are loosely coupled in rackable systems connected with generic LAN switches. Loose coupling and shared-nothing architecture make Map-Reduce/GFS clusters highly scalable. These nodes can be taken out of service with almost no impact to still-running Map-Reduce jobs. These clusters are called *Redundant Array of Independent (and Inexpensive) Nodes* (RAIN) [18]. GFS is essentially a RAIN management system.

- **Fault-Tolerant yet Easy to Administer**: Due to its high scalability, Map-Reduce jobs can run on clusters with thousands of nodes or even more. These nodes are not very reliable. At any point in time, a certain percentage of these commodity nodes or hard drives will be out of order. GFS and Map-Reduce are designed not to view this certain rate of failure as an anomaly; instead they use straightforward mechanisms to replicate data and launch backup tasks so as to keep still-running processes going. To handle crashed nodes, system administrators simply take crashed hardware off-line. New nodes can be plugged in at any time without much administrative hassle. There is no complicated backup, restore and recovery configurations and/or procedures like the ones that can be seen in many DBMS.

- **Simplified and Restricted yet Powerful**: Map-Reduce is a restricted programming model, it only provides straightforward map and reduce interfaces. However, most search-engine (and generic) data processing tasks can be effectively implemented in this model. These tasks can immediately enjoy high parallelism with only a few lines of administration and configuration code. This "simplified" philosophy can also be seen in many GFS designs. Developers can focus on formulating their tasks to the Map-Reduce interface, without worrying about such issues as implementing memory management, file allocation, parallel, multi-threaded, or network programming.

- **Highly Parallel yet Abstracted**: The most important contribution of Map-Reduce is perhaps its automatic parallelization and execution. Even though it might not be optimized for a specific task, the productivity gain from developing an application with Map-Reduce is far higher than doing it from scratch on the same requirements. Map-Reduce allows developers to focus mainly on the problem at hand rather than worrying about the administrative details.

- **High Throughput**: Deployed on low-cost hardware and modeled in simplified, generic frameworks, Map-Reduce systems are hardly optimized to perform like a massively parallel processing systems deployed with the same number of nodes. However, these disadvantages (or advantages) allow Map-Reduce jobs to run on thousands of nodes at relatively low cost. A scheduling system places each Map and Reduce task at a near-optimal node (considering the vicinity to data and load balancing), so that many Map-Reduce tasks can share the same cluster.

- **High Performance by the Large**: Even though Map-Reduce systems are generic, and not usually tuned to be high performance for specific tasks, they still can achieve high performance simply by being deployed on a large number of nodes. In [6], the authors mentioned a then world-record Terabyte [11] sorting benchmark by using Map-Reduce on thousands of machines. In short, sheer parallelism can generate high performance, and Map-Reduce programs can take advantage of it.

- **Shared-Disk Storage yet Shared-Nothing Computing**: In a Map-Reduce environment, every node has its own local hard drives. Mappers and reducers use these local disks to store intermediate files and these files are read remotely by reducers, i.e., Map-Reduce is a shared-nothing architecture. However, Map-Reduce jobs read input from and write output to GFS, which is shared by every node. GFS replicates disk chunks and uses pooled disks to support ultra large files. Map-Reduce's shared-nothing architecture makes it much more scalable than one that shares disk or memory. In the mean time, Map and Reduce tasks share an integrated GFS that makes thousands of disks behave like one.

- **Set-Oriented Keys and Values; File Abstracted**: With GFS's help, Map-Reduce can process thousands of file chunks in parallel. The volume can be far beyond the size limit set for an individual file by the underlying OS file system. Developers see data as keys and values, no longer raw bits and bytes, nor file descriptors.

- **Functional Programming Primitives**: The Map-Reduce interface is based on two functional-programming primitives [6]. Their signatures are re-produced

here:

$$\text{map: } (k_1, v_1) \rightarrow [(k_2, v_2)]$$
$$\text{reduce: } (k_2, [v_2]) \rightarrow [v_3]$$

The map function applies user-defined logic on every input key/value pair and transforms it into a list of intermediate key/value pairs. The reduce function applies user-defined logic to all intermediate values associated with the same intermediate key and produces a list of output values. This simplified interface enables developers to model their specific data processing into two-phase parallel tasks.

These signatures were informally defined for readability, they were not meant to be rigorous enough to pass a strongly-typed functional type checking mechanism. However, [14] pointed out that the reduce function output $[v_3]$ can be in different type from its input $[v_2]$.

- **Distributed Partitioning/Sorting Framework**: Map-Reduce system also includes phases that work on the intermediate data, and users usually do not need to deal with them directly. These phases include a partitioner function that partitions mapper outputs to reducer inputs, a sort-by-key function that sorts reducer inputs based on keys, and a group-by-key function that groups sorted key/value pairs with the same key into a single key/value pair of the same key and all the values. In its pure form, the system is essentially a 2-phase parallel sorter similar to the one in NOW [2].
- **Designed for Search Engine Operations yet Applicable to Generic Data Processing Tasks**: Map-Reduce is a generic framework, not limited to search engine operations. It can be applied to any data processing task that fits the simple map-reduce interface.

2.2 Homogenization

Despite all these advantages and design principles, Map-Reduce focuses mainly on processing homogeneous datasets. Through a process we called *homogenization*, Map-Reduce can be used to do *equi-joins* on multiple heterogeneous datasets. This homogenization process applies one map/reduce task on each dataset that it inserts a *data-source* tag into every value. It also extracts a key attribute common for all heterogeneous datasets. Transformed datasets now have two common attributes: key and data-source — they are *homogenized*. A final map/reduce task can then apply to all the homogenized datasets combined. Data entries from different datasets with the same key value will be grouped in the same reduce partition. User-defined logic can extract data-sources from values to identify their origins, then the entries from different sources can be merged.

This procedure takes lots of extra disk space, incurs excessive map-reduce communications, and is limited only to queries that can be rendered as equi-joins. In the next section, we will discuss a general approach of extending Map-Reduce to efficiently process multiple heterogeneous datasets.

3. MAP-REDUCE-MERGE

The *Map-Reduce-Merge* model enables processing multiple heterogeneous datasets. The signatures of the Map-Reduce-Merge primitives are listed below, where α, β, γ represent dataset lineages, k means keys, and v stands for value entities.

$$\text{map: } (k_1, v_1)_\alpha \rightarrow [(k_2, v_2)]_\alpha$$
$$\text{reduce: } (k_2, [v_2])_\alpha \rightarrow (k_2, [v_3])_\alpha$$
$$\text{merge: } ((k_2, [v_3])_\alpha, (k_3, [v_4])_\beta) \rightarrow [(k_4, v_5)]_\gamma$$

In this new model, the map function transforms an input key/value pair (k_1, v_1) into a list of intermediate key/value pairs $[(k_2, v_2)]$. The reduce function aggregates the list of values $[v_2]$ associated with k_2 and produces a list of values $[v_3]$, which is also associated with k_2. Note that inputs and outputs of both functions belong to the same lineage, say α. Another pair of map and reduce functions produce the intermediate output $(k_3, [v_4])$ from another lineage, say β. Based on keys k_2 and k_3, the merge function combines the two reduced outputs from different lineages into a list of key/value outputs $[(k_4, v_5)]$. This final output becomes a new lineage, say γ. If $\alpha = \beta$, then this merge function does a *self-merge*, similar to *self-join* in relational algebra.

Notice that the map and reduce signatures in the new model are almost the same as those in the original Map-Reduce. The only differences are the lineages of the datasets and the production of a key/value list from reduce instead of just values. These changes are introduced because the merge function needs input datasets organized (partitioned, then either sorted or hashed) by keys and these keys have to be passed into the function to be merged. In Google's Map-Reduce, the reduced output is final, so users pack whatever needed in $[v_3]$, while passing k_2 for next stage is not required.

To build a merge function that reads data from both lineages in an organized manner, the design of these signatures emphasizes having the key k_2 passed from map to reduce, then to merge functions. This is to make sure that data is partitioned, then sorted (or hashed) on the same keys before they can be merged properly. This condition, however, is too strong. Keys still can be transformed between phases and they do not even need to be of the same type (as implied by the same type descriptor k_2 used in every phase) as long as records pointed by transformed keys are still organized in the same way as the one by the mapped keys represented by k_2. For example, 4-digit integers can be transformed into 4-byte numerical strings padded with 0s. The order of integers and the one for transformed strings are the same, so they are compatible and replaceable between phases if compatible range partitioners are used in map functions. However, since users already can transform keys in the map function (from k_1 to k_2), there is hardly a need to transform them again in reduce and merge functions. Thus, to keep these signatures simple, we chose to have the same k_2 passed between phases.

As mentioned in [6], the map and reduce functions originate from functional programming. The merge function can be related to two-dimensional *list comprehension*, which is also popular in functional programming.

3.1 Example

In this section, we start with a simple example that will be continued to next sections. It shows how Map, Reduce, and Merge modules work together. There are two datasets in this example: *Employee* and *Department*. Employee's "key" attribute is *emp_id* and the others are packed into an *emp_info* "value." Department's "key" is *dept_id* and the

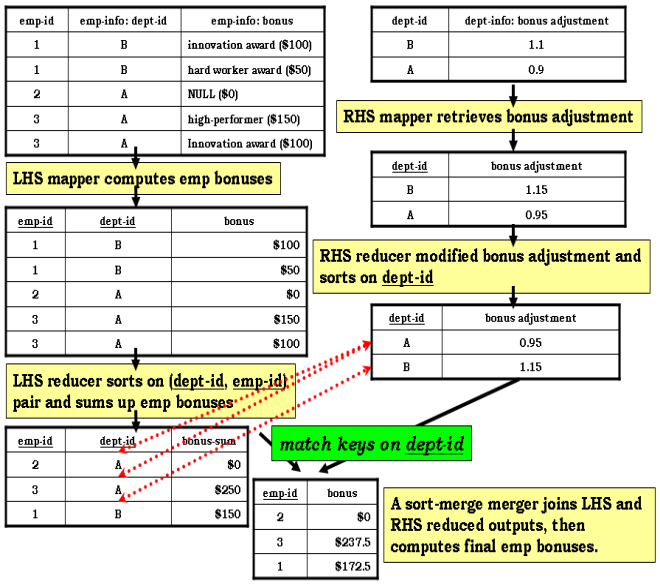

Figure 3: Example to join Employee and Department tables and compute employee bonuses (see § 3.1).

Algorithm 1 Map function for the Employee dataset.

```
1: map(const Key& key, /* emp_id */
2:       const Value& value /* emp_info */) {
3:   emp_id = key;
4:   dept_id = value.dept_id;
5:   /* compute bonus using emp_info */
6:   output_key = (dept_id, emp_id);
7:   output_value = (bonus);
8:   Emit(output_key, output_value);
9: }
```

others are packed into a *dept_info* "value." One example query is to join these two datasets and compute employee bonuses.

Before these two datasets are joined in a merger, they are first processed by a pair of mappers and reducers. A complete data flow is shown in Fig. 3. On the left hand side, a mapper reads Employee entries and computes a bonus for each entry. A reducer then sums up these bonuses for every employee and sorts them by dept_id, then emp_id. On the right hand side, a mapper reads Department entries and computes bonus adjustments. A reducer then sorts these department entries. At the end, a merger matches the output records from the two reducers on dept_id using the sort-merge algorithm, applies a department-based bonus adjustment on employee bonuses. Pseudocode for these mappers and reducers are shown in Alg. 1, 2, 3, and 4.

After these two pairs of Map-Reduce tasks are finished, a merger task takes their intermediate outputs, and joins them on *dept_id*. We will describe the details of major merge components in following sections.

3.2 Implementation

We have implemented a Map-Reduce-Merge framework, in which Map and Reduce components are inherited from Google Map-Reduce except minor signature changes. The new Merge module includes several new components: *merge*

Algorithm 2 Map function for the Department dataset.

```
1: map(const Key& key, /* dept_id */
2:       const Value& value /* dept_info */) {
3:   dept_id = key;
4:   bonus_adjustment = value.bonus_adjustment;
5:   Emit((dept_id), (bonus_adjustment));
6: }
```

Algorithm 3 Reduce function for the Employee dataset.

```
1: reduce(const Key& key, /* (dept_id, emp_id) */
2:         const ValueIterator& value
3:         /* an iterator for a bonuses collection */) {
4:   bonus_sum = /* sum up bonuses for each emp_id */
5:   Emit(key, (bonus_sum));
6: }
```

function, *processor* function, *partition selector*, and *configurable iterator*. We will use the employee-bonus example to explain the data and control flow of this framework and how these components collaborate.

The *merge* function (*merger*) is like *map* or *reduce*, in which developers can implement user-defined data processing logic. While a call to a map function (*mapper*) processes a key/value pair, and a call to a reduce function (*reducer*) processes a key-grouped value collection, a merger processes two pairs of key/values, that each comes from a distinguishable source.

At the Merge phase, users might want to apply different data-processing logic on data based on their sources. An example is the *build* and *probe* phases of a hash join, where *build* programming logic is applied on one table then *probe* the other. To accommodate this pattern, a *processor* is a user-defined function that processes data from one source only. Users can define two processors in Merge.

After map and reduce tasks are about done, a Map-Reduce-Merge coordinator launches mergers on a cluster of nodes (see Fig. 2). When a merger starts up, it is assigned with a merger number. Using this number, a user-definable module called *partition selector* can determine from which reducers this merger retrieves its input data. Mappers and reducers are also assigned with a number. For mappers, this number represents the input file split. For reducers, this number represents an input bucket, in which mappers partition and store their output data to. For Map-Reduce users, these numbers are simply system implementation detail, but in Map-Reduce-Merge, users utilize these numbers to associate input/output between mergers and reducers in partition selectors.

Like mappers and reducers, a merger can be considered as having logical iterators that read data from inputs. Each mapper and reducer have one logical iterator and it moves from the begin to the end of a data stream, which is an input file split for a mapper, or a merge-sorted stream for a reducer. A merger reads data from two sources, so it can be viewed as having two logical iterators. These iterators usually move forward as their mapper/reducer counterparts, but their relative movement against each others can be instrumented to implement a user-defined merge algorithm. Our Map-Reduce-Merge framework provides a user-configurable module (*iterator-manager*) that it is called for the information that controls the movement of these *configurable iterators*. Later, we will describe several iteration patterns from relational join algorithms. A Merge phase driver, as shown in Alg. 5, is needed to coordinate these

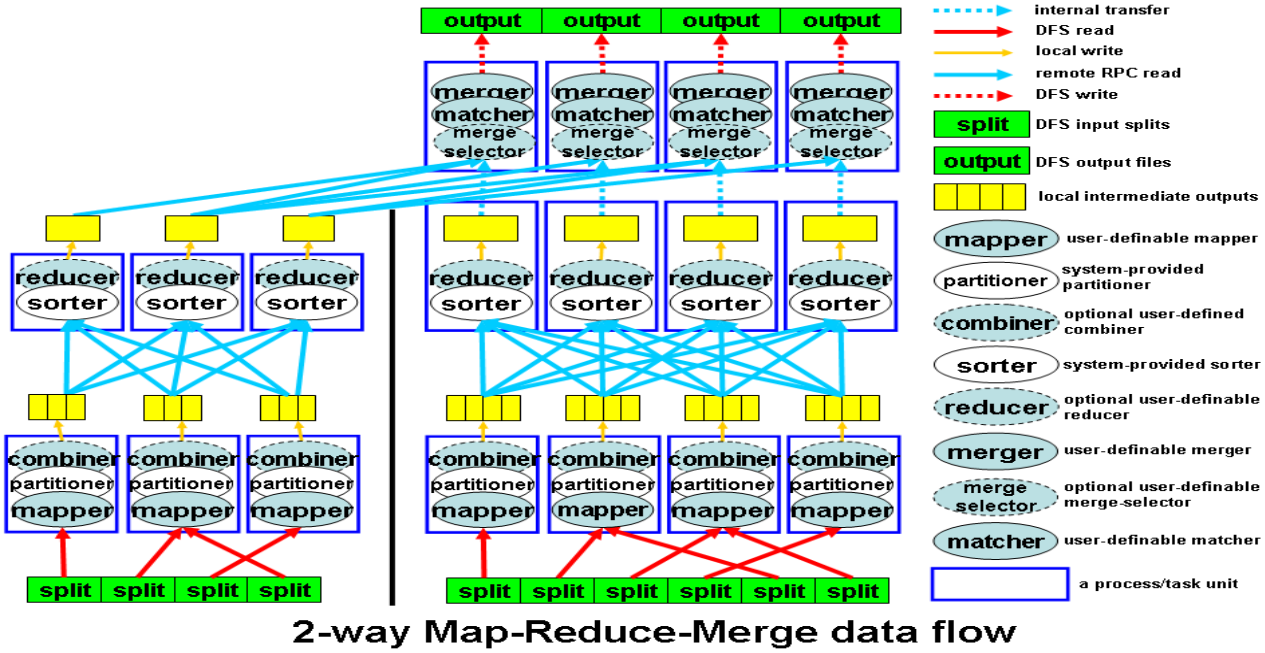

Figure 4: A 2-way Map-Reduce-Merge data flow. Data is processed by a mapper, partitioner, and combiner in the Map phase. Then, it is read remotely and processed by a sorter and reducer in the Reduce phase. In the Merge phase, selected reducer outputs are processed by a matcher and merger guided by a pair of configurable iterators.

Algorithm 4 Reduce function for the Department dataset.

```
1: reduce(const Key& key, /* (dept_id) */
2:          const ValueIterator& value
3:          /* an iterator on a bonus_adjustments collection */) {
4:   /* aggregate bonus_adjustments and
5:      compute a final bonus_adjustment */
6:   Emit(key, (bonus_adjustment));
7: }
```

Merge components and have them collaborate with each others.

3.2.1 Partition Selector

In a merger, a user-defined *partition selector* function determines which data partitions produced by up-stream reducers should be retrieved then merged. This function is given the current merger's number and two collections of reducer numbers, one for each data source. Users define logic in the selector to remove unrelated reducers from the collections. Only the data from the reducers left in the collections will be read and merged in the merger.

For the employee-bonus example, a simplified scenario stipulates that both sources have the same collection of reducer numbers and the same range partitioner function is applied to the *dept_id* key only in both mappers, so that both reducer outputs are completely sorted and partitioned into equal number of buckets. Notice that the employee mapper produces keys in pairs of (dept_id, emp_id), thus its reducer sorts data on this composite key, but partitioning is done on dept_id only. Based on these assumptions, a partition selector function can be defined to map reducers and mergers in an one-to-one relationship as in Alg. 6.

3.2.2 Processors

A processor is the place where users can define logic of processing data from an individual source. Processors can be defined if the hash join algorithm is implemented in Merge, where the first processor builds a hash table on the first source, and the second probes it while iterating through the second data source. In this case, the *merger* function is empty. Since we will apply the sort-merge algorithm on the bonus-computation join example, these processors stay empty.

3.2.3 Merger

In the merge function, users can implement data processing logic on data merged from two sources where this data satisfies a merge condition. Alg. 7 shows the last step of computing employee bonuses by adjusting an employee's raw bonus with a department-based adjustment.

3.2.4 Configurable Iterators

As indicated, by manipulating relative iteration of a merger's two logical iterators, users can implement different merge algorithms.

For algorithms like nested-loop joins, iterators are configured to move as looping variables in a nested loop. For algorithms like sort-merge joins, iterators take turns when iterating over two sorted collections of records. For hash-join-like algorithms, these two iterators scan over their data in separate passes. The first scans its data and *builds* a hash table, then the second scans its data and *probes* the already built hash table.

Allowing users to control iterator movement increases the risk of running into a never-ending loop. This risk always ex-

Algorithm 5 Merge phase driver.

```
1: PartitionSelector partitionSelector; // user-defined logic
2: LeftProcessor leftProcessor; // user-defined logic
3: RightProcessor rightProcessor; // user-defined logic
4: Merger merger; // user-defined logic
5: IteratorManager iteratorManager; // user-defined logic
6: int mergerNumber; // assigned by system
7: vector<int> leftReducerNumbers; // assigned by system
8: vector<int> rightReducerNumbers; // assigned by system
9: // select and filter left and right reducer outputs for this merger
10: partitionSelector.select(mergerNumber,
11:                          leftReducerNumbers,
12:                          rightReducerNumbers);
13: ConfigurableIterator left = /*initiated to point to entries
14:   in reduce outputs by leftReducerNumbers*/
15: ConfigurableIterator right =/*initiated to point to entries
16:   in reduce outputs by rightReducerNumbers*/
17: while(true) {
18:   pair<bool,bool> hasMoreTuples =
19:     make_pair(hasNext(left), hasNext(right));
20:   if (!hasMoreTuples.first && !hasMoreTuples.second) {break;}
21:   if (hasMoreTuples.first) {
22:     leftProcessor.process(left→key, left→value); }
23:   if (hasMoreTuples.second) {
24:     rightProcessor.process(right→key, right→value); }
25:   if (hasMoreTuples.first && hasMoreTuples.second) {
26:     merger.merge(left→key, left→value,
27:                  right→key, right→value); }
28:   pair<bool,bool> iteratorNextMove =
29:    iteratorManager.move(left→key, right→key, hasMoreTuples);
30:   if (!iteratorNextMove.first && !iteratorNextMove.second) {
31:     break; }
32:   if (iteratorNextMove.first) { left++; }
33:   if (iteratorNextMove.second) { right++; }
34: }
```

Algorithm 6 One-to-one partition selector.

```
1: bool select(int mergerNumber,
2:              vector<int>& leftReducerNumbers,
3:              vector<int>& rigthReducerNumbers) {
4:   if (find(leftReducerNumbers.begin(),
5:            leftReducerNumbers.end(),
6:            mergerNumber) == leftReducerNumbers.end()) {
7:     return false; }
8:   if (find(rightReducerNumbers.begin(),
9:            rightReducerNumbers.end(),
10:            mergerNumber) == rightReducerNumbers.end()) {
11:     return false; }
12:   leftReducerNumbers.clear();
13:   leftReducerNumbers. push_back(mergerNumber);
14:   rightReducerNumbers.clear();
15:   rightReducerNumbers. push_back(mergerNumber);
16:   return true;
17: }
```

ists in user-defined logic and is a great concern, especially in strictly-regulated DBMS systems. For programming models like the Map-Reduce and Map-Reduce-Merge, this issue is lesser because they are, after all, programming models and *data processing* frameworks.

Still, it is a nuisance if a task never ends, so a framework should provide a mechanism to reduce the chance of it happening. In our implementation, we use a boolean pair returned by a user-defined function to indicate whether to move an iterator to point to the next entity. This function is called after each merge operation; true indicates forward and false indicates stay. If both booleans are false, then the whole merge process is terminated.

Suppose reducers produce sorted outputs in an ascendant order, Alg. 8 shows the programming logic of coordinating iterator movement for sort-merge-alike algorithms. If both sources still have inputs, then move the iterator that points to a smaller key. If both keys are equivalent, then move the

Algorithm 7 Merge function for the employee-department join.

```
1: merge(const LeftKey& leftKey,
2:       /* (dept_id, emp_id) */
3:       const LeftValue& leftValue, /* sum of bonuses */
4:       const RightKey& rightKey, /* dept_id */
5:       const RightValue& rightValue /* bonus-adjustment */){
6:   if (leftKey.dept_id == rightKey) {
7:     bonus = leftValue * rightValue;
8:     Emit(leftKey.emp_id, bonus); }
9: }
```

Algorithm 8 Iteration logic for sort-merge joins.

```
1: move(const LeftKey& leftKey,
2:       const RightKey& rightKey,
3:       const pair<bool, bool>& hasMoreTuples) {
4:   if (hasMoreTuples.first && hasMoreTuples.second) {
5:     if (leftKey < rightKey) {
6:       return make_pair(true, false); }
7:     return make_pair(false, true); }
8:   return hasMoreTuples;
9: }
```

right iterator by default. If one source is exhausted, this information is stored in the input bool pair "hasMoreTuples," move the iterator for the source that still has data.

Alg. 9 is an implementation of nested-loop iteration pattern. In a nested loop, keys are ignored in determining how to move iterators. If the left and right sources are exhausted, then the merge process is terminated. It is a logic error if the right source still have data when the left is exhausted. If the left source is not exhausted, then move the right iterator only. When the right source is exhausted, move the left iterator and reset the right iterator to the beginning of its data source.

To implement algorithms that follow the hash join's two-scan iteration pattern, a merger first scans one data source from the beginning to the end, then repeats the scan on the other one, e.g., see Alg. 10.

Notice that, for the *employee-bonus* example, implementing configurable iterators is tied to the choosing of partitioners. Using the sort-merge-based configurable iterators requires a range partitioner in both mappers.

4. APPLICATIONS TO RELATIONAL DATA PROCESSING

One fundamental idea of Map-Reduce-Merge is to bring relational operations into parallel data processing at the search-engine scale. On the other hand, map, reduce, and merge can be used as standardized components in implementing parallel OLAP DBMS. Novel data-processing applications such as search engines and Map-Reduce's unorthodox principles and assumptions make it worthwhile to revisit parallel databases [7, 16].

4.1 Map-Reduce-Merge Implementations of Relational Operators

In our implementation, the Map-Reduce-Merge model assumes that a dataset is mapped into a relation R with an attribute set (schema) A. In map, reduce, and merge functions, users choose attributes from A to form two subsets: K and V. K represents the schema of the "key" part of a Map-Reduce-Merge record and V the "value" part. For each tuple t of R, this implies that t is concatenated by two field sets: k

Algorithm 9 Iteration logic for nested-loop joins.

```
1: move(const LeftKey& leftKey,
2:         const RightKey& rightKey,
3:         const pair<bool, bool>& hasMoreTuples) {
4:   if (!hasMoreTuples.first && !hasMoreTuples.second) {
5:     return make_pair(false, false); }
6:   if (!hasMoreTuples.first && hasMoreTuples.second)
7:     /* throw a logical-error exception */
8:   if (hasMoreTuples.first && !hasMoreTuples.second) {
9:     /* reset the right iterator to the beginning */
10:    return make_pair(true, false); }
11:  return make_pair(false, true);
12: }
```

Algorithm 10 Iteration logic for hash joins.

```
1: move(const LeftKey& leftKey,
2:         const RightKey& rightKey,
3:         const pair<bool, bool>& hasMoreTuples) {
4:   if (!hasMoreTuples.first && !hasMoreTuples.second){
5:     return make_pair(false, false); }
6:   if (hasMoreTuples.first) {
7:     return make_pair(true, false); }
8:   return make_pair(false, true);
9: }
```

and v, where K is the schema of k and V is the schema of v. It so happens that Map-Reduce-Merge calls k as "key" and v as "value". This naming is arbitrary in the sense that their attribute sets are decided solely by the user. This "key" is used in Map-Reduce-Merge functions for partitioning, sorting, grouping, matching, and merging tuples. By no means it has the same uniqueness meaning in relational languages. Below we describe how Map-Reduce-Merge can be used to implement primitive and some derived relational operators, so that Map-Reduce-Merge is *relationally complete*, while being load-balanced, scalable, and parallel.

- **Projection**: For each tuple $t = (k, v)$ of the input relation, users can define a mapper to transform it into a projected output tuple $t' = (k', v')$, where k' and v' are typed by schema K' and V', respectively. K' and V' are subsets of A. Namely, using mappers only can implement relational algebra's projection operator.

- **Aggregation**: At the Reduce phase, Map-Reduce (as well as Map-Reduce-Merge) performs the sort-by-key and group-by-key functions to ensure that the input to a reducer is a set of tuples $t = (k, [v])$ in which $[v]$ is the collection of all the values associated with the key k. A reducer can call aggregate functions on this grouped value list. Namely, reducers can easily implement the "group by" clause and "aggregate" operators in SQL.

- **Generalized Selection**: Mappers, reducers, and mergers can all act as filters and implement the selection operator. If a selection condition is on attributes of one data source, then it can be implemented in mappers. If a selection condition is on aggregates or a group of values from one data source, then it can be implemented in reducers. If a selection condition involves attributes or aggregates from more than one sources, then it can be implemented in mergers.

  Straightforward filtering conditions that involve only one relation in a SQL query's "where" and "having" clauses can be implemented using mappers and reducers, respectively. Mergers can implement complicated filtering conditions involving more than one relations, however, this filtering can only be accomplished after join (or Cartesian product) operations are properly configured and executed.

- **Joins**: § 4.2 describes in detail how joins can be implemented using mergers with the help from mappers and reducers.

- **Set Union**: Assume the union operation (as well as other set operations described below) is performed over two relations. In Map-Reduce-Merge, each relation will be processed by Map-Reduce, and the sorted and grouped outputs of the reducers will be given to a merger. In each reducer, duplicated tuples from the same source can be skipped easily. The mappers for the two sources should share the same range partitioner, so that a merger can receive records within the same key range from the two reducers. The merger can then iterate on each input simultaneously and produce only one tuple if two input tuples from different sources are duplicates. Non-duplicated tuples are produced by this merger as well.

- **Set Intersection**: First, partitioned and sorted Map-Reduce outputs are sent to mergers as described in the last item. A merger can then iterate on each input simultaneously and produce tuples that are shared by the two reducer outputs.

- **Set Difference**: First, partitioned and sorted Map-Reduce outputs are sent to mergers as described in the last item. A merger can then iterate on each input simultaneously and produce tuples that are the difference of the two reducer outputs.

- **Cartesian Product**: In a Map-Reduce-Merge task, the two reducer sets will produce two sets of reduced partitions. A merger is configured to receive one partition from the first reducer (F) and the complete set of partitions from the second one (S). This merger can then form a nested loop to merge records in the sole F partition with the ones in every S partition.

- **Rename**: It is trivial to emulate *Rename* in Map-Reduce-Merge, since map, reduce, and merge functions can select, rearrange, compare, and process attributes based on their indexes in the "key" and "value" subsets.

Map-Reduce-Merge is certainly more expressive than the relational algebra, since map, reduce, and merge can all contain user-defined programming logic.

4.2 Map-Reduce-Merge Implementations of Relational Join Algorithms

Join is perhaps the most important relational operator. In this section, we will describe how Map-Reduce-Merge can implement three most common join algorithms.

4.2.1 Sort-Merge Join

From [6], Map-Reduce is shown to be an effective parallel sorter. The key of sorting is to partition input records based on their actual values instead of, by Map-Reduce default, hashed values. That is, instead of using a *hash partitioner*,

users can configure the framework to use a *range partitioner* in mappers. Using this Map-Reduce-based sorter, the Map-Reduce-Merge framework can be implemented as a parallel, sort-merge join operator. The programming logic for each phase is:

- **Map**: Use a range partitioner in mappers, so that records are partitioned into ordered buckets, each is over a mutually exclusive key range and is designated to one reducer.
- **Reduce**: For each Map-Reduce lineage, a reducer reads the designated buckets from all the mappers. Data in these buckets are then merged into a sorted set. This sorting procedure can be done completely at the reducer side, if necessary, through an external sort. Or, mappers can sort data in each buckets before sending them to reducers. Reducers can then just do the merge part of the *merge sort* using a priority queue.
- **Merge**: A merger reads from two sets of reducer outputs that cover the same key range. Since these reducer outputs are sorted already, this merger simply does the merge part of the *sort-merge join*.

4.2.2 Hash Join

One important issue in distributed computing and parallel databases is to keep workload and storage balanced among nodes. One strategy is to disseminate records to nodes based on their hash values. This strategy is very popular in search engines as well as in parallel databases. It is the the default partitioning mechanism in Map-Reduce [6] and the only partitioning strategy in Teradata [16], a parallel RDBMS. Another approach is to run a preprocessing Map-Reduce task to scan the whole dataset and build a data density [6]. This density can be used by partitioners in later Map-Reduce tasks to ensure balanced workload among nodes. Here we show how to implement *hash join* [8] using the Map-Reduce-Merge framework:

- **Map**: Use a common hash partitioner in both mappers, so that records are partitioned into hashed buckets, each is designated to one reducer.
- **Reduce**: For each Map-Reduce lineage, a reducer reads from every mapper for one designated partition. Using the same hash function from the partitioner, records from these partitions can be grouped and aggregated using a hash table. This hash-based grouping is an alternative to the default sorting-based approach. It does not need a sorter, but requires maintaining a hashtable either in memory or disk.
- **Merge**: A merger reads from two sets of reducer outputs that share the same hashing buckets. One is used as a *build* set and the other *probe*. After the partitioning and grouping are done by mappers and reducers, the build set can be quite small, so these sets can be hash-joined in memory. Notice that, the number of reduce/merge sets must be set to an optimally large number in order to support an in-memory hash join, otherwise, an external hash join is required.

4.2.3 Block Nested-Loop Join

The Map-Reduce-Merge implementation of the *block nested-loop* join algorithm is very similar to the one for the hash join. Instead of doing an in-memory hash, a nested loop is implemented. The partitioning and grouping done by mappers and reducers concentrate the join sets, so this parallel nested-loop join can enjoy a high selectivity in each merger.

- **Map**: Same as the one for the hash join.
- **Reduce**: Same as the one for the hash join.
- **Merge**: Same as the one for the hash join, but a nested-loop join is implemented, instead of a hash join.

5. OPTIMIZATIONS

Map-Reduce provides several optimization mechanisms, including *locality* and *backup tasks* [6]. In this section, we describe some strategies that can reduce resources (e.g, the number of network connections and disk bandwidth) used in the Merge phase.

5.1 Optimal Reduce-Merge Connections

For a natural join over two datasets, A and B, suppose for A, there are M_A number of mappers and R_A number of reducers; and for B, M_B and R_B. Each A mapper produces R_A partitions, and each B mapper R_B. Conversely, each A reducer reads from every A mappers for the partitions designated for it. Same applies to B reducers from B mappers. To simplify the scenario, let $R_A = R_B = R$, then in total there would be at least $R \times (M_A + M_B)$ remote reads (not counting redundant connections incurred by backup jobs) among nodes where mappers and reducers reside. This is a lot of remote reads among nodes, but it is the price to pay to group and aggregate same-key records as these records were originally scattered around in the whole cluster.

For mergers, because data is already partitioned and even sorted after Map and Reduce phases, they do not need to connect to every reducer in order to get their data. The selector function in mergers can choose pertinent reduced partitions for merging. For example, in a simplified scenario, if there is also R number of mergers, then these mergers can have an one-to-one association with A reducers and also with B reducers. A user-defined selector can be like the one shown in Alg. 6. This selector receives two collections of reducer numbers for A and B reducers. It then picks the reducers who share the same number with the merger and removes other reducers' numbers from the collections. The merger then uses the selected reducer numbers to set up connections with and requests data from these reducers. In the one-to-one case, the number of connections between reducers and mergers is $2R$.

If one input dataset is much larger than the other, then it would be inefficient to partition both datasets into the same number of reducers. One can choose different numbers for R_A and R_B, but the selection logic is more complicated.

Selector logic can also be quite complicated in the case of θ-join. However, selector is a optimization mechanism that can help avoid excessive remote reads. A naive selection can always put only the merger number in one reducer number set and leave the other set intact (see the selection logic in 11) and still get the correct result. This is basically a Cartesian product between two reduced sets. The number of remote reads now becomes $R^2 + R$.

Before feeding data from selected reducer partitions to a user-defined merger function, these tuples can be compared and see if they should be merged or not. In short, this

Algorithm 11 Cartesian-product partition selector.

```
1: select(int mergerNumber,
2:          vector<int>& leftReducerNumbers,
3:          vector<int>& rightReducerNumbers) {
4:   if (find(leftReducerNumbers.begin(),
5:            leftReducerNumbers.end(),
6:            mergerNumber) == leftReducerNumbers.end()) {
7:     return false; }
8:   leftReducerNumbers.clear();
9:   leftReducerNumbers.push_back(mergerNumber);
10:   return true;
11: }
```

comparison can be done in a user-defined *matcher* that is simply a fine-grained selector.

5.2 Combining Phases

To accomplish a data processing task, it usually takes several Map-Reduce-Merge (or Map-Reduce) processes weaved in a workflow, in which the output of a process become the input of a subsequent one. The entire workflow may constitute many disk-read-write passes. For example, Fig. 6 shows a TPC-H Q2 join tree implemented with 13 Map-Reduce-Merge passes. These passes can be optimized and combined:

- **ReduceMap, MergeMap**: Reducer and merger outputs are usually fed into a down-stream mapper for a subsequent join operation. These outputs can simply be sent directly to a co-located mapper in the same process without storing them in secondary storage first.

- **ReduceMerge**: A merger usually takes two sets of reducer partitions. This merger can be combined with one of the reducers and gets its output directly while remotely reads data from the other set of reducers.

- **ReduceMergeMap**: An straightforward combination of ReduceMerge and MergeMap becomes ReduceMergeMap.

Another way of reducing disk accesses is to replace disk read-writes with network read-writes. This method requires connecting up- and down-stream Map-Reduce-Merge processes while they are running. This approach is arguably more complicated than saving intermediate data in local disks, thus it may not comply with the "simplified" philosophy of the Map-Reduce framework. When a process fails, this network-based I/O strategy can cause difficulties for upstream processes to recollect the data already computed and resend them to a new down-stream process.

6. ENHANCEMENTS

Besides optimizations, some Map-Reduce-Merge enhancements can make coding easier.

6.1 Map-Reduce-Merge Library

There are many variations and patterns for the merge module, such as the ones that implement relational operators or join algorithms. The selectors and configurable iterators for these common merge implementations can be put into a library and users can use them in their Map-Reduce-Merge tasks without reinventing the wheel.

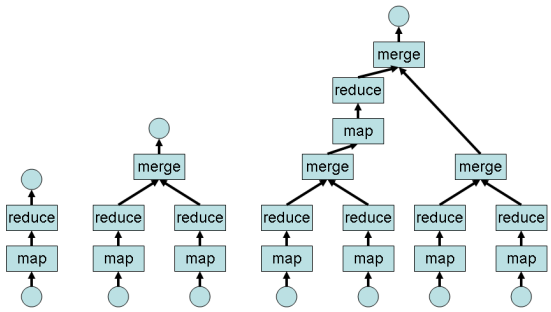

Figure 5: Map-Reduce-Merge workflows. The left is a typical 2-pass Map-Reduce workflow. The middle one is a typical 3-pass Map-Reduce-Merge workflow. The right is a multi-pass hierarchical workflow built with several Map, Reduce, and Merge modules.

6.2 Map-Reduce-Merge Workflow

Map-Reduce programs follow a strict two-phase workflow, i.e., mapping then reducing. Users have options to change default configurations, but some basic operations such as partitioning and sorting, are built-ins and cannot be skipped. This, sometimes, is a nuisance if users would like to do mapping only or to do reducing on already mapped datasets. These scenarios are quite common in real-world tasks as well as in debugging only one of the Map-Reduce modules. This constraint makes Map-Reduce simplified and it enables unified usage and implementation, but advanced users may want to see it relaxed, i.e., they may want to create a customized *workflow*. Since there are only two phases in Map-Reduce, it is not a serious issue. However, adding a new phase (Merge, as proposed in this paper; future Map-Reduce improvements might include other phases) creates many workflow combinations that can fit the specific needs of a data processing task. This is especially true for processing relational queries where an execution plan constitutes a workflow of several Map, Reduce, and Merge modules (see an example in § 7.2).

A Map-Reduce-Merge enhancement is to provide a configuration API for building a customized workflow. In Fig. 5, the left is a traditional Map-Reduce workflow. The middle one is a basic Map-Reduce-Merge workflow. The left one is a more complicated example.

When building a Map-Reduce-Merge workflow, an important issue is to avoid using a distributed file system (DFS) for storing intermediate data. In Google's Map-Reduce implementation, mapper outputs are stored in local hard drives, instead of in GFS. GFS is only used to store permanent datasets like the inputs and outputs of a Map-Reduce task. If a Map-Reduce implementation stores intermediate datasets in DFS, then it basically becomes a shared-disk architecture. This might make it not as scalable as a shared-nothing implementation [9].

Although we have only discussed hierarchical workflows so far, in fact, outputs can be used as inputs in a Map-Reduce-Merge workflow, making it recursive. These recursive workflows can be used to implement SQL recursive queries, for example.

7. CASE STUDIES

In this section, we will present two case studies applying

the Map-Reduce-Merge programming model to real-world data processing tasks. The first is a search-engine task, while the second is a rather complicated TPC-H query.

7.1 Join Webgraphs

In simple terms, a webgraph database for a search engine stores a table in which each row has one URL (regarded as the key) along with attributes such as its inlinks and outlinks. The number of attributes can be large, and for many operations, only a few of them are needed. As such, a webgraph database may store each column of the table in a separate file, distributed over many machines. This choice of storage creates a need for joins. As an example, consider the following three columns: URLs, inlinks, and outlinks. Suppose for each URL, we need to compute the intersection of its inlinks and outlinks. One way to compute the intersection is (1) to create a table of all three columns (URL, inlinks, outlinks), and (2) compute the intersection over each row and output (URL, inlinks intersect outlinks). Records of these columns are related to each other through row-ids. These row-ids are used in place of URLs as keys to these column files. The creation of the joined table can be implemented with two 2-way joins: (1) join URLs and inlinks using the row-ids as the common attribute, (2) join the first join's result dataset and outlinks using row-ids as the common attribute. Then, a simple Map-Reduce can scan the result dataset and find the inlink-outlink intersection.

These collections of inlinks and outlinks can be considered as nested tables. As the number of inlinks and outlinks for popular websites (e.g., www.yahoo.com) can be very large that reading them directly into a map, reduce, or merge process can overflow buffer. A safer approach is to flatten these nested tables and replicate row-id to every inlink (or outlink) record that belongs to the same URL. One sort-merge-based intersect can produce records (row-id, inoutlink) that are shared by both (row-id, inlink) and (row-id, outlink) datasets. An ensuing Map-Reduce-Merge natural join with the (row-id, URL) dataset can than replace row-ids with URLs and create the result dataset: (URL, inoutlink).

7.2 Map-Reduce-Merge Workflow for TPC-H Query 2

To demonstrate how the Map-Reduce-Merge programming model can be used to process complicated data relationships, we use the TPC-H [17] schema and its No. 2 query (see Fig. 6) as an example.

This query is rather complicated. It involves *five* tables, one *nested query*, one *aggregate* and *group by* clause, and at the end, the result dataset is *ordered by* several attributes. The conditions for the 5-way join are all equal conditions, while the nested query is only meant to select the tuples with the minimum *supply cost*. Though this nested query is also a 5-way join (4 tables in the *from* clause and one outer table), because it is essentially the same as the outer join, its logic can be processed during executing the outer one. Based on these observations, we use an execution plan that first does four 2-way joins for the overall 5-way join. Then, this plan does group-by and selection operations for the nested query and a sorting operation for the order-by clause. The join tree of this execution plan is shown in Fig. 7. This plan might not be the most efficient one. We just use it as an example for implementing a SQL query under the Map-Reduce-Merge framework. Notice that the *region* and

```
-- TPC-H/TPC-R Minimum Cost Supplier Query (Q2)
select
  s_acctbal,
  s_name,
  n_name,
  p_partkey,
  p_mfgr,
  s_address,
  s_phone,
  s_comment
from
  part,
  supplier,
  partsupp,
  nation,
  region
where
  p_partkey = ps_partkey
  and s_suppkey = ps_suppkey
  and p_size = :1
  and p_type like '%:2'
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = ':3'
  and ps_supplycost = (
    select
      min(ps_supplycost)
    from
      partsupp,
      supplier,
      nation,
      region
    where
      p_partkey = ps_partkey
      and s_suppkey = ps_suppkey
      and s_nationkey = n_nationkey
      and n_regionkey = r_regionkey
      and r_name = ':3'
  )
order by
  s_acctbal desc,
  n_name,
  s_name,
  p_partkey;
```

Figure 6: TPC-H Query 2.

nation tables are very small. They do not need a parallel join implementation with a complete suite of Map, Reduce, and Merge tasks. In fact, they can be read into memory as look-up tables by mappers for other tables, such as *supplier*.

In the join tree, *part* and *partsupp* are joined into a temporary table called *p_ps*. In parallel, *region* and *nation* are joined into *n_r*. Table *n_r* are then joined with *supplier* into *s_n_r*. Later, *p_ps* and *s_n_r* are joined into *p_ps_s_n_r*. Once these four 2-way joins are done for the overall 5-way join, *p_ps_s_n_r* is processed by two Map-Reduce tasks. The first one does the nested query's *group by* clause and its reducer selects the tuples with the minimum *supply-cost*. The final Map-Reduce task is simply a sorter for the *order by* clause.

In Fig. 7, we mechanically replace each join with a suite of Map, Reduce, and Merge tasks. Thirteen disk-read-write passes are needed to process the execution plan. In total, there are 10 mappers, 10 reducers, and 4 mergers.

These numbers can be reduced by a simple optimization that integrates merger and reducer modules with a follow-up mapper. This optimization reduces the number of passes to 9 with 5 mappers, 9 reducers, 4 merge-mappers, and 1 reduce-mapper.

If reducers and their follow-up mergers are further combined as suggested in § 5.2, then the number of passes is reduced to 6 with 5 mappers, 1 reducer, 4 reduce-merge-mappers, and 1 reduce-mapper (see Fig. 8).

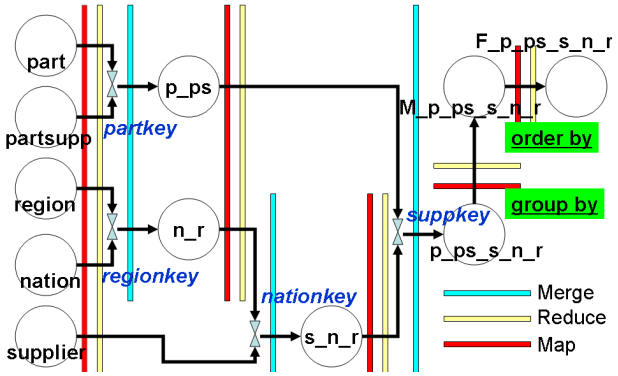

Figure 7: A join tree for TPC-H Query 2. It is implemented with 13 passes of Map-Reduce-Merge modules (10 mappers, 10 reducers, and 4 mergers).

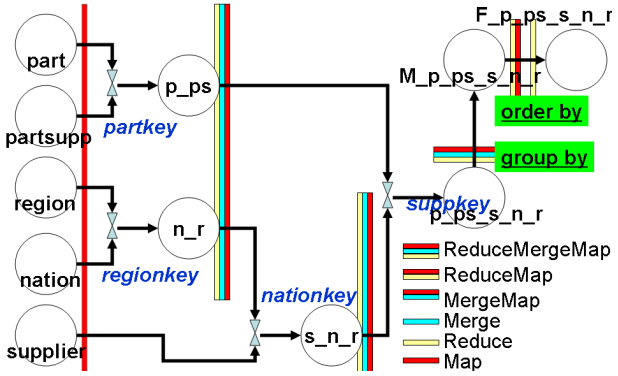

Figure 8: The join tree of Fig. 7 is re-implemented with 6 passes of combined Map-Reduce-Merge modules (5 mappers, 4 reduce-merge-mappers, 1 reduce-mapper, and 1 reducer).

8. CONCLUSIONS

Map-Reduce and GFS represent a rethinking of data processing that uses only the most critical database principles for their target applications, instead of relying on overly generalized DBMS. This "simplified" philosophy drives down hardware and software cost for data-intensive systems such as search engines, while Map-Reduce still provides great features like high-throughput, high-performance, fault-tolerant, and easy administration, etc. The most important feature of Map-Reduce is that it abstracts parallel programming into two simple primitives, map and reduce, so that developers can easily convert many real-world data processing jobs into parallel programs.

However, Map-Reduce does not directly support joins of heterogeneous datasets, so we propose adding a Merge phase. This new *Map-Reduce-Merge* programming model retains Map-Reduce's many great features, while adding relational algebra to the list of database principles it upholds. It also contains several configurable components that enable many data-processing patterns.

Map-Reduce-Merge can also be used as an infrastructure that supports parallel database functionality. We have demonstrated that the Map-Reduce-Merge framework can be used to implement many relational operators, particularly joins. A natural next step is to develop an SQL-like interface and an optimizer to simplify the process of developing a Map-Reduce-Merge workflow. This work can readily reuse well-studied RDBMS techniques.

Acknowledgments. We would like to thank reviewers and Yahoo! Search colleagues for suggestions and discussions.

9. REFERENCES

[1] Apache. Hadoop. *http://lucene.apache.org/hadoop/*, 2006.

[2] A. C. Arpaci-Dusseau et al. High-Performance Sorting on Networks of Workstations. In *SIGMOD 1997*, pages 243–254, 1997.

[3] E. A. Brewer. Combining Systems and Databases: A Search Engine Retrospective. In J. M. Hellerstein and M. Stonebraker, editors, *Readings in Database Systems, Fourth Edition*, Cambridge, MA, 2005. MIT Press.

[4] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[5] L. Chu et al. Optimizing Data Aggregation for Cluster-Based Internet Services. In *PPOPP*, pages 119–130. ACM, 2003.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[7] D. J. DeWitt et al. GAMMA - A High Performance Dataflow Database Machine. In *VLDB 1986*, pages 228–237, 1986.

[8] D. J. DeWitt and Gerber.R. Multiprocessor Hash-Based Join Algorithms. In *VLDB 1985*, 1985.

[9] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, 1992.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[11] J. Gray. Sort Benchmark. *http://research.microsoft.com/barc/SortBenchmark/*, 2006.

[12] J. Gray et al. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[13] M. Isard et al. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[14] R. Lämmel. Google's MapReduce Programming Model – Revisited. Draft; Online since 2 January, 2006; 26 pages, 22 Jan. 2006.

[15] R. Pike et al. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming Journal*, 13(4):227–298, 2005.

[16] Teradata. Teradata. *http://www.teradata.com/t/go.aspx*, 2006.

[17] TPC. TPC-H. *http://www.tpc.org/tpch/default.asp*, 2006.

[18] Wikipedia. Redundant Array of Inexpensive Nodes. *http://en.wikipedia.org/wiki/Redundant_Array_of_Inexpensive_Nodes*, 2006.