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

The HaLoop Approach to Large-Scale Iterative Data Analysis

Yingyi Bu · Bill Howe · Magdalena Balazinska · Michael D. Ernst

Received: date / Accepted: date

Abstract The growing demand for large-scale data mining and data analysis applications has led both industry and academia to design new types of highly scalable data-intensive computing platforms. MapReduce has enjoyed particular success. However, MapReduce lacks built-in support for iterative programs, which arise naturally in many applications including data mining, web ranking, graph analysis, and model fitting. This paper[1] presents HaLoop, a modified version of the Hadoop MapReduce framework, that is designed to serve these applications. HaLoop allows iterative applications to be assembled from existing Hadoop programs without modification, and significantly improves their efficiency by providing inter-iteration caching mechanisms and a loop-aware scheduler to exploit these caches. HaLoop retains the fault-tolerance properties of MapReduce through automatic cache recovery and task re-execution. We evaluated HaLoop on a variety of real applications and real datasets. Compared with Hadoop, on average, HaLoop improved runtimes by a factor of 1.85 and shuffled only 4% as much data between mappers and reducers in the applications that we tested.

Yingyi Bu
University of California - Irvine, Irvine, CA, USA 92697
E-mail: yingyib@ics.uci.edu

Bill Howe
University of Washington, Seattle, WA, USA 98195
E-mail: billhowe@cs.washington.edu

Magdalena Balazinska
University of Washington, Seattle, WA, USA 98195
E-mail: magda@cs.washington.edu

Michael D. Ernst
University of Washington, Seattle, WA, USA 98195
E-mail: mernst@cs.washington.edu

[1] This is an extended version of the VLDB 2010 paper "HaLoop: Efficient Iterative Data Processing on Large Clusters" [8].

1 Introduction

The need for highly scalable parallel data processing platforms is rising due to an explosion in the number of massive-scale data-intensive applications both in industry (e.g., web-data analysis, click-stream analysis, network-monitoring log analysis) and in the sciences (e.g., analysis of data produced by high-resolution, massive-scale simulations and new high-throughput sensors and devices).

MapReduce [12] is a popular framework for programming commodity computer clusters to perform large-scale data processing in a single pass. A MapReduce cluster can scale to thousands of nodes in a fault-tolerant manner. Although parallel database systems [13] may also serve these data analysis applications, they can be expensive, difficult to administer, and typically lack fault-tolerance for long-running queries [32]. Hadoop [17], an open-source MapReduce implementation, has been adopted by Yahoo!, Facebook, and other companies for large-scale data analysis. With the MapReduce framework, programmers can parallelize their applications simply by implementing a map function and a reduce function to transform and aggregate their data, respectively. The MapReduce model has been shown to be suitable for a variety of algorithms, including web-scale document analysis [12], relational query evaluation [20], and large-scale image processing [37].

However, many data analysis techniques require *iterative* computations, including PageRank [31], HITS (Hypertext-Induced Topic Search) [23], recursive relational queries [5], clustering [22], neural-network analysis [18], social network analysis [35], and internet traffic analysis [27]. These techniques have a common trait: data are processed iteratively until the computation satisfies a convergence or stopping condition. The MapReduce framework does not directly support these iterative data analysis applications. Instead, programmers must implement iterative pro-

url	rank
u_0	0.2
u_1	0.2
u_2	0.2
u_3	0.2
u_4	0.2

(a) Initial Rank Table R_0

url_source	url_dest
u_0	u_1
u_0	u_2
u_2	u_0
u_4	u_3
u_3	u_1
u_2	u_4
u_4	u_2
u_0	u_3
u_1	u_3

(b) Linkage Table L

$$MR_1 \begin{cases} T_{i_1} = R_i \bowtie_{url=url\_source} L & \text{(Computes relation: (url\_source, url\_dest, rank))} \\ T_{i_2} = \gamma_{url, rank, \frac{rank}{\text{COUNT}(url\_dest)} \rightarrow new\_rank}(T_{i_1}) & \text{(Computes: (url\_source,new\_intermediate\_rank))} \\ T_{i_3} = T_{i_2} \bowtie_{url=url\_source} L & \text{(Computes: (url\_source, url\_dest, new\_intermediate\_rank)} \end{cases}$$

$$MR_2 \begin{cases} R_{i+1} = \gamma_{url\_dest \rightarrow url, \frac{1-d}{N} + d * \text{SUM}(new\_intermediate\_rank) \rightarrow rank}(T_{i_3}) \\ \text{(for each url\_dest, aggregate populated ranks from different url\_source)} \\ \text{(d is the damping factor and usually is set to 0.85 while N the total count of distinct urls)} \end{cases}$$

(c) Loop Body

url	rank
u_0	0.084
u_1	0.28
u_2	0.13
u_3	0.42
u_4	0.08

(d) Rank Table R_3

Fig. 1 PageRank example

grams by issuing multiple MapReduce jobs and orchestrating their execution using a driver program [25].

In addition to being cumbersome for the developer, there are two key performance problems related to the use of a driver program to manage iteration. The first problem is that even though much of the data may be unchanged from iteration to iteration, the data must be re-loaded and re-processed at each iteration, wasting I/O, network bandwidth, and CPU resources. The second problem is that the termination condition may involve detecting when a *fixpoint* has been reached — i.e., when the application's output does not change between two consecutive iterations. Computing this condition may itself require an *extra* MapReduce job on each iteration, again incurring overhead in terms of scheduling extra tasks, reading extra data from disk, and moving data across the network. To illustrate these problems, consider the following two examples.

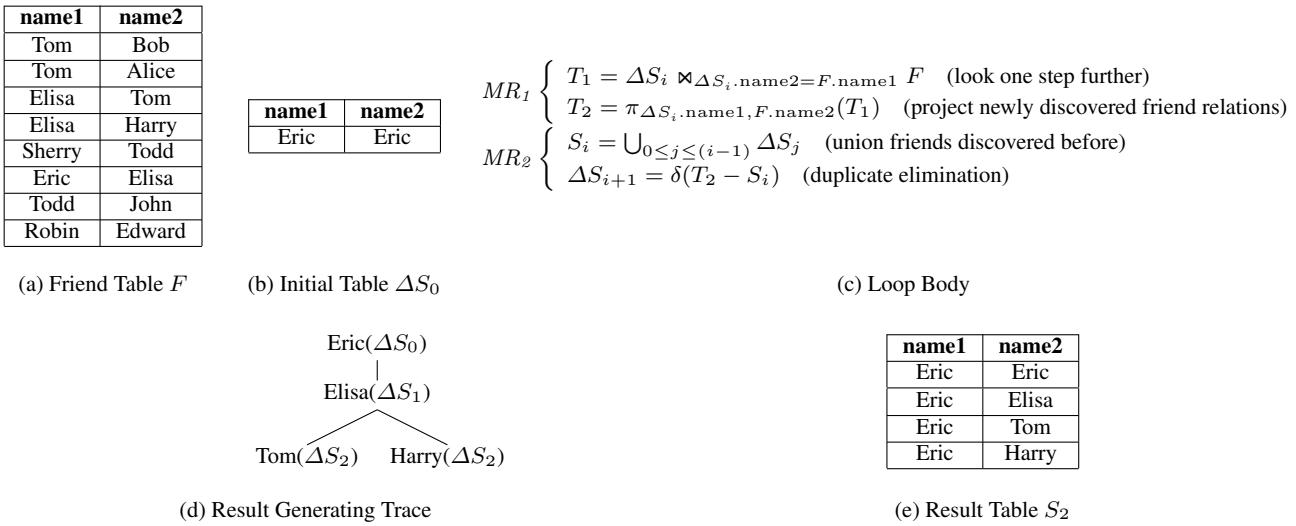
Example 1 (PageRank) PageRank [31] is a graph analysis algorithm that assigns weights (ranks) to each vertex by iteratively aggregating the weights of its inbound neighbors. In the relational algebra, each iteration of the PageRank algorithm can be expressed as two joins, two aggregations, and one update (Figure 1(c)). These steps must be repeated by a driver program until a termination condition is satisfied (e.g., either a specified number of iterations have been performed or the rank of each page converges).

Figure 1 shows a concrete example. R_0 (Figure 1(a)) is the initial rank table, and L (Figure 1(b)) is the linkage table. Two MapReduce jobs (MR_1 and MR_2 in Figure 1(c)) implement the loop body of PageRank. The first MapReduce job joins the rank and linkage tables using a typical *reduce-side join* algorithm [36]: the map phase hashes both relations by their join attributes, and the reduce phase computes the join for each key. As a simple optimization, this job also computes the rank contribution for each outbound edge, $new\_intermediate\_rank$. The second MapReduce job computes the aggregate rank of each unique destination URL: the map function is the identity function, and the reducers sum the rank contributions of each incoming edge. In each iteration, R_i is updated to R_{i+1}. For example, one could obtain R_3 (Figure 1(d)) by iteratively computing R_1, R_2, and finally R_3. The computation terminates when the rank values converge. Checking this termination condition requires yet another distributed computation on each iteration, implemented as yet another MapReduce job (not shown in Figure 1(c)).

This example illustrates several inefficiencies. In the PageRank algorithm, the linkage table L is invariant across iterations. Because the MapReduce framework is unaware of this property, however, L is processed and shuffled at each iteration. Worse, the invariant linkage data may frequently be larger than the resulting rank table. Finally, determining whether the ranks have converged requires an extra MapReduce job after each iteration.

As a second example, consider a simple recursive query.

Example 2 (Descendant Query) Given the social network relation in Figure 2(a), we wish to find everyone within two "friend-hops" of Eric. The relation ΔS_0={Eric, Eric} is the reflexive friend relationship that initiates the computation, as shown in Figure 2(b). This query can be implemented by a driver program that executes two MapReduce jobs (MR_1 and MR_2 in Figure 2(c)) for two iterations. The first MapReduce job finds a new generation of friends ΔS_i by joining



name1	name2
Tom	Bob
Tom	Alice
Elisa	Tom
Elisa	Harry
Sherry	Todd
Eric	Elisa
Todd	John
Robin	Edward

(a) Friend Table F

name1	name2
Eric	Eric

(b) Initial Table ΔS_0

$$MR_1 \begin{cases} T_1 = \Delta S_i \bowtie_{\Delta S_i.\text{name2}=F.\text{name1}} F & \text{(look one step further)} \\ T_2 = \pi_{\Delta S_i.\text{name1},F.\text{name2}}(T_1) & \text{(project newly discovered friend relations)} \end{cases}$$

$$MR_2 \begin{cases} S_i = \bigcup_{0 \le j \le (i-1)} \Delta S_j & \text{(union friends discovered before)} \\ \Delta S_{i+1} = \delta(T_2 - S_i) & \text{(duplicate elimination)} \end{cases}$$

(c) Loop Body

(d) Result Generating Trace

name1	name2
Eric	Eric
Eric	Elisa
Eric	Tom
Eric	Harry

(e) Result Table S_2

Fig. 2 Descendant query example

the friend table F with the friends discovered in the previous iteration. The second MapReduce job removes duplicates — those tuples in ΔS_i that also appear in any ΔS_j for any $j < i$. Figure 2(d) shows how results evolve from ΔS_0 to ΔS_2. The final result is the concatenation of the disjoint results from each iteration (Figure 2(e)). Other implementations are possible.

This example essentially implements the semi-naïve evaluation strategy for recursive logic programs [5]: we remove duplicate answers on each iteration to reduce the number of redundant facts derived. Implemented as a MapReduce program, however, this approach still involves wasted work. As in the PageRank example, a significant fraction of the data (the friend table F) remains constant throughout the execution of the query, yet still gets processed and shuffled at each iteration.

Many other data analysis applications have characteristics similar to the above two examples: a significant fraction of the processed data remains invariant across iterations, and the termination condition may involve a distributed computation. Other examples include statistical learning and clustering algorithms (e.g., k-means), web/graph ranking algorithms (e.g., HITS [23]), and recursive graph or network queries.

This paper presents a new system called *HaLoop* that is designed to efficiently handle these applications. HaLoop adopts four design goals: first, loop-invariant data should not be re-processed on each iteration; second, tests for fixpoints (e.g., PageRank convergence) should not require an extra MapReduce job on each iteration; third, any optimizations should not sacrifice Hadoop's fault-tolerance properties; fourth, porting existing Hadoop applications to HaLoop should not require changes to existing code, and should require as little new code as possible.

This paper makes the following contributions:

- *New Programming Model and Architecture for Iterative Programs*: HaLoop offers a programming interface to express iterative data analysis applications, and allows programmers to reuse existing mappers and reducers from conventional Hadoop applications (Section 2.2).
- *Loop-Aware Scheduling*: HaLoop's scheduler ensures that tasks are assigned to the same nodes across multiple iterations, allowing the use of local caches to improve performance (Section 3).
- *Caching for Loop-Invariant Data*: During the first iteration of an application, HaLoop creates indexed local caches for data that are invariant across iterations. Caching the invariant data reduces the I/O cost for loading and shuffling them in subsequent iterations (Section 4.2 and Section 4.4).
- *Efficient Fixpoint Evaluation*: HaLoop caches a reducer's local output to allow efficient comparisons of results between iterations. This technique can help avoid a superfluous MapReduce job to check convergence conditions (Section 4.3).
- *Failure Recovery*: HaLoop preserves the fault-tolerance properties of MapReduce by automatically reconstructing the caches from intermediate results and using them to re-execute failed tasks (Section 5).
- *System Implementation*: We implemented HaLoop by modifying the Hadoop MapReduce framework. As a result, mappers and reducers developed for Hadoop can be used unchanged as part of a HaLoop application (Section 6).
- *Experimental Evaluation*: We evaluated HaLoop on iterative programs that process both synthetic and real world datasets. HaLoop outperforms Hadoop on all tested metrics. On average, HaLoop improves query runtimes by a factor or 1.85, and shuffles only 4% of the data be-

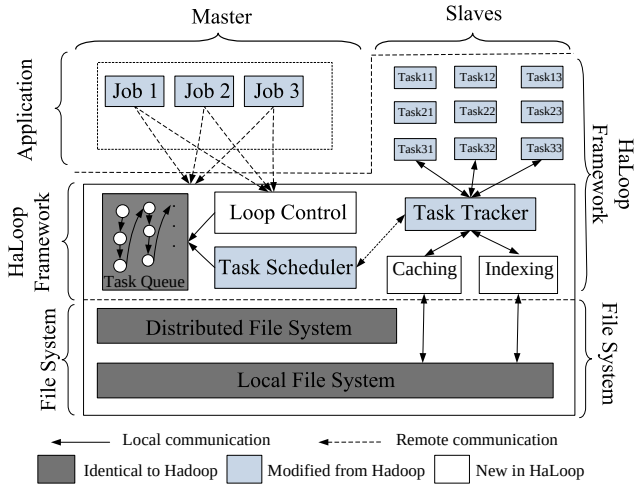


Fig. 3 The HaLoop framework, a variant of the Hadoop MapReduce framework. There are three jobs running in the system: job 1, job 2, and job 3. Each job has three tasks running concurrently on slave nodes.

tween mappers and reducers in the applications that we test (Section 7).

2 HaLoop Overview

This section first introduces HaLoop's architecture and compares our design choices with other alternatives, and then illustrates HaLoop's application programming model and interface.

2.1 Architecture

Figure 3 illustrates the architecture of HaLoop, a modified version of the open-source MapReduce implementation Hadoop [17].

HaLoop inherits the basic distributed computing model and architecture of Hadoop. The latter relies on a distributed file system (HDFS [19]) that stores each job's input and output data. A Hadoop cluster is divided into two parts: one master node and many slave nodes. A client submits jobs consisting of mapper and reducer implementations to the master node. For each submitted job, the master node schedules a number of parallel tasks to run on slave nodes. Every slave node has a task tracker daemon process to communicate with the master node and manage each task's execution. Each task is either a map task or a reduce task. A map task performs transformations on an input data partition by executing a user-defined map function on each ⟨key, value⟩ pair. A reduce task gathers all mapper output assigned to it by a potentially user-defined hash function, groups the output by keys, and invokes a user-defined reduce function on each key-group.

HaLoop uses the same basic model. In order to accommodate the requirements of iterative data analysis applications, however, HaLoop includes several extensions. First, HaLoop extends the application programming interface to express iterative MapReduce programs (Section 2.2). Second, HaLoop's master node contains a new loop control module that repeatedly starts new MapReduce steps that compose the loop body, continuing until a user-specified stopping condition is satisfied (Section 2.2). Third, HaLoop caches and indexes application data on slave nodes' local disks (Section 4). Fourth, HaLoop uses a new loop-aware task scheduler to exploit these caches and improve data locality (Section 3). Fifth, if failures occur, the task scheduler and task trackers coordinate recovery and allow iterative jobs to continue executing (Section 5). As shown in Figure 3, HaLoop relies on the same file system and has the same task queue structure as Hadoop, but the task scheduler and task tracker modules are modified and the loop control, caching, and indexing modules are new. The HaLoop task tracker not only manages task execution, but also manages caches and indices on the slave node, and redirects each task's cache and index accesses to the local file system.

2.2 Programming Model

In HaLoop, the programmer specifies an iterative computation by providing: 1) the computation performed by each map and reduce in the loop body, 2) the names of the files produced and accessed by each step in the loop body, 3) the termination condition, and 4) the application-specific caching behavior. This section describes each of these interfaces.

Map and reduce computations. In HaLoop, the `Map` function and `Reduce` function have exactly the same interfaces and semantics as they do in Hadoop:

- `Map` transforms an input ⟨key, value⟩ tuple into intermediate ⟨inter_key, inter_value⟩ tuples.
- `Reduce` processes intermediate tuples sharing the same inter_key, to produce ⟨out_key, out_value⟩ tuples.

To express an iterative program, the user must specify the loop body and the termination condition. Each step in the loop body accepts one or more input datasets and produces a single output dataset. The termination condition is either specified as a fixed number of iterations or as a *fixpoint*, which is satisfied when a specified output dataset does not change from one iteration to the next. Each loop iteration comprises one or more MapReduce steps. The programmer constructs a sequence of MapReduce steps as the loop body by calling the following function once for each step:

- `AddMapReduceStep(int step, Map map_impl, Reduce reduce_impl)` creates a MapReduce step consisting of a map function `map_impl` and a reduce

function `reduce_impl` as well as an integer that indicates its execution step in the loop body.

Inputs/output for each loop step. To specify the inputs and output of each job in the loop body, the programmer implements two functions:

- `GetInputPaths(int iteration, int step)` returns a list of input file paths for the step in the iteration. Frequently, a step refers to two kinds of datasets: 1) loop-invariant "constant" datasets, and 2) datasets produced by steps executed in current or prior iterations.
- `GetOutputPath(int iteration, int step)` tells the framework the output path for the step in the iteration. Each (iteration, step) pair is expected to have a unique output path; no dataset should be overwritten in order to preserve fault-tolerance properties.

Termination condition. To express an iterative program, the user must specify the loop body and the termination condition. Each step in the loop body accepts any input datasets and produces any output datasets. The termination condition is either a given number of iterations or a *fixpoint*; that is, when a specified output does not change from one iteration to the next.

A fixpoint is typically defined by exact equality between iterations, but HaLoop also supports the concept of an *approximate fixpoint*, where the computation terminates when the difference between two consecutive iterations is less than a user-specified threshold. HaLoop also supports terminating the computation after a maximum number of iterations have occurred. Either or both of these termination conditions may be used in the same loop. For example, PageRank may halt when the ranks do not change more than a user-specified convergence threshold ϵ from one iteration to the next, but never run more than, say, 10 iterations [31].

To specify the termination condition, the programmer may use one or both of the following functions:

- `SetFixedPointThreshold(DistanceMeasure dist, double threshold)` specifies a distance measure and sets a bound on the distance between the output of one iteration and the next. If the threshold is exceeded, then the approximate fixpoint has not yet been reached, and the computation continues.
  The user-defined `dist` function calculates the distance between two reducer output value sets sharing the same reducer output key, without knowing what the key is. The total distance between the last-step reducer outputs of the current iteration i and the previous iteration $i-1$ is

$$\sum_{out\_key} \texttt{dist}(R_i|_{out\_key}, R_{i-1}|_{out\_key})$$

  where $R|_{out\_key} = \{out\_value \mid \langle out\_key, out\_value\rangle \in R\}$. It is straightforward to support other aggregations besides sum.
  A common use case for the distance function is to test a convergence criterion. For example, the k-means algorithm terminates when the k centroids do not move by more than some threshold in a given iteration. As another example, a distance function could return 0 if the two value sets are non-empty and 1 if either value set is empty; the total distance is then the number of keys in the symmetric difference of the two key-sets.
- `SetMaxNumOfIterations` provides further control of the loop termination condition. HaLoop terminates a job if the maximum number of iterations has been executed, regardless of the distance between the current and previous iteration's outputs. On its own, `SetMaxNumOfIterations` can also be used to implement a simple `for`-loop.

Caching Behavior. HaLoop supports three types of caches: the mapper input cache, the reducer input cache, and the reducer input cache. The reducer input cache and mapper input cache can be created automatically in the first iteration by analyzing the dependency graph of the loop body (see Section 4.1). But HaLoop also exposes an interface for fine-grained control over the caches. In particular, we find that applications may need to specify the data to cache on a tuple-by-tuple basis. In Example 1, reducer slave nodes only cache tuples from the L table, rather than those from R_i. However, the framework may not know which tuple is from L nor which is from R_i, since the table tag is embedded in the reducer input key or value, which is application specific. HaLoop introduces the `CacheFilter` interface for programmers to implement, with one method:

- `isCache(K key, V value, int id)` returns true if the ⟨key, value⟩ tuple needs to be cached, and returns false otherwise. The parameter *id* provides auxiliary information for decision making. For a mapper input or reducer output tuple, *id* is the number of tuples consumed or produced so far. For a reducer input tuple, *id* is the number of tuples processed so far with the same *key*. The *id* is useful in certain cases, such as multi-way joins, like the PageRank example implementation in Figure 20.

There are several methods to enable and disable different cache types. Additionaly, HaLoop exposes a method accepting a `CacheFilter` implementation and a cache type, which HaLoop applies to the data before creating the cache of the specified type.

In summary, the HaLoop programming model supports programs of the form

```
repeat until termination condition:
    Output_0 = MR_0(Inputs_0)
    Output_1 = MR_1(Inputs_1)
    ...
    Output_{N-1} = MR_{N-1}(Inputs_{N-1})
```

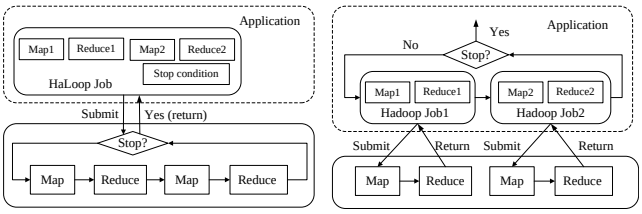


Fig. 4 Boundary between an iterative application and the framework (HaLoop vs. Hadoop). HaLoop knows and controls the loop, while Hadoop only knows jobs with one MapReduce pair.

where each $\texttt{MR}_i$ is a MapReduce job. Figure 4 shows the difference between HaLoop and Hadoop, from the application's perspective: in HaLoop, a user program specifies loop settings and the framework controls the loop execution, but in Hadoop, it is the application's responsibility to control the loops.

3 Loop-aware Task Scheduling

This section introduces the HaLoop task scheduler. The scheduler provides potentially better schedules for iterative programs than Hadoop's scheduler by co-locating tasks with the caches they need. Sections 3.1 and 3.2 explain the desired schedules and scheduling algorithm, respectively.

3.1 Inter-Iteration Locality

The goal of HaLoop's scheduler is to schedule tasks that access the same data on the same physical machine, even if these tasks occur in different iterations or different steps in the same iteration. This capability allows the use of caches to store data across iterations, reducing redundant work.

For example, Figure 5 is a sample schedule for two iterations of the join step (MR_1 in Figure 1(c)) of the PageRank application from Example 1. The relation L (which may span several nodes) represents the original graph, and does not change from iteration to iteration. The relation R (which may also span several nodes) represents the ranks computed on each iteration. To improve performance, L is cached and re-used on each iteration.

There are three slave nodes involved in the job. The scheduling of iteration 0 is no different than in Hadoop: the map tasks are arbitrarily distributed across the three slave nodes, as are the reduce tasks. In the join step of iteration 0, the input tables are L and R_0. Three map tasks are executed, each of which loads a part of one or the other input data file (a.k.a., a file split). As in ordinary Hadoop, the mapper output key (the join attribute in this example) is hashed to determine the reduce task to which it should be assigned. Then, three reduce tasks are executed, each of which loads a partition of the collective mapper output. In Figure 5, reducer R_{00} processes mapper output keys whose hash value is 0, reducer R_{01} processes keys with hash value 1, and reducer R_{02} processes keys with hash value 2.

The scheduling of the join step of iteration 1 can take advantage of inter-iteration locality: the task (either a mapper or reducer) that processes a specific data partition D is scheduled on the physical node where D was processed in iteration 0. Note that the two file inputs to the join step in iteration 1 are L and R_1.

The schedule in Figure 5 provides the ability to reuse loop-invariant data from past iterations. Because L is loop-invariant, mappers M_{10} and M_{11} would compute identical results to M_{00} and M_{01}. There is no need to re-compute these mapper outputs, nor to communicate them to the reducers. In iteration 0, if reducer input partitions 0, 1, and 2 are stored on nodes n_2, n_0, and n_1 respectively, then in iteration 1, L need not be loaded, processed, or shuffled again. In that case, in iteration 1, only one mapper M_{12} for R_1-split0 needs to be launched, and thus the three reducers will only copy intermediate data from M_{12}. With this strategy, the reducer input is no different, but it now comes from two sources: the output of the mappers (as usual) and the local disk.

Definition: *inter-iteration locality*. Let d be a file split (mapper input partition) or a reducer input partition[2], and let T_d^i be a task consuming d in iteration i. Then we say that a schedule exhibits *inter-iteration locality* if for all $i > 0$, T_d^{i-1} and T_d^i are assigned to the same physical node if T_d^{i-1} exists.

The goal of task scheduling in HaLoop is to achieve inter-iteration locality. To achieve this goal, the only restriction is that HaLoop requires that the number of reduce tasks should be invariant across iterations, such that the hash function assigning mapper outputs to reducer nodes remains unchanged.

3.2 Scheduling Algorithm

The HaLoop scheduler is a modified version of the standard Hadoop *TaskQueue* scheduler. As in the TaskQueue scheduler, each slave node has a fixed number of slots to hold running tasks (either map or reduce). If the slots are all being used, the slave node is considered fully loaded. Each slave node sends periodic heartbeats to the master, and the master decides whether to schedule a task to the hosting slave upon each heartbeat. If the slave node already has a full load, the master re-assigns its tasks to a nearby slave node. Other improved strategies such as delay scheduling [38] are out of our scope and are orthogonal to the HaLoop system.

[2] Mapper input partitions in both Hadoop and HaLoop are represented by an input file URL plus an offset and length; reducer input partitions are represented by an integer hash value. Two partitions are assumed to be equal if their representations are equal.

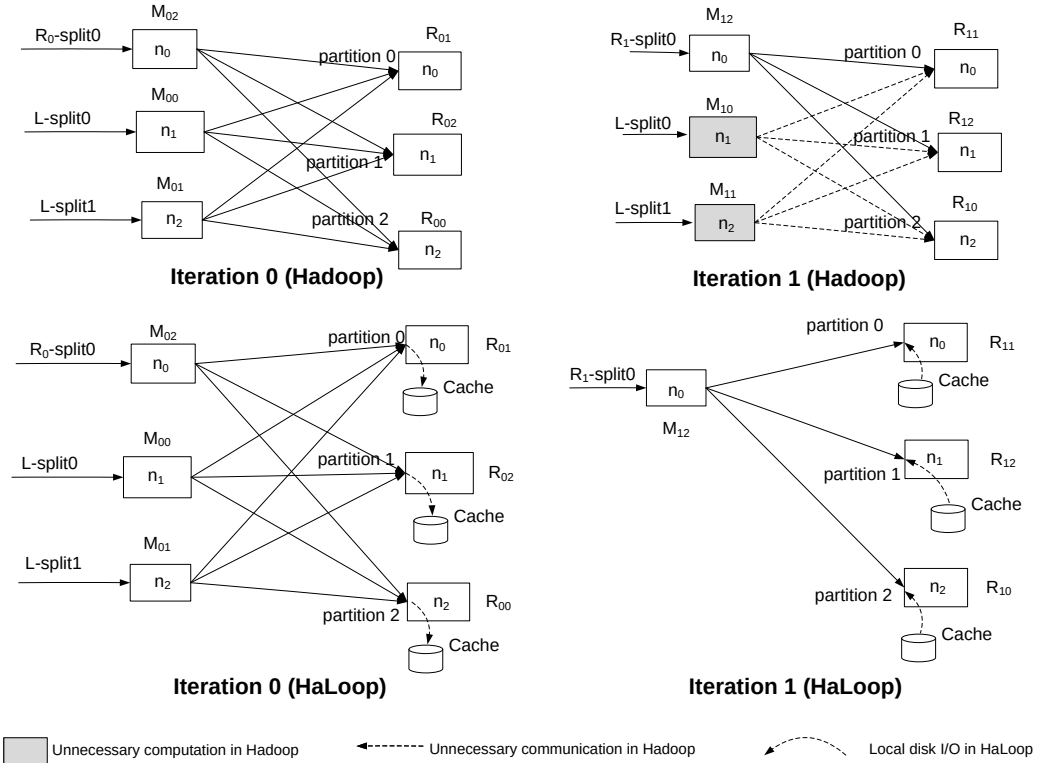


Fig. 5 A schedule exhibiting inter-iteration locality. HaLoop schedules tasks processing the same inputs on consecutive iterations to the same physical nodes. M_{ij} is the mapper task consuming input partition j in iteration i, and R_{ij} is the reducer task consuming mapper output partition j in iteration i. No partition is assumed to fit in main memory of a single node.

The HaLoop's scheduler keeps track of the data partitions processed by each map and reduce task on each physical machine, and it uses that information to schedule subsequent tasks taking inter-iteration locality into account.

Figure 6 shows the pseudocode for the scheduling algorithm. In a job's iteration 0, the schedule is exactly the same as that produced by Hadoop (line 2). Note that, on line 2, hadoopSchedule does *not* return the entire hadoop schedule. Rather, it returns one "heartbeat worth" of the entire schedule. After scheduling, the master remembers the association between data and node (lines 3 and 12). In later iterations, the scheduler tries to retain previous data-node associations (lines 12 and 13). If the associations is no longer appropriate due to increased load, the master node will associate the data with another node (lines 5–8). Each time the scheduling algorithm is called, at most one partition is assigned to the node (line 3 and 12). Task failures or slave node failures can also cause the scheduler to break inter-iteration locality, as we discuss in Section 5.

4 Caching and Indexing

Thanks to the inter-iteration locality offered by the task scheduler, access to a particular loop-invariant data partition is usually only needed by one physical node. To reduce I/O cost, HaLoop caches those data partitions on the physical node's local disk for subsequent re-use. To further accelerate processing, it indexes the cached data. Section 5 discusses how caches are reloaded in case of failures or changes in the task-to-node mapping. This section presents the details of how HaLoop detects loop-invariant datasets and builds and uses three types of caches: reducer input cache, reducer output cache, and mapper input cache. Each one fits a number of application scenarios. Application programmers can choose to enable or disable a cache type. HaLoop also has an interface that allows for programmers to do fine-grained cache control (see Section 2.2).

4.1 Detecting Loop-Invariant Datasets

Files that are suitable for inter-iteration caching are those that are not produced by any jobs and are read repeatedly by the same step across multiple iterations. Our experimental results show that caching is worthwhile even in the case where the file is read just twice.

Before an iterative job is executed, the master node identifies caching opportunities by examining the dependency graph of the overall job. It constructs a directed bipartite dependency graph for the first n iterations (typically just two) as follows. The graph contains a node for each step $\langle i, j \rangle$ at iteration i, step j, and also add a node for each unique file input and file output returned by `GetInputPaths(i,j)` and

Task Scheduling
Input: Node node, int iteration
// The current iteration's schedule; initially each list is empty
Global variable: Map⟨Node, List⟨Partition⟩⟩ current
// Initially, the latest iteration's schedule; eventually each list is empty
Global variable: Map⟨Node, List⟨Partition⟩⟩ previous

```
1:  if iteration == 0 then
2:      Partition part = hadoopSchedule(node);
3:      current.get(node).add(part);
4:  else if node.hasFullLoad() then
5:      Partition part = previous.get(node).get(0);
6:      Node substitution = findNearestIdleNode(node);
7:      previous.get(node).remove(part);
8:      previous.get(substitution).add(part));
9:  else if previous.get(node).size()>0 then
10:     Partition part = previous.get(node).get(0);
11:     schedule(part, node);
12:     current.get(node).add(part);
13:     previous.get(node).remove(part);
14: end if
```

Fig. 6 Task scheduling algorithm, whose goal is to assign the slave node an unassigned task that uses data cached on that node. If there are running jobs, this function is called whenever the master node receives a heartbeat from a slave, which happens multiple times per iteration. Before each iteration, `previous` is set to `current`, and then `current` is set to a new map containing empty lists.

`GetOutputPath(i, j)`. The graph contains an edge from step $\langle i, j\rangle$ to file x if $x \in$ `GetInputPaths`(i, j), and an edge from file x to step $\langle i, j\rangle$, if $x =$ `GetOutputPath`(i, j). A file is *cachable* if its node x in the dependency graph 1) has no outgoing edges, and 2) all incoming edges involve the same step. From the detected loop-invariant datasets, the HaLoop framework determines which step in the first iteration writes the cache and which step(s) in upcoming iterations read the cache.

4.2 Reducer Input Cache

The reducer input cache avoids the need to re-process the same data with the same mapper in multiple iterations. For those datasets determined to be cacheable by the loop-invariant detection procedure, HaLoop will cache the reducer inputs across all reducers and create a local index for the cached data. Reducer inputs are cached before each reduce function invocation, so that tuples in the reducer input cache are sorted and grouped by reducer input key.

Consider the social network example (Example 2) to see how the reducer input cache works. Three physical nodes n_0, n_1, and n_2 are involved in the job, and the number of reducers is set to 2. In the join step of the first iteration, there are three mappers: one processes F-split0, one processes F-split1, and one processes ΔS_0-split0. The three splits are shown in Figure 7. The two reducer input partitions are shown in Figure 8. The reducer on n_0 corresponds to hash value 0, while the reducer on n_1 corresponds to hash value 1. Then, since table F (with table ID "#1") is detected to be invariant, every reducer will cache the tuples with table ID "#1" in its local file system.

name1	name2
Tom	Bob
Tom	Alice
Elisa	Tom
Elisa	Harry

(a) F-split0

name1	name2
Sherry	Todd
Eric	Elisa
Todd	John
Robin	Edward

(b) F-split1

name1	name2
Eric	Eric

(c) ΔS_0-split0

Fig. 7 Mapper input splits in Example 2

name1	name2	table ID
Elisa	Tom	#1
Elisa	Harry	#1
Robin	Edward	#1
Tom	Bob	#1
Tom	Alice	#1

(a) partition 0

name1	name2	table ID
Eric	Elisa	#1
Eric	Eric	#2
Sherry	Todd	#1
Todd	John	#1

(b) partition 1

Fig. 8 Reducer input partitions in Example 2 ("#1" is the tag for tuples from F and "#2" is the tag for tuples from ΔS_0.)

In later iterations, when a reduce task passes a shuffled key with associated values to the user-defined `Reduce` function, it also searches for the key in the local reducer input cache with the current step number to find associated values, appends them to the sequence of shuffled values, and passes the key and merged value iterator to the `Reduce` function.

In the physical layout of the cache, keys and values are separated into two files, and each key has an associated pointer to its corresponding values. Sometimes the selectivity in the cached loop-invariant data is low. In this case, after reducer input data are cached to local disk, HaLoop creates an index (ISAM of level 2) over the keys and stores it in the local file system too. Since the reducer input cache is sorted and then accessed by reducer input key in the same sorted order, the disk seek operations are only conducted in a forward manner, and in the worst case, in each iteration, the input cache is sequentially scanned from local disk only once. Currently, HaLoop does not support full outer join, right outer join, or other computations that involve iterating over all keys in the cache. Keys that exist only in the cache (the right relation) but not in the shuffled input (the left relation) do not trigger reduce invocations. An extension to remove this limitation without sacrificing the performance benefits in the equijoin case is to allow the programmer to specify a flag that indicates whether all keys in the cache should be considered or not. This extension has been implemented in the current code base, but was not implemented during the experiments in this paper.

The reducer input cache is suitable for PageRank, HITS, various recursive relational queries, and any other algorithm with repeated joins against large invariant data. The reducer input cache requires that the partition function f for every

mapper output tuple t satisfies three conditions: (1) f must be deterministic, (2) f must remain the same for the same step across iterations, and (3) f must not take any inputs other than the tuple t. In HaLoop, the number of reduce tasks is unchanged for the same step across iterations, therefore the default hash partitioning satisfies these conditions. In addition, the `Reduce` function implementation must be aware that the order of values might be changed due to caching: For each reducer input key, we append cache values *after* the shuffled values, when they may have originally appeared at the beginning.

4.3 Reducer Output Cache

The reducer output cache is used in applications where fixpoint evaluation should be conducted after each iteration. For example, in PageRank, a user may set a convergence condition specifying that the total rank difference from one iteration to the next is below a given threshold. With the reducer output cache, the fixpoint can be evaluated in a distributed manner without requiring a separate MapReduce step. The distance between current output and latest output is accumulated after each `Reduce` function call. After all `Reduce` function invocations are done, each reducer evaluates the fixpoint condition within the reduce process and reports local evaluation results to the master node, which computes the final answer.

The reducer output cache requires that in the corresponding MapReduce pair of the loop body, the mapper output partition function f and the `reduce` function satisfy the following condition: if $(k_{o1}, v_{o1})\in$`reduce`(k_i, V_i), $(k_{o2}, v_{o2})\in$`reduce`(k_j, V_j), and $k_{o1}=k_{o2}$, then $f(k_i)=f(k_j)$. That is, if two `Reduce` function calls produce the same output key from two different reducer input keys, both reducer input keys must be in the same partition. This condition ensures that both keys are sent to the same reduce task. Further, f should also meet the requirements of the reducer input cache. Satisfying these requirements guarantees that the step's reducer output tuples in different iterations but with the same output key are produced on the same physical node, which ensures the usefulness of reducer output cache and the correctness of the local fixpoint evaluation. Our PageRank, descendant query, and k-means clustering implementations on HaLoop all satisfy these conditions, as do all jobs with reducers that use the same input key as output key (e.g., word count).

4.4 Mapper Input Cache

Hadoop attempts to co-locate map tasks with their input data. On a real-world Hadoop cluster like CluE [10], the rate of data-local mappers is around 60%–95% or even lower, depending on the runtime environment. HaLoop's mapper input cache aims to avoid non-local data reads in mappers during non-initial iterations. In the first iteration, if a mapper performs a non-local read on an input split, the split will be cached in the local disk of the mapper's physical node. Then, with loop-aware task scheduling, in later iterations, all mappers read data only from local disks, either from HDFS or from the local file system. The mapper input cache can be used by model-fitting applications such as k-means clustering, neural network analysis, and any other iterative algorithm consuming mapper inputs that do not change across iterations.

5 Failure Recovery

The goal of the fault-tolerance mechanism of Hadoop is to ensure that a single task failure triggers a bounded amout of recovery work. A failed map task can be rescheduled on a different node, and its input can be reloaded from the distributed file system (perhaps accessing a redundant copy). A failed reduce task, similarly rescheduled, can reconstruct its input from the mapper outputs, which are written to disk for this purpose (Figure 9(a)). In each case, only one task must be re-executed. If a node fails, multiple tasks will fail. Moreover, local map output that may be needed by other tasks will become unavailable. In this case, Hadoop reschedules the failed tasks and also reschedules the map tasks needed to recompute the lost data. In both cases, the number of tasks that must be re-executed is a small constant factor of the number of failures.

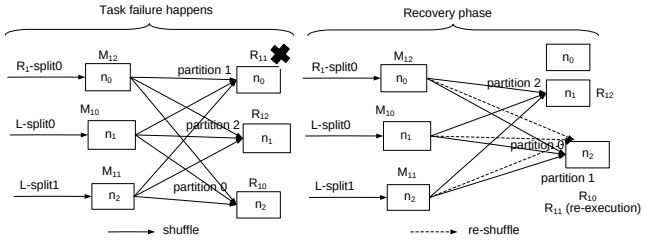
In HaLoop, the more general iterative programming model introduces the risk of *recursive recovery*, where a failure in one step of one iteration may require re-execution of tasks in all preceding steps in the same iteration or all preceding iterations. Avoiding recursive recovery is a key design goal of the fault-tolerance mechanism in HaLoop.

In previous sections, we have described how HaLoop extends Hadoop with inter-iteration caches to reduce redundant work. These caches must be reconstructed in response to failures. The key insight is that the source data for each of these caches is guaranteed to be available, assuming the fault-tolerance features of the underlying distributed filesystem are reliable. In this section, we explain how recovery occurs for each failure type.

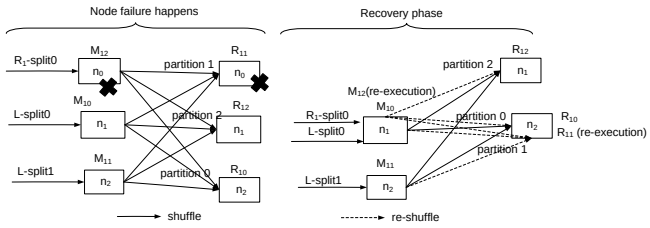
5.1 Failure Types

We consider three categories of failures:

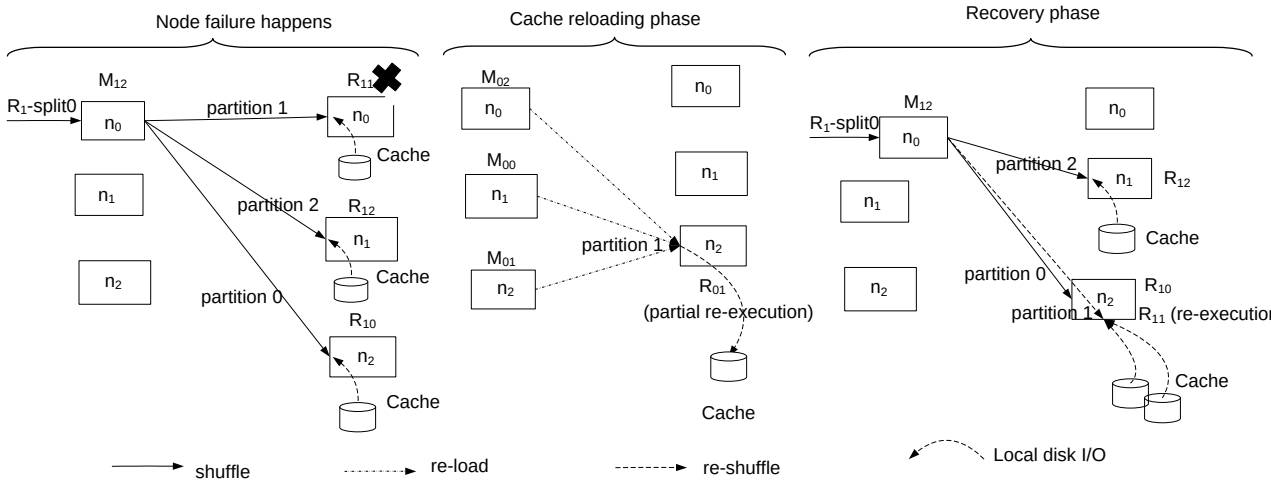
– *Task failure* A map or reduce task may fail due to programming errors (e.g., unanticipated data formats), or due to transient system errors (e.g., network timeouts during file access). A task may also be killed by the job tracker if



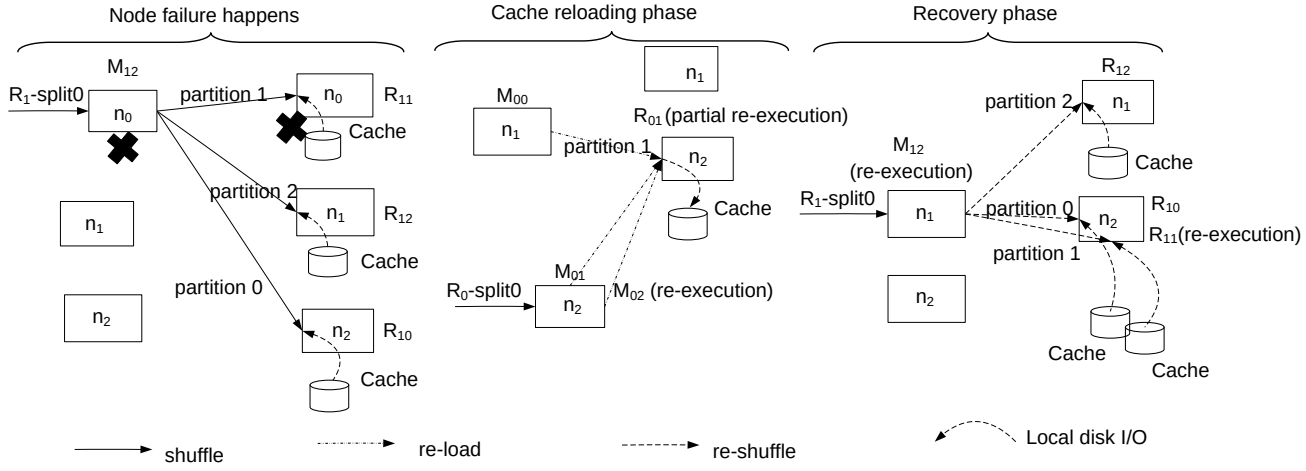
(a) When a reduce task fails in Hadoop, the reduce task is rescheduled onto another node, and the corresponding mapper outputs are accessed to reconstruct the reduce task's input. (Note that a failed task usually is rescheduled on another node, but not necessary.)



(b) If an entire node fails, the corresponding mapper output may have been lost. In this case, the map tasks are re-executed as needed.



(c) In HaLoop, a reduce task failure may involve reconstructing the reducer input cache. The procedure is similar to reducer task recovery in Hadoop, except that the cache must be reconstructed from the mapper output of iteration 0. (Note that there are only one mapper launched in iteration 1 because the invariant dataset L uses a reducer input cache.)



(d) If an entire node fails, the mapper output from iteration 0 may be lost. In this case, the corresponding map tasks are re-executed as needed.

Fig. 9 Recovery from failures affecting map and reduce tasks in HaLoop and Hadoop. A map task M_{ip} consumes input partition p in iteration i. Similar notation is used to refer to reduce tasks.

Table 1 Source Data Properties of Different Cache Options

Cache Type	Primary Data Source	Primary Data Source Location	Secondary Data Source	Secondary Data Source Location
Mapper input cache (i, j)	Step input (0, j)	distributed file system	N/A	N/A
Reducer input cache (i, j)	Mapper Output (0, j)	all mappers' local disks	Step input (0, j)	distributed file system
Reducer output cache (i, j)	Mapper Output (i-1, j)	all mappers' local disks	Step input (i-1, j)	distributed file system

it does not send a status update after some threshold delay. HaLoop inherits Hadoop's strategy for task failures: a failed task is restarted some number of times before it causes the job to fail.

– *Slave node failure*: An entire slave node, potentially hosting many tasks, may fail due to hardware or operating system errors, or a network partition may leave the node inaccessible. The master node concludes a slave node has failed if it does not receive a heartbeat message from the slave node after some user-specified duration. In these cases, all work assigned to the slave node must be rescheduled. Further, any data held on the failed node must either be reconstructed, or an alternative source for the data must be identified. We will describe slave node failure in more detail in this section.
– *Master node failure*: If the master node fails, all running jobs are considered aborted and are re-executed from scratch once the master node recovers. The next generation of Hadoop may build fail-over functionality among multiple master nodes[3]. This technique is orthogonal to what we propose in HaLoop; we consider master node failure recovery out of scope for this work.

5.2 Recovery Process: HaLoop vs. Hadoop

In both Hadoop and HaLoop, slave node failure recovery proceeds by initiating task recovery for each task running on the failed node. Task recovery requires 1) reconstituting the input for the task and 2) rescheduling the task. In HaLoop, the input for a failed task may include inter-iteration caches managed by the HaLoop system. We discuss the primary and secondary data sources used to to reconstruct each of the three different caches in this section. In both systems, recovery is transparent to user programs.

5.2.1 Source Data for Each Cache Type

In HaLoop, the mapper output (in the local file system) and the reducer output (in the distributed file system) from every step of every iteration are retained until the iterative job completes. To maximize expressive power, the API allows the output of any step in any iteration to be used as input of any future step in any future iteration. The programmer controls this behavior by providing implementations of the `GetOutputPath` and `GetInputPath` functions (see Section 2.2). However, HaLoop assumes that the output files are unique for each step and iteration; i.e., nothing is overwritten. This condition prevents any previous output from becoming unavailable for cache reconstruction.

[3] http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen

The primary data source that HaLoop uses to reconstruct the caches is described in Table 1. Recall that the mapper input cache avoids the need to pull data across the network on iterations beyond the first. Therefore, the primary data source for the mapper input cache for step j in iteration i is the original input data for step j in iteration 0. This data is stored and replicated in the underlying distributed file system, and we can therefore assume that a copy is available even in the presence of a node failure.

The reducer input cache avoids the need to reprocess static data beyond the first iteration. The primary data source for the reducer input cache for step j in iteration i is therefore the mapper output from step j in iteration 0. This primary data source is written only to local disk; it is *not* replicated to the distributed file system. Therefore, a node failure potentially could make one or more mappers' outputs permanently unavailable. In this case, the appropriate map tasks from iteration 0 are re-executed. In this case, we say that the *secondary data source* for the reducer input cache at step j is the input data for step j on iteration 0, which *is* replicated in the distributed file system.

The reducer output cache allows efficient comparison of two consecutive iterations; it is typically used for fixpoint calculations. The primary source data of the reducer output cache for step j in iteration i is the mapper output of step j in iteration $i-1$ (the previous iteration's result). As with the reducer input cache, the primary source is stored on the local disk of the mapper nodes. When a reduce task for step j in iteration i fails and re-executes on a new node, it first runs the appropriate reduce task (which has the same partition value) in step j of iteration $i-1$ to reconstruct the reduce output cache. The secondary data source is the mapper input of step j in iteration $i-1$, which is guaranteed to be available.

The primary source data information needed to reconstruct any cache are guaranteed to be available without having to recompute an entire parallel step. However, in some failure cases, one or more map tasks may need to be re-executed to reconstruct the *unavailable portion* of the primary source data; these map tasks rely on the secondary data sources described in Figure 1.

5.2.2 Recovery from Task Failures

In Hadoop, if a map or reduce task fails, the scheduler reschedules the failed task. For a failed map task, the rescheduled map task processes the same input data partition as the original task. The input data may or may not need to be copied across the network, depending on where the new task is scheduled. For a failed reduce task, the rescheduled reduce task processes the same partition of the overall mapper output; i.e., the same key groups. This partition is reconstructed from the individual mapper outputs from the

preceding phase. In both cases, the failed task is typically rescheduled on a different physical node in order to avoid repeating failures resulting from the node itself.

Figure 9(a) illustrates how a Hadoop cluster of three nodes (n_0, n_1, n_2) recovers from a reduce task failure using the PageRank example of Figure 5. Recall that each map task is labeled M_{ip}, indicating that it processes partition p in iteration i. Each reduce task is labeled similarly. Consider the second iteration (iteration 1): map tasks M_{10}, M_{11}, and M_{12} are scheduled on node n_1, n_2, and n_0, respectively, while reduce tasks R_{10}, R_{11}, and R_{12} are scheduled on node n_2, n_0 and n_1, respectively. If task R_{11} fails, the Hadoop scheduler re-schedules R_{11} to execute on node n_2. Once R_{11} starts running on n_2, the newly scheduled task recopies the data constituting partition 1 from the mapper outputs stored on local disks of n_1, n_2, and n_0, sorts and groups the copied data, and then re-executes the reduce function once per group.

In HaLoop, task recovery involves not only the re-execution of the failed task, but also the reconstruction of all relevant caches from their primary data source. Consider the reconstruction of a reducer input cache, which stores data from the first iteration to avoid unnecessary reshuffling and resorting on future iterations. Once the failed task is rescheduled, the task re-shuffles, re-sorts, and re-groups the mapper output from *from the first iteration*. This process is similar to the process Hadoop uses to recover from a reducer failure (Figure 9(a)), but must access data from the first iteration.

This process is illustrated for the PageRank application in Figure 9(c). In the second iteration (iteration 1), map task M_{12} is scheduled on node n_0, while reduce tasks R_{10}, R_{11}, and R_{12} are scheduled on nodes n_2, n_0 and n_1 respectively. If R_{11} fails, the master node reschedules it to run on node n_2. Then, the newly scheduled task discovers that its cache is unavailable and triggers reconstruction. The task on node n_2 copies mapper outputs from iteration 0 (not iteration 1). After copying, the new task on n_2 sorts and groups the shuffled data and writes them to local disk, reconstructing the cache. Finally, since the reduce input cache is ready, R_{11} gets re-executed on node n_2.

Now consider the case when the reducer output cache is enabled. The primary source data for the reducer output cache of step j is the corresponding mapper output of step j from the previous iteration $i-1$. If some mapper output is unavailable, we re-execute the corresponding map task. The failed reduce task of the current iteration is then re-executed, and the fixpoint condition is checked to bring the job back to normal execution.

Finally, consider the mapper input cache in the presence of failures. When a failed map task is rescheduled on another machine, the newly scheduled task fetches its input data partition from the distributed file system as usual, then reconstructs the cache for use in future iterations.

These cache reconstruction procedures may also be invoked during normal processing in response to scheduling conflicts. If a task cannot be scheduled on the node that holds a cache the task requires, the task can be rescheduled elsewhere and the cache reconstructed as we have described. This situation arises when a node containing a cache has no free task slots and a task is scheduled on a substitution node (Figure 6).

5.2.3 Recovery from slave node failures

In Hadoop, if a slave node fails, the scheduler reschedules all its hosted tasks on other slave nodes. Figure 9(b) illustrates how iteration 1 of PageRank job survives the failure of node n_1: tasks hosted by n_1 (M_{12} and R_{11}) are re-executed on other nodes (n_1 and n_2, respectively).

The process is similar in HaLoop. The additional failure situations introduced by the inter-iteration caches are addressed by identifying the appropriate source data (Table 1) and re-executing tasks, as we have discussed.

In Figure 9(d), we show an example of recovery from node failure for PageRank. Node n_0 fails during iteration 1. After the failure, tasks M_{12} and R_{11} are rescheduled to node n_1 and n_2 respectively. However, R_{11} depends on data that was lost when node n_1 failed. Therefore, before these two tasks can be executed, the map task from the prior iteration, M_{02}, is re-executed. Further, the reducer input cache for partition 1 on n_2 is reconstructed by pulling the appropriate mapper output across the network. Finally, M_{12} and R_{11} are re-executed, and the job proceeds.

5.3 HaLoop Scheduler Logging

In order to keep track of the binding between partitions (mapper input data partitions or reducer input partitions) and slave nodes, the HaLoop scheduler maintains a *scheduling history log*. The log records the binding between partitions and slave nodes in each step of the iterations that create caches (typically the first iteration). This information is used to inform the iteration-aware task scheduling as described in Section 3. The scheduling log also helps the failure recovery module to reconstruct caches, since the locations of the tasks of all previous steps and previous iterations, and therefore the locations of the primary and secondary data sources, are recorded.

For example, consider Figure 9(d). Prior to the failure of node n_1, the log records these bindings: (R_0-split$_0$, n_0), (L-split$_0$, n_1), and (L-split$_1$, n_2); (reduce partition 0, n_2), (reduce partition 1, n_0), and (reduce partition 2, n_1). On the next iteration, HaLoop's iteration-aware scheduler uses this information as the schedule. After the recovery, the log stores the bindings (R_0-split$_0$, n_1), (L-split$_0$, n_1), and (L-split$_1$, n_2); (reduce partition 0, n_2), (reduce partition 1, n_2),

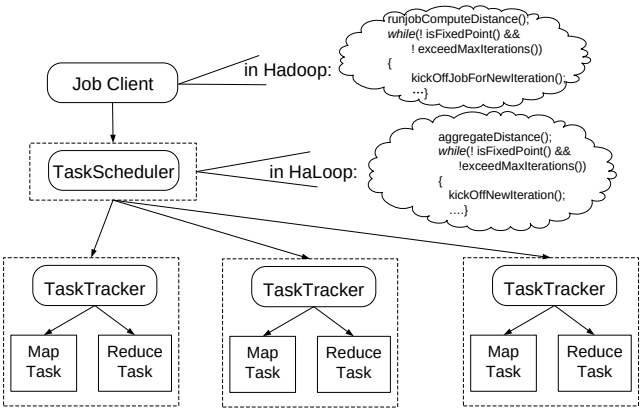


Fig. 10 Job execution: HaLoop vs. Hadoop

and (reduce partition 2, n_1), and the updated log will be used for future scheduling.

5.4 Comparisons with Other Alternatives

There are at least two alternative techniques for reducer output cache recovery: 1) We can ignore the loss of a reducer output cache. To check a fixpoint condition, we can schedule a separate MapReduce job to compare the current iteration with the previous iteration. This technique may be effective if the cost of the extra job is modest. 2) We can replicate the reducer output cache to the distributed file system, ensuring that it is available despite failures. This technique is only effective if the probability of failure is high. Both techniques are worth exploration in future work.

6 System Implementation

This section presents additional implementation details for the HaLoop system. We first provide some background on the Hadoop system and then introduce several important extensions to Hadoop in HaLoop. The HaLoop source code is publicly available at: http://code.google.com/p/haloop/.

6.1 Background on Hadoop

In Hadoop, client programs must implement the fixpoint evaluation on their own, either in a centralized way or by an extra MapReduce job. They must also decide when to launch a new MapReduce job. The Mahout [25] project has implemented multiple iterative machine learning and data mining algorithms with this approach. Figure 10 demonstrates how an iterative program is executed in Hadoop. The major building blocks of the Hadoop system include:

– *Master node daemon*. In Hadoop, interface *TaskScheduler* and class *JobInProgress* play the role of master node: they accept heartbeats from slave nodes and manage task scheduling.
– *Slave node daemon*. Class *TaskTracker* is a daemon process on every slave node. It sends heartbeats to the master node including information about completed tasks. It receives task execution commands from the master node.
– *Map and reduce task*. Class *MapTask* and *ReduceTask* are containers for user-defined *Mapper* and *Reducer* classes. These wrapper classes load, preprocess and pass data to user code. Once a *TaskTracker* gets task execution commands from the *TaskScheduler*, it kicks off a process to start a *MapTask* or *ReduceTask* process.
– *Speculative execution*. Speculative execution plays a key role in fault-tolerance. If a task's running time is much longer than that of other peers, the task does not send a status update for longer than a pre-defined period of time, or a tasktracker's heartbeat message is not received for longer than a pre-defined time-period, the Hadoop task scheduler will schedule speculative tasks to re-execute the potentially failed tasks. If a speculative task execution is based on a wrong decision, the task will be killed after the original "false-negative" task completes.

6.2 HaLoop Extensions to Hadoop

In the HaLoop framework, we extended and modified Hadoop as follows:

– *Loop control and API*. We implemented HaLoop's loop control and task scheduler by implementing our own *TaskScheduler* and modifying the class *JobInProgress*. Additionally, HaLoop provides an extended API to facilitate client programming, with functions to set up the loop body, associate the input files with each iteration, specify a loop termination condition, enable/disable caches. *JobConf* class represents a client job and hosts these APIs.
– *Caching*. We implemented HaLoop's caching mechanisms by modifying classes *MapTask*, *ReduceTask* and *TaskTracker*. In map/reduce tasks, HaLoop creates a directory in the local file system to store the cached data. The directory is under the task's working directory, and is tagged with the iteration number and step number. With this approach, a task accessing the cache in the future can access the data for a specitic iteration and step number as needed. After the iterative job finishes, all files storing cached data are deleted.
– *Fixpoint evaluation*. HaLoop evaluates the fixpoint in a distributed fashion. After the reduce phase of the specified step, a *ReduceTask* computes the sum of the user-defined distances between the current output and that of the previous iteration by executing the user-defined distance function. Then, the host *TaskTracker* sends the aggregated value back to *JobInProgress*. *JobInProgress* computes the

sum of the locally pre-aggregated distance values returned by each *TaskTracker* and compares the overall distance value with the fixpoint threshold (set by the application). If the distance is less than the specified threshold or the current iteration number is greater than the maximum number of iterations set by the application, *JobInProgress* will raise a "job complete" event to terminate the job execution. Otherwise, *JobInProgress* will put a number of tasks in its task queue to start a new iteration. Figure 10 also shows how HaLoop executes a job. In particular, we see that the *TaskScheduler* manages the lifecycle of an iterative job execution.

– *Failure Recovery*. HaLoop reuses speculative execution mechanism to achieve fault-tolerance. In contrast with Hadoop, HaLoop adds tags to speculative tasks to indicate they are to be used for recovery from task failure or slave node failure. Cache reconstruction is handled by a *RecoveryTask* class. A recovery task is initiated as a a speculative task. The implementation follows the two kinds of recovery discussed in Section 5.2.

7 Experimental Evaluation

We compared the performance of iterative data analysis applications on HaLoop and Hadoop. Since use of the reducer input cache, reducer output cache, and mapper input cache are independent options, we evaluated them separately in Sections 7.1–7.3. Additionally, we evaluated the overhead for failure recovery in Section 7.4 and the programming effort with HaLoop programming interface in Section 7.5.

7.1 Evaluation of Reducer Input Cache

This suite of experiments used virtual machine clusters of 50 and 90 slave nodes in Amazon's Elastic Compute Cloud (EC2). There is always one master node. The applications were PageRank and descendant query. Both are implemented in both HaLoop (using our new programming model) and Hadoop (using the traditional driver approach).

All nodes in these experiments are default Amazon small instances[4], with 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of instance storage (150 GB plus 10 GB for the root partition), 32-bit platform, and moderate I/O performance.

We used both semi-synthetic and real-world datasets: Livejournal (18GB, social network data), Triples (120GB, semantic web data) and Freebase (12GB, concept linkage graph).

[4] http://aws.amazon.com/ec2/instance-types/

Name	**Nodes**	**Edges**	size
Livejournal	4,847,571	68, 993,773	18GB
Triples	1,464,829,200	1,649,506,981	120GB
Freebase	7,024,741	154,544,312	12GB

Fig. 11 Dataset descriptions

Livejournal is a semi-synthetic dataset generated from a base real-world dataset[5]. The base dataset consists of all edge tuples in a social network, and its size is 1GB. We substituted all node identifiers with longer strings to make the dataset larger without changing the network structure. The extended Livejournal dataset is 18GB.

Triples is an RDF benchmark (resource description framework) graph dataset from the billion triple challenge[6]. Each raw tuple in Triples is a line of ⟨subject, predicate, object, context⟩. We ignore the predicate and context columns, and treat the dataset as a graph where each unique string that appears as either a subject or an object is a node, and each ⟨subject, object⟩ tuple is an edge. The filtered Triples dataset is 120GB in size.

Freebase is another real-world dataset[7], where many concepts are connected by various relationships. If we search for a keyword or concept ID on the Freebase website, it returns the description of a matched concept, as well as outgoing links to the connected concepts. Therefore, we filter the Freebase raw dataset (which is the crawl of the whole Freebase website) to extract tuples of the form ⟨concept_id1, concept_id2⟩. The filtered Freebase dataset (12.2GB in total) is actually a concept-connection graph, where each unique concept_id is a node and each tuple represents an edge. Detailed dataset statistics are in Figure 11.

We run PageRank on the Livejournal and Freebase datasets because ranking on social network and crawl graphs makes sense in practice. Similarly, we run the descendant query on the Livejournal and Triples datasets. In the social network application, a descendant query finds one's friend network, while for the RDF triples, such a query finds a subject's impacted scope. The initial source node in the query is chosen at random.

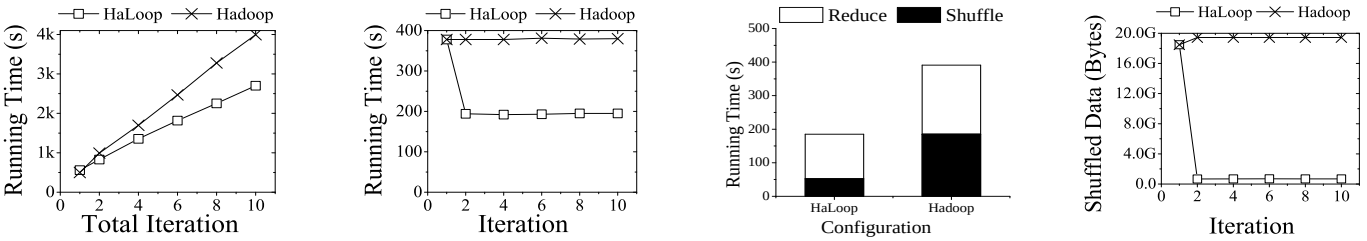
By default, experiments on Livejournal are run on a 50-node cluster, while experiments for both Triples and Freebase are executed on a 90-node cluster.

We executed the PageRank query on the Livejournal and Freebase datasets and the descendant query on the Livejournal and Triples datasets. Figures 12–15 show the results for Hadoop and HaLoop. The number of reduce tasks is set to the number of slave nodes. The performance with fail-overs has not been quantified; all experimental results are obtained without any node failures.

[5] http://snap.stanford.edu/data/

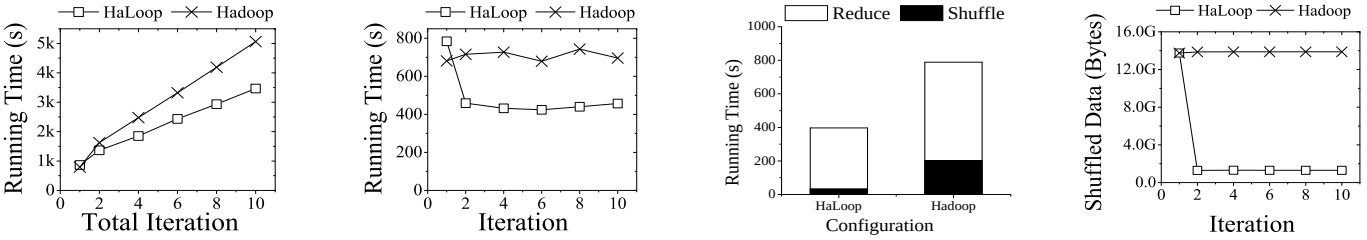
[6] http://challenge.semanticweb.org/

[7] http://www.freebase.com/



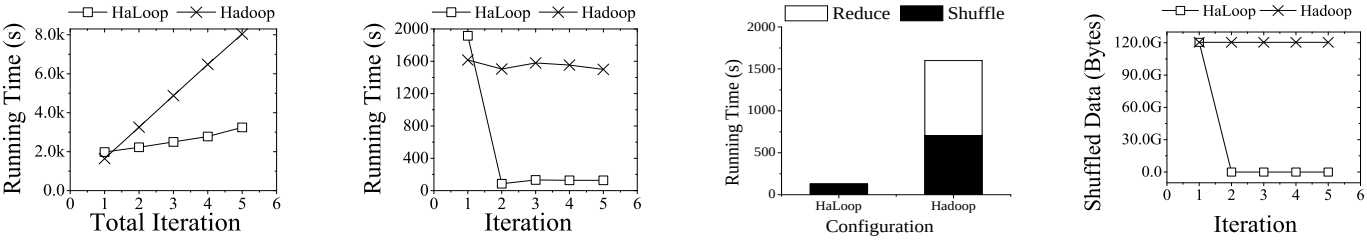
(a) Overall Performance (b) Join Step (c) Cost Distribution (d) Shuffled Bytes

Fig. 12 PageRank performance: HaLoop vs. Hadoop (Livejournal dataset, 50 nodes)



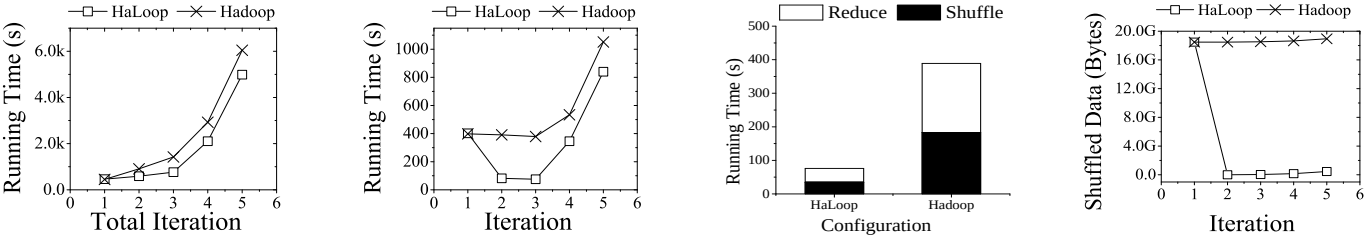
(a) Overall Performance (b) Join Step (c) Cost Distribution (d) Shuffled Bytes

Fig. 13 PageRank performance: HaLoop vs. Hadoop (Freebase dataset, 90 nodes)



(a) Overall Performance (b) Join Step (c) Cost Distribution (d) Shuffled Bytes

Fig. 14 Descendant query performance: HaLoop vs. Hadoop (Triples dataset, 90 nodes)



(a) Overall Performance (b) Join Step (c) Cost Distribution (d) Shuffled Bytes

Fig. 15 Descendant query performance: HaLoop vs. Hadoop (Livejournal dataset, 50 nodes)

Overall, as the figures show, for a 10-iteration job, HaLoop lowers the runtime by 1.85 on average when the reducer input cache is used. As we discuss later, the reducer output cache creates an additional gap between Hadoop and HaLoop but the impact is less significant on overall runtime. We now present these results in more detail.

Overall Run Time. In this experiment, we used `SetMaxNumOfIterations`, rather than `fixedPoint-Threshold` and `ResultDistance`, to specify the loop termination condition. The results are plotted in Figure 12(a), Figure 13(a), Figure 14(a), and Figure 15(a).

In the PageRank algorithm, there are two steps in every iteration: join and aggregation. The running time in Fig-

ure 12(a) and Figure 13(a) is the sum of join time and aggregation time over all iterations. In the descendant query algorithm, there are also two steps per iteration: join and duplicate elimination. The running time in Figure 14(a) and Figure 15(a) is the sum of join time and "duplicate elimination" time over all iterations.

HaLoop always performs better than Hadoop. The descendant query on the Triples dataset has the best improvement, PageRank on Livejournal and Freebase have intermediate gains, but the descendant query on the Livejournal dataset has the least improvement. Livejournal is a social network dataset with high fan-out and reachability. As a result, the descendant query in later iterations (>3) produces so many duplicates that duplicate elimination dominates the cost, and HaLoop's caching mechanism does not significantly reduce overall runtime. In contrast, the Triples dataset is less connected, thus the join step is the dominant cost and the cache is crucial.

Join Step Run Time. HaLoop's task scheduling and reducer input cache potentially reduce join step time, but do not reduce the cost of the "duplicate elimination" step for the descendant query, nor the final aggregation step in PageRank. Thus, to partially explain why overall job running time is shorter with HaLooop, we compare the performance of the join step in each iteration. Figure 12(b), Figure 13(b), Figure 14(b), and Figure 15(b) plot join time in each iteration. HaLoop significantly outperforms Hadoop.

In the first iteration, HaLoop is slower than Hadoop, as shown in (a) and (b) of all four figures. The reason is that HaLoop performs additional work in the first iteration: HaLoop caches the sorted and grouped data on each reducer's local disks, creates an index for the cached data, and stores the index to disk. That is, in the first iteration, HaLoop does the exact same thing as Hadoop, but also writes caches to local disk.

Cost Distribution for Join Step. To better understand HaLoop's improvements to each phase, we compared the cost distribution of the join step across Map and Reduce phases. Figure 12(c), Figure 13(c), Figure 14(c), and Figure 15(c) show the cost distribution of the join step in a certain iteration (here it is iteration 3). The measurement is time spent on each phase. In both HaLoop and Hadoop, reducers start to copy data immediately after the first mapper completes. "Shuffle time" is normally the time between reducers starting to copy map output data, and reducers starting to sort copied data; shuffling is concurrent with the rest of the unfinished mappers. The first completed mapper's running time in the two algorithms is very short, e.g., 1–5 seconds to read data from one 64MB HDFS block. If we were to plot the first mapper's running time as "map phase", the duration would be too brief to be visible compared to shuffle phase and reduce phase. Therefore we let the "shuffle time" in the plots be the usual shuffle time *plus* the first completed mapper's running time. The "reduce time" in the plots is the total time a reducer spends after the shuffle phase, including sorting and grouping, as well as accumulated `Reduce` function call time. Note that in the plots, "shuffle time" plus "reduce time" constitutes what we have referred to as the "join step". Considering all four plots, we conclude that HaLoop outperforms Hadoop in both phases.

The "reduce" bar is not visible in Figure 14(c), although it is present. The "reduce time" is not 0, but rather very short compared to "shuffle" bar. It takes advantage of the index HaLoop creates for the cache data. Then the join between ΔS_i and F will use an index seek to search qualified tuples in the cache of F. Also, in each iteration, there are few new records produced, so the join's selectivity on F is very low. Thus the cost becomes negligible. By contrast, for PageRank, the index does not help much, because the selectivity is high. For the descendants query on Livejournal (Figure 15), in iteration>3, the index does not help either, because the selectivity becomes high.

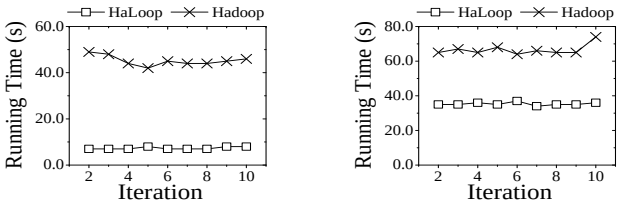
I/O in Shuffle Phase of Join Step. To tell how much shuffling I/O is saved, we compared the amount of shuffled data in the join step of each iteration. Since HaLoop caches loop-invariant data, the overhead of shuffling these invariant data are completely avoided. These savings contribute an important part of the overall performance improvement. Figure 12(d), Figure 13(d), Figure 14(d), and Figure 15(d) plot the sizes of shuffled data. On average, HaLoop's join step shuffles 4% as much data as Hadoop's does.

7.2 Evaluation of Reducer Output Cache

This experiment shares the same hardware and dataset as the reducer input cache experiments. To see how effective HaLoop's reducer output cache is, we compared the cost of fixpoint evaluation in each iteration. Since descendant query has a trivial fixpoint evaluation step that only requires testing to see if a file is empty, we run the PageRank implementation in Section 10.1 on Livejournal and Freebase. In the Hadoop implementation, the fixpoint evaluation is implemented by an extra MapReduce job. On average, compared with Hadoop, HaLoop reduces the cost of this step to 40%, by taking advantage of the reducer output cache and a built-in distributed fixpoint evaluation. Figure 16(a) and (b) shows the time spent on fixpoint evaluation in each iteration.

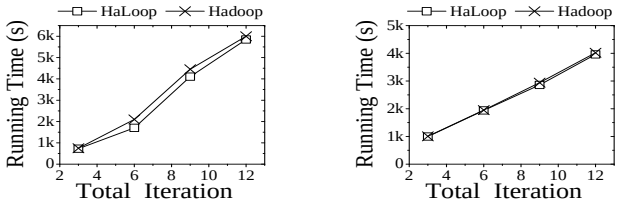
7.3 Evaluation of Mapper Input Cache

Since the mapper input cache aims to reduce data transportation between slave nodes but we do not know the disk I/O implementations of EC2 virtual machines, this suite of experiments uses an 8-node physical machine cluster. All



(a) Livejournal, 50 nodes (b) Freebase, 90 nodes

Fig. 16 Fixpoint evaluation overhead in PageRank: HaLoop vs. Hadoop



(a) Cosmo-dark, 8 nodes (b) Cosmo-gas, 8 nodes

Fig. 17 Performance of k-means: HaLoop vs. Hadoop

nodes in these experiments contain a 2.60GHz dual quad-core Intel Xeon CPU with 16GB of RAM.

PageRank and descendant query cannot utilize the mapper input cache because their inputs change from iteration to iteration. Thus, the application used in the evaluation is the k-means clustering algorithm. We used two real-world Astronomy datasets (multi-dimensional tuples): cosmo-dark (46GB) and cosmo-gas (54GB). The Cosmo dataset[8] is a snapshot from an astronomy simulation of the universe. The simulation covered a volume of 110 million light years on a side, with 900 million particles total. Tuples in Cosmo are multi-dimensional vectors.

We vary the number of total iterations, and plot the algorithm running time in Figure 17. The mapper locality rate is around 95% since there are not concurrent jobs in our lab HaLoop cluster. By avoiding non-local data loading, HaLoop performs marginally better than Hadoop.

7.4 Evaluation of Failure Recovery

We conducted experiments with either *speculative execution* enabled (then there are "logical" task failures, for example, a very slow task is treated as "failed"), or failures injected as slave node failures (by killing one task tracker process at the last iteration's cache-reading MapReduce step). These experiments uses an 8-node physical machine cluster. All nodes in these experiments contain a 2.60GHz dual quad-core Intel Xeon CPU with 16GB of RAM. We use the synthetic livejournal dataset at a scale of 18GB to run PageRank application. We vary the number of total iterations, and plot the program running time in Figure 9, where Hadoop(p) and HaLoop(p) are for the case of speculative execution, and Hadoop(n) and HaLoop(n) are for the case when slave node failure is injected.

[8] http://nuage.cs.washington.edu/benchmark/astro-nbody/dataset.php

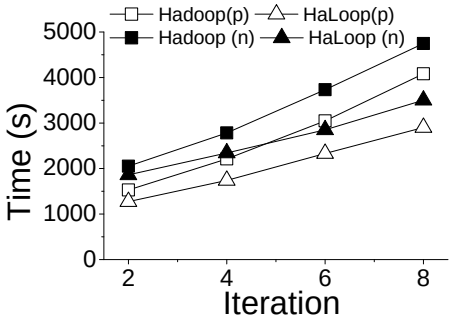


Fig. 18 Overheads of failure recovery: HaLoop vs. Hadoop

7.5 Evaluation of Programming Effort

We measure the programming effort by line of code in JAVA, including package imports, comments, and actual code. Comments spread near-uniformly across source code files. For the three applications we implemented, we report in Figure 19 the line of code for mapper/reducer implementations (shared between Hadoop and HaLoop), extra code on Hadoop (mostly for loop control), and extra code on HaLoop (including hook function implementations and loop body construction). From the figure, one can find that HaLoop incurs a small amount of extra programming overheads.

8 Related Work

Parallel database systems [13] partition data storage and parallelize query workloads to achieve better performance. However, they are sensitive to failures and have not been shown to scale to thousands of nodes. Various optimization techniques for evaluating recursive queries have been proposed in the literature [5,41]. The existing work has not been shown to operate at large scale. Further, most of these techniques are orthogonal to our research; we provide a low-level foundation for implementing data-intensive iterative programs.

More recently, MapReduce [12] has emerged as a popular alternative for massive-scale parallel data analysis in shared-nothing clusters. Hadoop [17] is an open-source implementation of MapReduce. MapReduce has been followed by a series of related system platforms for data-intensive computing, including Dryad [21], Clustera [14], Hyracks [7], Nephele/PACTs [6] and Hadoop++ [15]. However, like Hadoop, none of these systems provides explicit

App	Mapper/Reducer Loc	Extra (Hadoop)	Extra (HaLoop)
PageRank	302	120	166
Descendant query	318	90	183
k-means	291	43	79

Fig. 19 Code size (lines of code)

support and optimizations for iterative or recursive types of analysis. Hive [20], Pig [30], SCOPE [9] and HadoopDB [1] provide high-level language supports for data processing on Hadoop or Dryad, but none of them supports recursive queries. Some work is related to MapReduce but orthogonal to HaLoop: ParaTimer [28] proposes an accurate progress indicator for MapReduce DAGs; literature [34] investigates efficient fuzzy join methods on MapReduce.

MapReduce online [11] modifies Hadoop to enable streaming between mappers and reducers, and also allows to loop over several map-reduce phases (for example, streaming from reducers to mappers) to naturally support iterative computations. Various system challenges such as fault-tolerance and task scheduling have been addressed in MapReduce online to make the architecture work. However, MapReduce online does not provide caching options across iterations at the infrastructure level because it is not aware of loops.

Mahout [25] is a project whose goal is to build a set of scalable machine learning libraries on top of Hadoop. Since most machine learning algorithms are model fitting applications, nearly all of them involve iterative programs. Mahout uses an outside driver program to control the loops, and new MapReduce jobs are launched in each iteration. The drawback of this approach has been discussed in Section 1. Like Mahout, we are trying to help iterative data analysis algorithms work on scalable architectures, but we are different in that we are modifying the fundamental system: we inject the iterative capability into a MapReduce engine.

Nova [29] is a workflow manager built on top of an unmodified Pig/Hadoop software stack, which processes continuous data using a graph of Pig programs. Nova supports incremental data processing. However, without changing Hadoop's task scheduling mechanims, application programmers cannot explore cache (either disk or memory) locality to avoid redundant shuffling of static data in iterative data processing such as the PageRank example.

Delay scheduling [38] improves disk locality of the map phase in multi-user scenarios by adding a wait time before scheduling potentially non-local map tasks. This delay improves the probability that a data-local node will become available. Scarlett [3] addresses the same problem by placing blocks in the distributed file system according to data popularity. However, these techniques do not exploit the data flow and dependency information we exploit to make Hadoop "iteration-aware". Moreover, they are orthogonal to our contributions related to using reduce-side caches to avoid redundant shuffling, sorting, and grouping. This problem and solution are only relevant in the context of iterative computations. General performance optimizations for iterative jobs in a multi-user environment represent a promising direction of investigation, but remain future work.

A study by Ganesh et al. [4] shows that disk-locality is not that important when network performance is high and job input/output data are compressed. However, HaLoop targets a commodity network environment (e.g., EC2) where network I/O performance is moderate. Improving performance by compressing data between map and reduce phases may be beneficial, but is largely orthogonal to our work.

Twister [16] is a stream-based MapReduce framework that supports iterative programs, in which mappers and reducers are long running with distributed memory caches to improve performance. Twister provides fault-tolerance by checkpointing and replicating each iteration's output to distributed file system. Due to the elimination of mapper output materialization, Twister has no intra-iteration fault-tolerance; a single failure will cause re-execution of an entire loop body (which may involve several MapReduce steps). Additionally, Twister does not provide the reducer-side cache mechanisms to eliminate redundant data shuffling, sorting and grouping.

Spark [40, 39] is a system that supports dataset reusing across parallel operations for iterative machine learning algorithms. The data reuse is achieved by *resilient distributed datasets*, which are cached in the memory of cluster machines. The Spark architecture is suitable for clusters having tens of gigabytes of memory per node (Ganesh et al. report that a cluster at Facebook has 16GB - 32GB per node). Our initial design goal is for clusters having only around 1 gigabytes of memory per node, thus we did not focus on buffer management.

We consider memory-based processing orthogonal to our approach. Similar to Twister and Spark, we speculate that HaLoop could be extended with a memory-based buffer manager layer built on task trackers to further improve efficiency, without changes to the task scheduler and caching mechanisms we explore. Different in-memory cache management policies such as LRU, FIFO, and MRU are also considered orthogonal to this work; evaluation and optimization of such policies in this context represent important open questions. Currently, we follow the Hadoop model that emphasizes disk-oriented processing for fault-tolerance and scalability beyond main memory sizes.

BOOM [2] uses the Overlog [24] language to implement API-compatible with Hadoop and HDFS, and adds high-availability and debugging support. Thus, recursive queries could be naturally supported. However, it does not explore the direction of distributed caching to improve the performance.

Piccolo [33] provides system support for iterative data processing, using partitioned table data model with user- defined partitioning. The partitioned tables are in distributed memory and users can give locality hint to improve locality. Iterative applications use message-passing to update states in partitioned tables. Piccolo achieves fault-tolerance by writing consistent global checkpointing: a global snapshot of the program state. This is different from the data checkpointing and materialization mechanisms in all other systems. The comparison on the two basic fault-tolerance mechanisms is worth further investigation. Our work has focused on identifying the minimal changes required to Hadoop to support efficient iterative computation.

Pregel [26] is a distributed system for processing large-size graph datasets, where each vertex receiving messages, updating status, and sending messages out independently. However, Pregel cannot support non-graph iterative computations such as the k-means. Pregel achieves fault-tolerance by checkpointing vertices' states after each iteration. Thus it requires fully re-executing all tasks(vertices) in an iteration from scratch on all nodes to recover from a failure, while MapReduce and HaLoop only require re-executing those failed tasks as well as a limited number of dependent tasks.

Last but not the least, compared to Spark, BOOM, Piccolo and Pregel, HaLoop can support easy migration from those already widely-existing Hadoop iterative applications while others may require building applications from scratch.

9 Conclusion and Future Work

This paper presents the design, implementation, and evaluation of HaLoop, a novel parallel and distributed system that supports large-scale iterative data analysis applications. HaLoop is built on top of Hadoop and extends it with a new programming model and several important optimizations that include (1) a loop-aware task scheduler, (2) loop-invariant data caching, and (3) caching for efficient fixpoint verification. Besides, HaLoop employs similar failure recovery mechanisms to Hadoop and jobs can sustain process failures and slave node failures. We evaluated our HaLoop prototype on several large datasets and iterative queries. Our results demonstrate that pushing support for iterative programs into the MapReduce engine greatly improves the overall performance of iterative data analysis applications. In future work, we would like to implement a simplified Datalog evaluation engine on top of HaLoop, to enable large-scale iterative data analysis programmed in a declarative way.

Acknowledgements

The HaLoop project is partially supported by NSF CluE grants IIS-0844572 and IIS-0844580, NSF CAREER Award IIS-0845397, NSF grants CNS-0855252 and IIS-0910989, Woods Hole Oceanographic Institute Grant OCE-0418967, Amazon, University of Washington eScience Institute, and the Yahoo! Key Scientific Challenges program. Thanks for suggestions and comments from Michael J. Carey, Rares Vernica, Vinayak R. Borkar, Hongbo Deng, Congle Zhang, and the anonymous reviewers.

References

1. Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *VLDB*, 2(1):922–933, 2009.
2. Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, pages 223–236, 2010.
3. Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert G. Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *EuroSys*, pages 287–300, 2011.
4. Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, 2011.
5. François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD Conference*, pages 16–52, 1986.
6. Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.
7. Vinayak Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE Conference*, 2011.
8. Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.
9. Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.
10. Cluster Exploratory (CluE) program. `http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm`. Accessed July 7, 2010.
11. Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *NSDI 2010*, 2010.
12. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
13. David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

14. David J. DeWitt, Erik Paulson, Eric Robinson, Jeffrey F. Naughton, Joshua Royalty, Srinath Shankar, and Andrew Krioukov. Clustera: an integrated computation and data management system. *PVLDB*, 1(1):28–41, 2008.
15. Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010.
16. Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.
17. Hadoop. `http://hadoop.apache.org/`. Accessed July 7, 2010.
18. Martin T. Hagan, Howard B. Demuth, and Mark H. Beale. *Neural Network Design*. PWS Publishing, 1996.
19. Hdfs. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`. Accessed July 7, 2010.
20. Hive. `http://hadoop.apache.org/hive/`. Accessed July 7, 2010.
21. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.
22. Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.
23. Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.
24. Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *SOSP*, pages 75–90, 2005.
25. Mahout. `http://lucene.apache.org/mahout/`. Accessed July 7, 2010.
26. Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.
27. Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In *SIGMETRICS*, pages 50–60, 2005.
28. Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *SIGMOD Conference*, pages 507–518, 2010.
29. Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B. N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. Nova: continuous pig/hadoop workflows. In *SIGMOD Conference*, pages 1081–1090, 2011.
30. Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
31. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.
32. Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178, 2009.
33. Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.
34. Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD Conference*, pages 495–506, 2010.
35. Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.
36. Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.
37. Keith Wiley, Andrew Connolly, Simon Krughoff, Je Gardner, Magdalena Balazinska, Bill Howe, YongChul Kwon, and Yingyi Bu. Astronomical image processing with hadoop. In *Astronomical Data Analysis Software and Systems*, 2010.
38. Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010.
39. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, Jul 2011.
40. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.
41. Weining Zhang, Ke Wang, and Siu-Cheung Chau. Data partition and parallel evaluation of datalog programs. *IEEE Trans. Knowl. Data Eng.*, 7(1):163–176, 1995.

10 Appendix: Application Implementations

In this section, we discuss how to implement iterative applications on top of HaLoop, using three examples: PageRank, descendant query, and k-means. The source code of the three examples is publicly available at: http://code.google.com/p/haloop/.

10.1 PageRank Implementation

Let us walk through how PageRank (from Example 1) is implemented on top of HaLoop. Before the rank calculation loop, there is a pre-process MapReduce job to count outgoing links for each URL, where a temporary table C containing (URL, count) pairs is produced. Without this pre-process step, the join step's reducers need to hold input value objects in memory to get the outgoing URL count for each input key (source URL), which will waste CPU resources on JAVA garbage collections. There are two steps in PageRank's loop body: one is to join R_i (tagged as "0"), C (tagged as "1") and L (tagged as "2") and populate ranks from source URLs to destination URLs; the other is to aggregate ranks on each destination URL.

The join step comprises of mapper *ComputeRankMap* and reducer *ComputeRankReduce*. This MapReduce pair is similar to the reduce-side join in literature [36], where reducer input data are secondarily sorted on tables' tag. Therefore, at the reducer side, input values sharing the same input key are sorted and grouped by their source table tag (R_i tuple first, C tuple second, and L tuples third). Different from pure join, in each reduce function call, the *ComputeRankReduce* calculates the populated rank for destination URLs (values after the first two in the value iterator), where each destination URL's rank is assigned to the source URL's

```
public class RankLoopInputOutput implements LoopInputOutput {

        @Override
        public List<String> getInputPaths(JobConf conf, int iteration, int step) {
                List<String> paths = new ArrayList<String>();
                int currentPass = 2 * iteration + step;
                if(step == 0) {
                        paths.add(conf.getOutputPath() + "/count");
                        paths.add(conf.getInputPath());
                }
                paths.add(conf.getOutputPath() + "/i" + (currentPass - 1));
                return paths;
        }

        @Override
        public String getOutputPath(JobConf conf, int iteration, int step) {
                int currentPass = 2 * iteration + step;
                return (conf.getOutputPath() + "/i" + currentPass);
        }

}
public class RankReduceCacheFilter implements CacheFilter {

        @Override
        public boolean isCache(Object key, Object value, int id) {
                if (id <= 1)
                        return false;
                else
                        return true;
        }
}
public static void main(String[] args) {
        ...
        /**
         * Join map/reduce step, to populate rank value from
         * source to destination
         */
        JobConf conf1 = new JobConf();
        conf1.setMapperClass(NaivePageRank.ComputeRankMap.class);
        conf1.setReducerClass(NaivePageRank.ComputeRankReduce.class);
         ...

        /**
         * Aggregate map/reduce step, to aggregate rank value
        * for each unique url
         */
        JobConf conf2 = new JobConf();
        conf2.setMapperClass(NaivePageRank.RankAggregateMapper.class);
        conf2.setReducerClass(NaivePageRank.RankAggregateReducer.class);
        ...

        /**
         * set the as-a-whole iterative job conf
         */
        JobConf conf = new JobConf(PageRankNew.class);
        conf.setNumReduceTasks(numReducers);
        conf.setLoopInputOutput(RankLoopInputOutput.class);
        conf.setLoopReduceCacheFilter(RankReduceCacheFilter.class);
        // set up the m-r step pipeline
        conf.setStepConf(0, conf1);
        conf.setStepConf(1, conf2);
        conf.setIterative(true);
        conf.setNumIterations(specIteration);
        JobClient.runJob(conf);
}
```

Fig. 20 The loop setting implementation for the PageRank example.

rank (the first one in value iterator) divided by the number (the second one in value iterator) of destination URLs.

The aggregation step includes *RankAggregateMapper* and *RankAggregateReduce*, where *RankAggregateMap* reads raw ranks produced by *ComputeRankReduce*, and *RankAggregateReduce* sums the local ranks for each URL. Combiner is enabled to improve efficiency.

Figure 20 shows the loop setting implementation for PageRank. *RankLoopInputOutput* specifies that L, C and R_i are loaded for step 0, but step 1 only consumes the output from step 0. The *RankReduceCacheFilter* specifies that tuples from L and C are cached, but tuples from R_i is not. In

```
public class DescendantLoopInputOutput implements LoopInputOutput {

        @Override
        public List<String> getInputPaths(JobConf conf, int iteration, int step) {
                List<String> paths = new ArrayList<String>();
                int currentPass = 2 * iteration + step;
                if (step == 0) {
                        // join step
                        paths.add(conf.getOutputPath() + "/i" + (currentPass - 1));
                        paths.add(conf.getInputPath());
                }
                if (step == 1) {
                        // for the duplicate elimination step
                        for (int i = 1; i < currentPass; i += 2)
                                paths.add(conf.getOutputPath() + "/i" + i);
                        paths.add(conf.getOutputPath() + "/i" + (currentPass - 1));
                }
                return paths;
        }

        @Override
        public String getOutputPath(JobConf conf, int iteration, int step) {
                int currentPass = 2 * iteration + step;
                return (conf.getOutputPath() + "/i" + currentPass);
        }

}
public class DescendantReduceCacheFilter implements CacheFilter {

        // table tag for \delta S in the join
        Text tag0 = new Text("0");

        // table tag for L in the join
        Text tag1 = new Text("1");

        @Override
        public boolean isCache(Object key, Object value, int id) {
                TextPair tv = (TextPair) value;
                else if (tv.getSecond().equals(tag1))
                        return true;
                else
                        return true;
        }

}
public static void main(String[] args) {
        ...
        // Join map/reduce step
        JobConf job1 = new JobConf(NaiveDescendant.class);
        job1.setMapperClass(JoinMap.class);
        job1.setReducerClass(JoinReduce.class);
           ...

        //Duplicate removal map/reduce step
        JobConf job2 = new JobConf(NaiveDescendant.class);
        job2.setMapperClass(DuplicateEliminateMap.class);
        job2.setReducerClass(DuplicateEliminateReduce.class);
        job2.setNumReduceTasks(numReducers);
         ...

        // set the as-a-whole iterative join job conf
        conf = new JobConf(Descendant.class);
        conf.setLoopInputOutput(DescendantLoopInputOutput.class);
        conf.setLoopReduceCacheFilter(DescendantReduceCacheFilter.class);

        // set up the m-r step pipeline
        conf.setStepConf(0, job1);
        conf.setStepConf(1, job2);
        conf.setIterative(true);
        conf.setNumIterations(specIteration);
        JobClient.runJob(conf);
           ...
}
```

Fig. 21 The loop setting implementation for the descendant query example,

the *main* function, two job configurations are created, each one for a step. Then, a container job configuration is created, and it sets the loop body and the loop setting implementation classes.

10.2 Descendant Query Implementation

Similar to PageRank example, the loop body of descendant query also has two steps: one is join (to find friends-of-friends by looking one hop further), and the other one is duplicate elimination (to remove duplicates in the extended friends set). The join step is also implemented in the same way as the reduce-side join in literature [36]. After new friends are produced by the join step, the duplicate elimination step removes duplicates from the latest discovered friends, considering all the friends produced so far. Here, MapReduce pair *JoinMap* and *JoinReduce* compose the join step, while MapReduce pair *DuplicateEliminationMap* and *DuplicateEliminationReduce* form the duplicate elimination step.

Figure 21 demonstrates the implementation for loop setting interfaces and the *main* function. In class *DescendantLoopInputOutput*, it returns different input paths for join step (loads the latest discovered friends) and duplicate elimination step (loads all friends discovered so far). *DescendantReduceCacheFilter* specifies that tuples from initial friends table F (tagged with "1") are cached at reducer side, and tuples from latest generated friends table ΔS_i (tagged with "0") are not cached. Similar to the PageRank example, the *main* function configures loop body steps and sets up the three loop setting implementation classes.

10.3 K-means Implementation

K-means clustering is another popular iterative data analysis algorithm that can be implemented on top of HaLoop. Unlike the previous two examples, however, k-means takes advantage of the mapper input cache rather than the reducer input/output cache, because the input data to mappers at each iteration are invariant, while the reducer input data keep changing. From iteration to iteration, reducers only output the current k cluster means, and there is a final classification job after the loop to assign each tuple to a cluster. Thus, in the iterative job, the output from each iteration has a very small size (only k cluster means), thus there is no need to enable reducer output cache.

The implementation of mapper (*KMeansMapper*) and reducer (*KMeansReducer*) is similar to that in Mahout [25]. The mapper assigns tuples to the closest cluster, and the reducer re-calculate the means of clusters. Also, the combiner is enabled for local aggregation to reduce data shuffling. Figure 22 shows the loop setting implementation of k-means. It is very straightforward since there is only one step in an iteration. Every iteration, the input path is the same (by *KMeansLoopInputOutput*). Every tuple is qualified for caching (by *KMeansLoopMapCacheFilter*). Similarly, the *main* function sets up loop body and other implementation classes.

```
public class KMeansLoopInputOutput implements LoopInputOutput {

        @Override
        public List<String> getInputPaths(JobConf conf, int iteration, int step) {
                List<String> paths = new ArrayList<String>();
                // only input the dataset, cluster means are
                //read from HDFS in mappers
                paths.add(conf.getInputPath());
                return paths;
        }

        @Override
        public String getOutputPath(JobConf conf, int iteration, int step) {
                return (conf.getOutputPath() + "/i" + iteration);
        }

}
public class KMeansLoopMapCacheFilter implements  CacheFilter {

        @Override
        public boolean isCache(Object key, Object value, int id) {
                // cache every tuple
                return true;
        }

}
public static void main(String[] args){
      ...
      /**
       * kmeans loop body: only one map/reduce step
       */
      JobConf conf = new JobConf(KMeans.class);
      conf.setMapperClass(KMeansMapper.class);
      conf.setCombinerClass(KMeansReducer.class);
      conf.setReducerClass(KMeansReducer.class);
      ...

      // as-a-while iterative job
      JobConf job = new JobConf(KMeans.class);
      job.setStepConf(0, conf);
      job.setIterative(true);
      job.setLoopMapCacheFilter(KMeansLoopMapCacheFilter.class);
      job.setLoopInputOutput(KMeansLoopInputOutput.class);
      job.setNumIterations(specIteration);
      job.setJobName("iterative k-means");
      JobClient.runJob(job);
       ...
}
```

Fig. 22 The loop setting implementation for the k-means example.

10.4 Summary

Since MapReduce has been used as a foundation to express relational algebra operators, it is straightforward to translate these SQL queries into MapReduce jobs. Essentially, PageRank, descendant query, and k-means clustering all share a recursive join structure. Our PageRank and descendant query implementations are similar to MapReduce joins in Hive [20], while k-means implementation is similar to Hive's map-side joins; the difference is that these three applications are recursive, which neither Hive nor MapReduce has built-in support.