# HaLoop: Efficient Iterative Data Processing on Large Clusters

Yingyi Bu* Bill Howe Magdalena Balazinska Michael D. Ernst
Department of Computer Science and Engineering
University of Washington, Seattle, WA, U.S.A.
yingyib@ics.uci.edu, {billhowe, magda, mernst}@cs.washington.edu

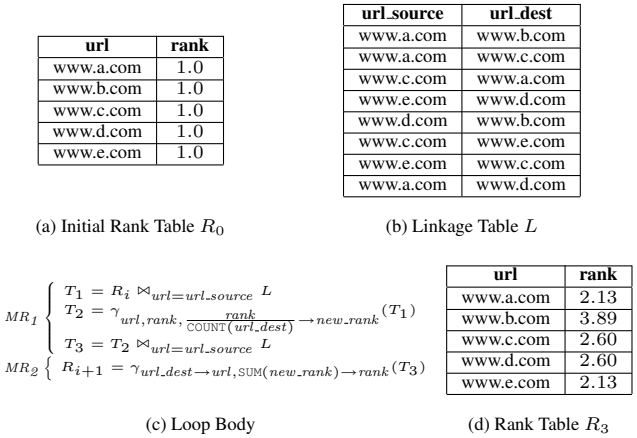
## ABSTRACT

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 and Dryad are two popular platforms in which the dataflow takes the form of a directed acyclic graph of operators. These platforms lack built-in support for iterative programs, which arise naturally in many applications including data mining, web ranking, graph analysis, model fitting, and so on. This paper presents HaLoop, a modified version of the Hadoop MapReduce framework that is designed to serve these applications. HaLoop not only extends MapReduce with programming support for iterative applications, it also dramatically improves their efficiency by making the task scheduler loop-aware and by adding various caching mechanisms. We evaluated HaLoop on real queries and real datasets. Compared with Hadoop, on average, HaLoop reduces query runtimes by 1.85, and shuffles only 4% of the data between mappers and reducers.

## 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 massive-scale simulations, sensor deployments, high-throughput lab equipment).

MapReduce [4] is a well-known 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 [5] may also serve these data analysis applications, they can be expensive, difficult to administer, and lack fault-tolerance for long-running queries [16]. Hadoop [7], 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. Many algorithms naturally fit into the MapReduce model, such as word counting, equi-join queries, and inverted list construction [4].



| url | rank |
|---|---|
| www.a.com | 1.0 |
| www.b.com | 1.0 |
| www.c.com | 1.0 |
| www.d.com | 1.0 |
| www.e.com | 1.0 |

(a) Initial Rank Table $R_0$

| url_source | url_dest |
|---|---|
| www.a.com | www.b.com |
| www.a.com | www.c.com |
| www.c.com | www.a.com |
| www.e.com | www.d.com |
| www.d.com | www.b.com |
| www.c.com | www.e.com |
| www.e.com | www.c.com |
| www.a.com | www.d.com |

(b) Linkage Table $L$

$$MR_1 \begin{cases} T_1 = R_i \bowtie_{url=url\_source} L \\ T_2 = \gamma_{url,rank,\frac{rank}{\texttt{COUNT}(url\_dest)} \rightarrow new\_rank}(T_1) \\ T_3 = T_2 \bowtie_{url=url\_source} L \end{cases}$$
$$MR_2 \left\{ R_{i+1} = \gamma_{url\_dest \rightarrow url, \texttt{SUM}(new\_rank) \rightarrow rank}(T_3) \right.$$

(c) Loop Body

| url | rank |
|---|---|
| www.a.com | 2.13 |
| www.b.com | 3.89 |
| www.c.com | 2.60 |
| www.d.com | 2.60 |
| www.e.com | 2.13 |

(d) Rank Table $R_3$

**Figure 1: PageRank example**

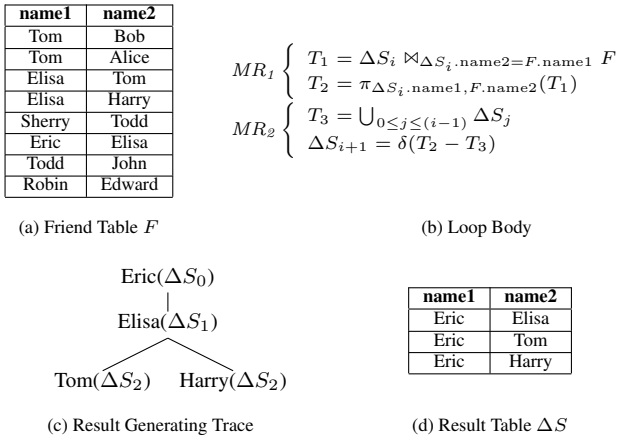
However, many data analysis techniques require *iterative* computations, including PageRank [15], HITS (Hypertext-Induced Topic Search) [11], recursive relational queries [3], clustering, neural-network analysis, social network analysis, and network traffic analysis. These techniques have a common trait: 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 programs by manually issuing multiple MapReduce jobs and orchestrating their execution using a driver program [12].

There are two key problems with manually orchestrating an iterative program in MapReduce. 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 for two consecutive iterations. 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 is a link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively computing the weight of each vertex based on the weight of its inbound neighbors. In the relational algebra, the PageRank algorithm can be expressed as a join followed by an update with*

*Work was done while the author was at University of Washington, Seattle. Current affiliation: Yingyi Bu - University of California, Irvine.


*Proceedings of the VLDB Endowment,* Vol. 3, No. 1




(a) Friend Table $F$ (b) Loop Body

(c) Result Generating Trace (d) Result Table $\Delta S$

**Figure 2: Descendant query example**

*two aggregations. These steps must be repeated by a driver program until a termination condition is satisfied (e.g., the rank of each page converges or a specified number of iterations has been performed).*

*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)) are required to implement the loop body of PageRank. The first MapReduce job joins the rank and linkage tables. Mappers emit records from the two relations with the join column as the key and the remaining columns as the value. Reducers compute the join for each unique source URL, as well as the rank contribution for each outbound edge ($new\_rank$). 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 $R_1$, $R_2$, $R_3$.*

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 on each iteration.

EXAMPLE 2. (Descendant Query) *Given the social network relation in Figure 2(a), who is within two friend-hops from Eric? To answer this query, we can first find Eric's direct friends, and then all the friends of these friends. A related query is to find all people who can be reached from Eric following the friend relation $F$. These queries can be implemented by a driver program that executes two MapReduce jobs ($MR_1$ and $MR_2$ in Figure 2(b)), either for two iterations or until fixpoint, respectively. The first MapReduce job finds a new generation of friends by joining the friend table $F$ with the friends discovered in the previous iteration, $\Delta S_i$. The second MapReduce job removes duplicate tuples from $\Delta S_i$ that also appear in $\Delta S_j$ for $j < i$. The final result is the union of results from each iteration.*

*Let $\Delta S_i$ be the result of the join after iteration $i$, computed by joining $\Delta S_{i-1}$ with $F$ and removing duplicates. $\Delta S_0$=$\{$Eric, Eric$\}$ is the trivial friend relationship that initiates the computation. Figure 2(c) shows how results evolve from $\Delta S_0$ to $\Delta S_2$. Finally, $\Delta S = \bigcup_{0<i\le 2} \Delta S_i$ is returned as the final result, as in Figure 2(d).*

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 analysis should typically continue until a fixpoint is reached. Examples include most iterative model-fitting algorithms (such as $k$-means clustering and neural network analysis), most web/graph ranking algorithms (such as HITS [11]), and recursive graph or network queries.

This paper presents a new system called *HaLoop* that is designed to efficiently handle the above types of applications. HaLoop extends MapReduce and is based on two simple intuitions. First, a MapReduce cluster can cache the invariant data in the first iteration, and then reuse them in later iterations. Second, a MapReduce cluster can cache reducer outputs, which makes checking for a fixpoint more efficient, without an extra MapReduce job.

This paper makes the following contributions:

- *New Programming Model and Architecture for Iterative Programs*: HaLoop handles loop control that would otherwise have to be manually programmed. It offers a programming interface to express iterative data analysis applications (Section 2).
- *Loop-Aware Task Scheduling*: HaLoop's task scheduler enables data reuse across iterations, by physically co-locating tasks that process the same data in different iterations (Section 3).
- *Caching for Loop-Invariant Data*: HaLoop caches and indexes data that are invariant across iterations in cluster nodes during the first iteration of an application. Caching the invariant data reduces the I/O cost for loading and shuffling them in subsequent iterations (Section 4.1 and Section 4.3).
- *Caching to Support Fixpoint Evaluation*: HaLoop caches and indexes a reducer's local output. This avoids the need for a dedicated map-reduce step for fixpoint or convergence checking (Section 4.2).
- *Experimental Study*: We evaluated our system on iterative programs that process both synthetic and real world datasets. HaLoop outperforms Hadoop in all metrics; on average, HaLoop reduces query runtimes by 1.85, and shuffles only 4% of the data between mappers and reducers (Section 5).

# 2. HALOOP OVERVIEW

This section introduces HaLoop's architecture and its application programming model.

## 2.1 Architecture

Figure 3 illustrates the architecture of HaLoop, a modified version of the open source MapReduce implementation Hadoop [7].

HaLoop inherits the basic distributed computing model and architecture of Hadoop. HaLoop relies on a distributed file system (HDFS [8]) that stores each job's input and output data. The system is divided into two parts: one master node and many slave nodes. A client submits jobs 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 (which usually performs transformations on an input data partition, and calls a user-defined map function with one ⟨key, value⟩ pair each time) or a reduce task (which usually copies the corresponding partition of mapper output, groups the input keys, and invokes a user-defined reduce function with one key and its associated values each time). For example, in Figure 3, there are three jobs running in the system: job

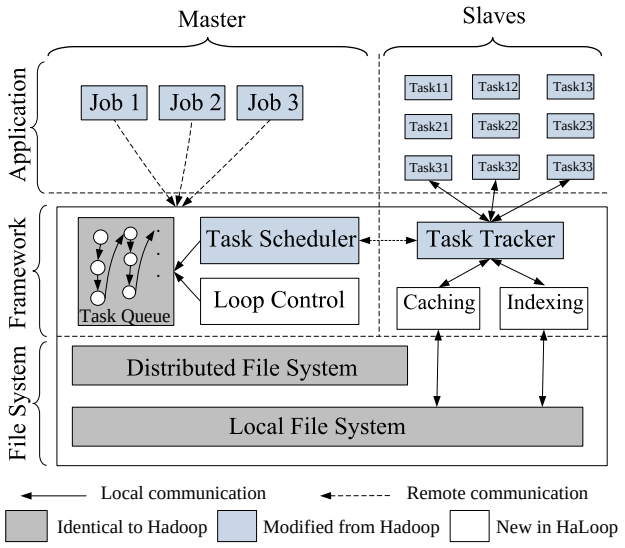



**Figure 3: The HaLoop framework, a variant of Hadoop MapReduce framework.**

1, job 2, and job 3. Each job has three tasks running concurrently on slave nodes.

In order to accommodate the requirements of iterative data analysis applications, we made several changes to the basic Hadoop MapReduce framework. First, HaLoop exposes a new application programming interface to users that simplifies the expression of iterative MapReduce programs (Section 2.2). Second, HaLoop's master node contains a new loop control module that repeatedly starts new map-reduce steps that compose the loop body, until a user-specified stopping condition is met (Section 2.2). Third, HaLoop uses a new task scheduler for iterative applications that leverages data locality in these applications (Section 3). Fourth, HaLoop caches and indexes application data on slave nodes (Section 4). 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 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 local file system.

## 2.2 Programming Model

The PageRank and descendant query examples are representative of the types of iterative programs that HaLoop supports. Here, we present the general form of the recursive programs we support and a detailed API.

The iterative programs that HaLoop supports can be distilled into the following core construct:

$$R_{i+1} = R_0 \cup (R_i \bowtie L)$$

where $R_0$ is an initial result and $L$ is an invariant relation. A program in this form terminates when a fixpoint is reached — when the result does not change from one iteration to the next, i.e. $R_{i+1} = R_i$. This formulation is sufficient to express a broad class of recursive programs.[1]

[1]SQL (ANSI SQL 2003, ISO/IEC 9075-2:2003) queries using the `WITH` clause can also express a variety of iterative applications, including complex analytics that are not typically implemented in SQL such as $k$-means and PageRank; see Section 9.5.

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 either the difference between two consecutive iterations is less than a user-specified threshold, or the maximum number of iterations has been reached. Both kinds of approximate fixpoints are useful for expressing convergence conditions in machine learning and complex analytics. For example, for PageRank, it is common to either use a user-specified convergence threshold $\epsilon$ [15] or a fixed number of iterations as the loop termination condition.

Although our recursive formulation describes the class of iterative programs we intend to support, this work does not develop a high-level declarative language for expressing recursive queries. Rather, we focus on providing an efficient foundation API for iterative MapReduce programs; we posit that a variety of high-level languages (e.g., Datalog) could be implemented on this foundation.

To write a HaLoop program, a programmer specifies the loop body (as one or more map-reduce pairs) and optionally specifies a termination condition and loop-invariant data. We now discuss HaLoop's API (see Figure 16 in the appendix for a summary). `Map` and `Reduce` are similar to standard MapReduce and are required; the rest of the API is new and is optional.

To specify the loop body, the programmer constructs a multi-step MapReduce job, using the following functions:

- `Map` transforms an input ⟨key, value⟩ tuple into intermediate ⟨in_key, in_value⟩ tuples.
- `Reduce` processes intermediate tuples sharing the same in_key, to produce ⟨out_key, out_value⟩ tuples. The interface contains a new parameter for cached invariant values associated with the in_key.
- `AddMap` and `AddReduce` express a loop body that consists of more than one MapReduce step. `AddMap` (`AddReduce`) associates a `Map` (`Reduce`) function with an integer indicating the order of the step.

HaLoop defaults to testing for equality from one iteration to the next to determine when to terminate the computation. To specify an approximate fixpoint termination condition, the programmer uses the following functions.

- `SetFixedPointThreshold` sets a bound on the distance between one iteration and the next. If the threshold is exceeded, then the approximate fixpoint has not yet been reached, and the computation continues.
- The `ResultDistance` function calculates the distance between two out_value sets sharing the same out_key. One out_value set $v_i$ is from the reducer output of the current iteration, and the other out_value set $v_{i-1}$ is from the previous iteration's reducer output. The distance between the reducer outputs of the current iteration $i$ and the last iteration $i-1$ is the sum of `ResultDistance` on every key. (It is straightforward to support additional aggregations besides `sum`.)
- `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. `SetMaxNumOfIterations` can also be used to implement a simple `for`-loop.

To specify and control inputs, the programmer uses:

- `SetIterationInput` associates an input source with a specific iteration, since the input files to different iterations may be different. For example, in Example 1, at each iteration $i+1$, the input is $R_i \cup L$.

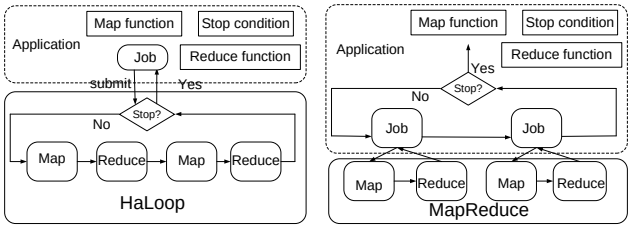



**Figure 4: Boundary between an iterative application and the framework (HaLoop vs. Hadoop). HaLoop knows and controls the loop, while Hadoop only knows jobs with one map-reduce pair.**

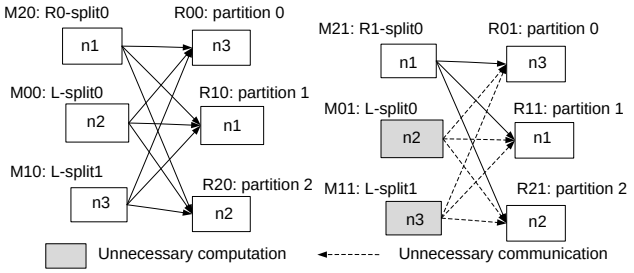



**Figure 5: A schedule exhibiting inter-iteration locality. Tasks processing the same inputs on consecutive iterations are scheduled to the same physical nodes.**

- `AddStepInput` associates an additional input source with an intermediate map-reduce pair in the loop body. The output of preceding map-reduce pair is always in the input of the next map-reduce pair.
- `AddInvariantTable` specifies an input table (an HDFS file) that is loop-invariant. During job execution, HaLoop will cache this table on cluster nodes.

This programming interface is sufficient to express a variety of iterative applications. The appendix sketches the implementation of PageRank (Section 9.2), descendant query (Section 9.3), and $k$-means (Section 9.4) using this programming interface. 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. Sections 3.1 and 3.2 illustrate the desired schedules and scheduling algorithm respectively.

## 3.1 Inter-Iteration Locality

The high-level goal of HaLoop's scheduler is to place *on the same physical machines* those map and reduce tasks that occur in different iterations but access the same data. With this approach, data can more easily be cached and re-used between iterations. For example, Figure 5 is a sample schedule for the join step ($MR_1$ in Figure 1(c)) of the PageRank application from Example 1. There are two iterations and three slave nodes involved in the job.

The scheduling of iteration 1 is no different than in Hadoop. In the join step of the first iteration, 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 $R_{10}$ processes keys with hash value 1, and reducer $R_{20}$ processes keys with hash value 2.

The scheduling of the join step of iteration 2 can take advantage of inter-iteration locality: the task (either mapper or reducer) that processes a specific data partition $D$ is scheduled on the physical node where $D$ was processed in iteration 1. Note that the two file inputs to the join step in iteration 2 are $L$ and $R_1$.

The schedule in Figure 5 provides the feasibility to reuse loop-invariant data from past iterations. Because $L$ is loop-invariant, mappers $M_{01}$ and $M_{11}$ would compute identical results to $M_{00}$ and $M_{10}$. There is no need to re-compute these mapper outputs, nor to communicate them to the reducers. In iteration 1, if reducer input partitions 0, 1, and 2 are stored on nodes $n_3$, $n_1$, and $n_2$ respectively, then in iteration 2, $L$ need not be loaded, processed or shuffled again. In that case, in iteration 2, only one mapper $M_{21}$ for $R_1$-split0 needs to be launched, and thus the three reducers will only copy intermediate data from $M_{21}$. 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.

We refer to the property of the schedule in Figure 5 as *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 $i > 1$, $T_d^i$ and $T_d^{i-1}$ 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, so that the hash function assigning mapper outputs to reducer nodes remains unchanged.

## 3.2 Scheduling Algorithm

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.

More specifically, the HaLoop scheduler works as follows. Upon receiving a heartbeat from a slave node, the master node tries to assign the slave node an unassigned task that uses data cached on that node. To support this assignment, the master node maintains a mapping from each slave node to the data partitions that this node processed in the previous iteration. If the slave node already has a full load, the master re-assigns its tasks to a nearby slave node.

Figure 6 gives pseudocode for the scheduling algorithm. Before each iteration, `previous` is set to `current`, and then `current` is set to a new empty `HashMap` object. In a job's first iteration, the schedule is exactly the same as that produced by Hadoop (line 2). After scheduling, the master remembers the association between data and node (lines 3 and 13). In later iterations, the scheduler tries to retain previous data-node associations (lines 11 and 12). If the associations can no longer hold due to the load, the master node will associate the data with another node (lines 6–8).

# 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

[2]Mapper input partitions 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.

***Task Scheduling***
Input: Node node
*// The current iteration's schedule; initially empty*
Global variable: Map⟨Node, List⟨Partition⟩⟩ current
*// The previous iteration's schedule*
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
5:      if node.hasFullLoad() then
6:          Node substitution = findNearestIdleNode(node);
7:          previous.get(substitution).addAll(previous.remove(node));
8:          return;
9:      end if
10:     if previous.get(node).size()>0 then
11:         Partition part = previous.get(node).get(0);
12:         schedule(part, node);
13:         current.get(node).add(part);
14:         previous.remove(part);
15:     end if
16: end if
```

**Figure 6: Task scheduling algorithm. If there are running jobs, this function is called when master node receives a heartbeat from a slave.**

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. If a cache becomes unavailable, it is automatically re-loaded, either from map task physical nodes, or from HDFS. HaLoop maintains three types of caches: reducer input cache, reducer output cache, and mapper input cache. Each of them fits a number of application scenarios. Application programmers can choose to enable or disable a cache type using the HaLoop API (see Appendix 9.1).

## 4.1 Reducer Input Cache

If an intermediate table is specified to be loop-invariant (via the HaLoop API `AddInvariantTable`) and the reducer input cache is enabled, HaLoop will cache reducer inputs across all reducers and create a local index for the cached data. Note that 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.

Let us consider the social network example (Example 2) to see how the reducer input cache works. Three physical nodes $n_1$, $n_2$, and $n_3$ 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 $\Delta S_0$-split0. The three splits are shown in Figure 7. The two reducer input partitions are shown in Figure 8. The reducer on $n_1$ corresponds to hash value 0, while the reducer on $n_2$ corresponds to hash value 1. Then, since table $F$ (with table ID "#1") is set to be invariant by the programmer using the `AddInvariantTable` function, every reducer will cache the tuples with table ID "#1" in its local file system.

In later iterations, when a reducer 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 to find associated values and passes them together to the `Reduce` function (note that HaLoop's modified `Reduce` interface accepts this parameter; see details in Appendix 9.1). Also, if the reducer input cache is enabled, mapper outputs in the first iteration are cached in the corresponding mapper's local disk, for future reducer cache reloading.

In the physical layout of the cache, keys and values are separated

| 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) $\Delta S_0$-split0

**Figure 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

**Figure 8: Reducer Input Partitions in Example 2**

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. Thus, after reducer input data are cached to local disk, HaLoop creates an index 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.

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 that: (1) $f$ must be deterministic, (2) $f$ must remain the same across iterations, and (3) $f$ must not take any inputs other than the tuple $t$. In HaLoop, the number of reduce tasks is unchanged across iterations, therefore the default hash partitioning satisfies these conditions.

## 4.2 Reducer Output Cache

The reducer output cache stores and indexes the most recent local output on each reducer node. This cache is used to reduce the cost of evaluating fixpoint termination conditions. That is, if the application must test the convergence condition by comparing the current iteration output with the previous iteration output, the reducer output cache enables the framework to perform the comparison in a distributed fashion.

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. 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 last map-reduce pair of the loop body, the mapper output partition function $f$ and the `reduce` function satisfy the following conditions: 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 so that they are sent to the same reduce task. Further, $f$ should also meet the requirements of the reducer input cache. Satisfying these requirements guarantees that 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.

## 4.3 Mapper Input Cache

Hadoop [7] attempts to co-locate map tasks with their input data. On a real-world Hadoop cluster [1], the rate of data-local mappers is around 70%–95%, 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.

## 4.4 Cache Reloading

There are a few cases where the cache must be re-constructed: (1) the hosting node fails, or (2) the hosting node has a full load and a map or reduce task must be scheduled on a different substitution node. A reducer reconstructs the reducer input cache by copying the desired partition from all first-iteration mapper outputs. To reload the mapper input cache or the reducer output cache, the mapper/reducer only needs to read the corresponding chunks from the distributed file system, where replicas of the cached data are stored. Cache re-loading is completely transparent to user programs.

# 5. 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 5.1–5.3.

## 5.1 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).

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). Detailed hardware and dataset descriptions are in Section 9.6.

We executed the PageRank query on the Livejournal and Freebase datasets and the descendant query on the Livejournal and Triples datasets. Figures 9–12 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.

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 `fixedPointThreshold` and `ResultDistance`, to specify the loop termination condition. The results are plotted in Figure 9(a), Figure 10(a), Figure 11(a), and Figure 12(a).

In the PageRank algorithm, there are two steps in every iteration: join and aggregation. The running time in Figure 9(a) and Figure 10(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 11(a) and Figure 12(a) is the sum of join time and "duplicate elimination" time over all iterations.

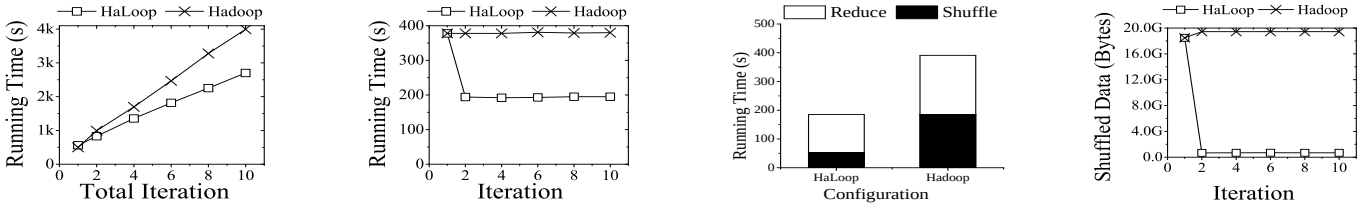
HaLoop always performs better than Hadoop. The descendant query on the Triples dataset has the best improvement, PageRank on Livejournal and Freebase have intermediate gains, but the descendant query on the Livejournal dataset has the least improvement. Livejournal is a social network dataset with high 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 9(b), Figure 10(b), Figure 11(b), and Figure 12(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.

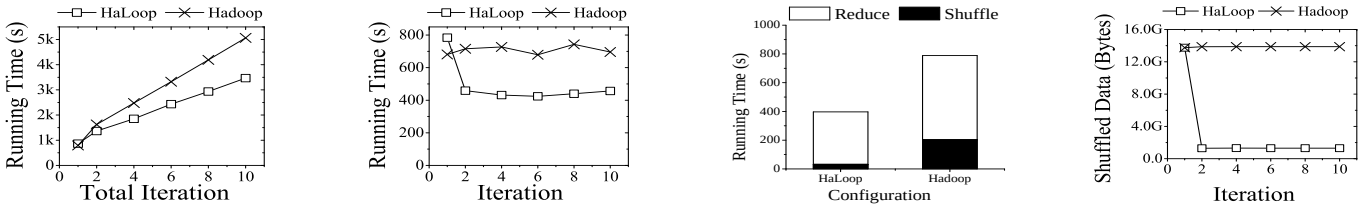
**Cost Distribution for Join Step**. To better understand HaLoop's improvements to each phase, we compared the cost distribution of the join step across Map and Reduce phases. Figure 9(c), Figure 10(c), Figure 11(c), and Figure 12(c) show the cost distribution of the join step in a certain iteration (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 11(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 $\Delta S_i$ and $F$ will use an index seek to search qualified tuples in the cache of $F$. Also, in



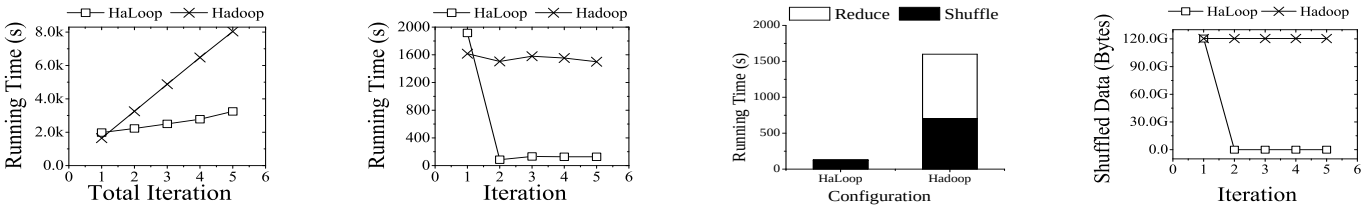

(a) Overall Performance (b) Join Step (c) Cost Distribution (d) Shuffled Bytes

**Figure 9: PageRank Performance: HaLoop vs. Hadoop (Livejournal Dataset, 50 nodes)**



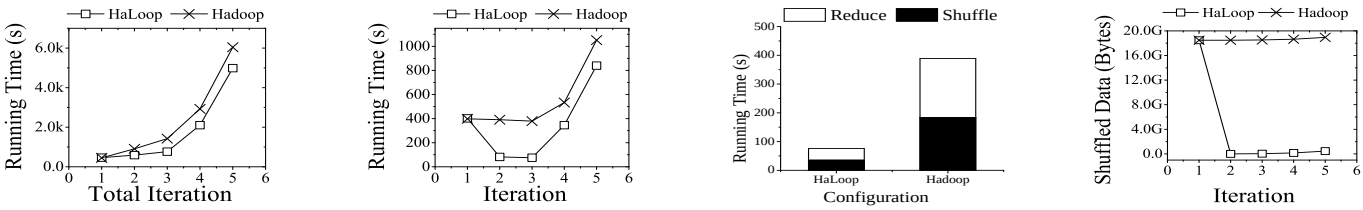

(a) Overall Performance (b) Join Step (c) Cost Distribution (d) Shuffled Bytes

**Figure 10: PageRank Performance: HaLoop vs. Hadoop (Freebase Dataset, 90 nodes)**




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

**Figure 11: Descendant Query Performance: HaLoop vs. Hadoop (Triples Dataset, 90 nodes)**




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

**Figure 12: Descendant Query Performance: HaLoop vs. Hadoop (Livejournal Dataset, 50 nodes)**

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 12), in iteration$>$3, the index does not help either, because the selectivity becomes high.

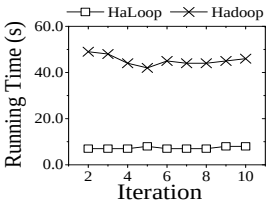
**I/O in Shuffle Phase of Join Step**. To tell how much shuffling I/O is saved, we compared the amount of shuffled data in the join step of each iteration. Since HaLoop caches loop-invariant data, the overhead of shuffling these invariant data are completely avoided. These savings contribute an important part of the overall performance improvement. Figure 9(d), Figure 10(d), Figure 11(d), and Figure 12(d) plot the sizes of shuffled data. On average, HaLoop's join step shuffles 4% as much data as Hadoop's does.

## 5.2 Evaluation of Reducer Output Cache

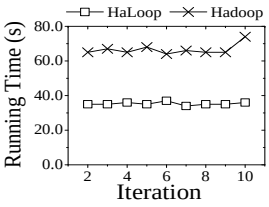
This experiment shares the same hardware and dataset as the reducer input cache experiments. To see how effective HaLoop's reducer output cache is, we compared the cost of fixpoint evaluation in each iteration. Since descendant query has a trivial fixpoint evaluation step that only requires testing to see if a file is empty, we run the PageRank implementation in Section 9.2 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 13(a) and (b) shows the time spent on fixpoint evaluation in each iteration.

## 5.3 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. 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). Detailed hardware and dataset descriptions are in Section 9.6. We vary the number of total iterations, and plot the algorithm running time in Figure 14. The mapper locality rate is around 95% since there are not concurrent jobs in our

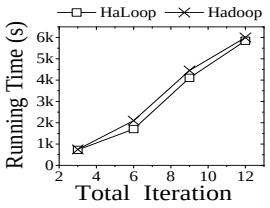



(a) Livejournal, 50 nodes

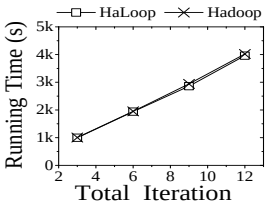



(b) Freebase, 90 nodes

**Figure 13: Fixpoint Evaluation Overhead in PageRank: HaLoop vs. Hadoop**




(a) Cosmo-dark, 8 nodes




(b) Cosmo-gas, 8 nodes

**Figure 14: Performance of $k$-means: HaLoop vs. Hadoop**

lab HaLoop cluster. By avoiding non-local data loading, HaLoop performs marginally better than Hadoop.

# 6. RELATED WORK

Parallel database systems [5] 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 [3, 17]. 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 [4] has emerged as a popular alternative for massive-scale parallel data analysis in shared-nothing clusters. Hadoop [7] is an open-source implementation of MapReduce. MapReduce has been followed by a series of related systems including Dryad [10], Hive [9], Pig [14], and HadoopDB [2]. Like Hadoop, none of these systems provides explicit support and optimizations for iterative or recursive types of analysis.

Mahout [12] 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.

Twister [6] is a stream-based MapReduce framework that supports iterative programs, in which mappers and reducers are long running with distributed memory caches. They are established to avoid repeated mapper data loading from disks. However, Twister's streaming architecture between mappers and reducers is sensitive to failures, and long-running mappers/reducers plus memory cache is not a scalable solution for commodity machine clusters, where each node has limited memory and resources.

Finally, Pregel [13] is a distributed system for processing large-size graph datasets, but it does not support general iterative programs.

# 7. 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. 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 grant CNS-0855252, 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.

# 8. REFERENCES

[1] `http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm`. Accessed July 7, 2010.

[2] 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.

[3] François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD Conference*, pages 16–52, 1986.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

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

[6] Jaliya Ekanayake and Shrideep Pallickara. MapReduce for data intensive scientific analysis. In *IEEE eScience*, pages 277–284, 2008.

[7] Hadoop. `http://hadoop.apache.org/`. Accessed July 7, 2010.

[8] Hdfs. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`. Accessed July 7, 2010.

[9] Hive. `http://hadoop.apache.org/hive/`. Accessed July 7, 2010.

[10] 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.

[11] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[12] Mahout. `http://lucene.apache.org/mahout/`. Accessed July 7, 2010.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

# 9. APPENDIX

This appendix presents additional implementation details for the HaLoop system and our sample applications, experiment setup details, and a discussion.

## 9.1 HaLoop Implementation Details

We first provide some additional details about HaLoop's extensions of Hadoop.

### *9.1.1 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 [12] project has implemented multiple iterative machine learning and data mining algorithms with this approach.

Figure 15 demonstrates how an iterative program is executed in Hadoop. It also shows how the following classes fit together in the Hadoop system.

**Hadoop master node**. In Hadoop, interface `TaskScheduler` and class `JobInProgress` play the role of master node: they accept heartbeats from slave nodes and manage task scheduling.

**Hadoop slave nodes**. 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.

**User-defined map and reduce functions**. 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` thread.

### *9.1.2 HaLoop Extensions to Hadoop*

We extended and modified Hadoop as follows:

**Hadoop master node: loop control and new 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, and inform HaLoop about any loop-invariant data. `JobConf` class represents a client job and hosts these APIs. Figure 16 shows the descriptions of this API.

**Hadoop slave nodes: 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 iteration number. Therefore, tasks accessing the cache in the future could know the data is generated from which iteration. After the iterative job finishes, the whole cache related to the job will be erased.

**User-defined map and reduce functions: iterations**. We added abstract classes `MapperIterative` and `ReducerIterative` to wrap the `Mapper`/`Reducer` interfaces in Hadoop. They both provide an empty implementation for the user-defined `map`/`reduce` functions and add new `map`/`reduce` functions to accept both parameters for ordinary `map`/`reduce` functions and iteration-related parameters such as current iteration number. ReduceIterative's new `reduce` function also adds another new parameter, which stores the cached reducer input values associated with the key.

**User-defined map and reduce functions: fixpoint evaluation**. HaLoop evaluates the fixpoint in a distributed fashion.

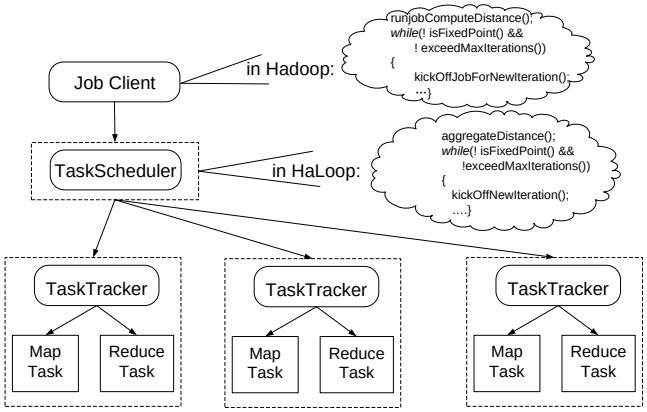



**Figure 15: Job Execution: HaLoop V.s. Hadoop**

| Name | Functionality |
|---|---|
| AddMap & AddReduce | specify a step in loop |
| SetDistanceMeasure | specify a distance for results |
| SetInput | specify inputs to iterations |
| AddInvariantTable | specify loop-invariant data |
| SetFixedPointThreshold | a loop termination condition |
| SetMaxNumOfIterations | specify the max iterations |
| SetReducerInputCache | enable/disable reducer input caches |
| SetReducerOutputCache | enable/disable reducer output caches |
| SetMapperInputCache | enable/disable mapper input caches |

**Figure 16: HaLoop API**

After the final reduce phase of an iteration, `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 `fixedPointThreshold`. If the distance is less than `fixedPointThreshold` or current iteration number is already `maxNumOfIterations`, `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 15 also shows how HaLoop executes a job. In particular, we see that the `TaskScheduler` manages the lifecycle of an iterative job execution.

## 9.2 PageRank Implementation

Let us walk through how PageRank (from Example 1) is implemented on top of HaLoop. Figure 17 shows the pseudo-code of this implementation. There are two steps in PageRank's loop body: one is to join $R_i$ and $L$ and populate ranks; the other is to aggregate ranks on each URL. Each step is a map-reduce pair. Each pair is added to the overall iterative program by calling HaLoop's `AddMap` and `AddReduce` functions (line 2-5 in `Main`).

The join step is comprised of two user-defined functions, `Map_Rank` and `Reduce_Rank`. In the first iteration, `Map_Rank` reads an input tuple, either from the linkage table $L$ or the initial rank table $R_0$. It outputs the join column as key ($L$.url_src or $R_0$.url) and the rest of the input tuple as the value. It also attaches a table ID to each output tuple to distinguish their sources. In Figure 17, #1 is the table ID for $L$, while #2 is the table ID for rank table $R_i$. In later iterations, `Map_Rank` simply reads tuples from $R_i$, outputs column `url` as the key and column `rank` as the value, and attaches the table ID as before.

On each iteration, the `Reduce_Rank` calculates the local rank for destination URLs (in `invariantValues`), where each destination

***Map_Rank***
Input: Key k, Value v, int iteration
1: **if** v from $L$ **then**
2: Output(v.url_src, v.url_dest, #1);
3: **else**
4: Output(v.url, v.rank, #2);
5: **end if**

***Reduce_Rank***
Input: Key key, Set values, Set invariantValues,
int iteration
1: **for** url_dest in invariantValues **do**
2: Output(url_dest, values.get(0)/invariantValues.size());
3: **end for**

***Map_Aggregate***
Input: Key k, Value v, int iteration
1: Output(v.url, v.rank);

***Reduce_Aggregate***
Input: Key key, Set values, int iteration
1: Output(key, AggregateRank(values));

***ResultDistance***
Input: Key out_key, Set $v_{i-1}$, Set $v_i$
1: return $|v_i.get(0) - v_{i-1}.get(0)|$;

***IterationInput***
Input: int iteration
1: **if** iteration==1 **then**
2: return $L \cup R_0$;
3: **else**
4: return $R_{\text{iteration}-1}$
5: **end if**

***Main***
1: Job job = new Job();
2: job.AddMap(Map_Rank, 1);
3: job.AddReduce(Reduce_Rank, 1);
4: job.AddMap(Map_Aggregate, 2);
5: job.AddReduce(Reduce_Aggregate, 2);
6: job.SetDistanceMeasure(ResultDistance);
7: job.AddInvariantTable(#1);
8: job.SetInput(IterationInput);
9: job.SetFixedPointThreshold(0.1);
10: job.SetMaxNumOfIterations(10);
11: job.SetReducerInputCache(true);
12: job.SetReducerOutputCache(true);
13: job.Submit();

**Figure 17: Implementation of Example 1 on HaLoop**

URL's rank is assigned to the source URL's rank divided by the number of destination URLs.

The aggregation step includes `Map_Aggregate` and `Reduce_Aggregate`, where `Map_Aggregate` reads raw ranks produced by `Reduce_Rank`, and `Reduce_Aggregate` sums the local ranks for each URL.

The distance measure between reducer outputs from consecutive iterations is simply the rank difference (`ResultDistance` and line 6 in `Main`). Table $L$ is set as loop-invariant (line 1-2 in `Map_Rank` and line 7 in `Main`). `IterationInput` and line 8 in `Main` specify the input to each iteration: $\{L, R_0\}$ for the first iteration and $\{R_{i-1}\}$ for later iteration $i$. Therefore, in `Reduce_Rank`, `invariantValues` are obtained by querying `key`(in the input to `Reduce_Rank`) from the cached $L$ partition and projecting on the `url_dest` column. The `fixedPointThreshold` is set to 0.1, while the `maxNumOfIterations` is set to 10 (line 9-10 in `Main`). Lines 11-12 in `Main` enable the reducer input cache to improve the performance of the join step and enable the reducer output cache to support distributed fixpoint evaluation. Finally, the job is submitted to the HaLoop master node (line 13 in `Main`).

***Map_Join***
Input: Key k, Value v, int iteration
1: **if** v from $F$ **then**
2: Output(v.name1, v.name2, #1);
3: **else**
4: Output(v.name2, v.name1, #2);
5: **end if**

***Reduce_Join***
Input: Key key, Set values, Set invariantValues,
int iteration
1: Output(Product(values, invariantValues));

***Map_Distinct***
Input: Key k, Value v, int iteration
1: Output(v.name1, v.name2, iteration);

***Reduce_Distinct***
Input: Key key, Set values, int iteration
1: **for** name in values **do**
2: **if** (name.iteration $<$ iteration) **then**
3: set_old.add(name);
4: **else** set_new.add(name);
5: **end for**
6: Output(Product(key, Distinct(set_new-set_old)));

***IterationInput***
Input: int iteration
1: **if** iteration==1 **then**
2: return $F \cup \Delta S_0$;
3: **else**
4: return $\Delta S_{\text{iteration}-1}$
5: **end if**

***StepInput***
Input: int step, int iteration
1: **if** step==2 **then**
2: return $\bigcup_{0 \le j \le (\text{iteration}-1)} \Delta S_j$
3: **end if**

***ResultDistance***
Input: Key out_key, Set $v_{i-1}$, Set $v_i$
1: return $v_i$.size();

***Main***
1: Job job = new Job();
2: job.AddMap(Map_Join, 1);
3: job.AddReduce(Reduce_Join, 1);
4: job.AddMap(Map_Distinct, 2);
5: job.AddReduce(Reduce_Distinct, 2);
6: job.SetDistanceMeasure(ResultDistance);
7: job.SetInput(IterationInput);
8: job.AddInvariantTable(#1);
9: job.SetFixedPointThreshold(1);
10: job.SetMaxNumOfIterations(2);
11: job.SetReducerInputCache(true);
12: job.AddStepInput(StepInput);
13: job.Submit();

**Figure 18: Implementation of Example 2 on HaLoop**

## 9.3 Descendant Query Implementation

We present the pseudo-code for the HaLoop implementation of Example 2 (descendant query) in Figure 18. Similar to PageRank example, the loop body also has two steps: one is join (to find friends-of-friends by looking one hop further), and the other one is

***Map_Kmeans_Configure***
1: loadLatestCluster();

***Map_Kmeans***
Input: Key k, Value v, int iteration
1: Output(assignCluster(v), v);

***Reduce_Kmeans***
Input: Key key, Set values, Set invariantValues, int iteration
1: Output(key, AVG(values));

***IterationInput***
Input: int iteration
1: return "input";

***ResultDistance***
Input: Key out_key, Set $v_{i-1}$, Set $v_i$
1: return Manhattan_Distance($v_i$.get(0), $v_{i-1}$.get(0));

***Main***
1: Job job = new Job();
2: job.AddMap(Map_Kmeans, 1);
3: job.AddReduce(Reduce_Kmeans, 1);
4: job.SetDistanceMeasure(ResultDistance);
5: job.SetFixedPointThreshold(0.01);
6: job.SetMaxNumOfIterations(12);
7: job.SetInput(IterationInput);
8: job.SetMapperInputCache(true);
9: job.Submit();

**Figure 19: $K$-means Implementation on HaLoop**

duplicate elimination (to remove duplicates in the extended friends set). We still utilize reducer input cache (line 11 in `Main`), and set $F$ to be loop invariant (line 1-2 in `Map_Join` and line 8 in `Main`). `Map_Join` and `Reduce_Join` form the join step. In the first iteration, `Map_Join` reads input tuples from both $F$ and $\Delta S_0$, and outputs the join column as key and the remaining columns and the table ID as value. In this example, #1 is the ID of the friend table $F$ and #2 is the ID of $\Delta S_{i-1}$. In later iteration $i$, `Map_Join` simply reads $\Delta S_{i-1}$ tuples and attaches the table ID to them as output. For each `key` ($\Delta S_{i-1}$.name2), `Reduce_Join` computes the cartesian product of the corresponding `values` ($\Delta S_{i-1}$.name1) and `invariantValues` ($F$.name2). The duplicate elimination step includes `Map_Distinct` and `Reduce_Distinct`. `Map_Distinct` emits tuples with column `name1` as key and column `name2` as value, while `Reduce_Distinct` outputs distinct $\langle$key, value$\rangle$ ($\langle\Delta S_i$.name1, $\Delta S_i$.name2$\rangle$) pairs. The binding to `IterationInput` at line 7 in `Main` specifies the input to each iteration: $\{F, \Delta S_0\}$ for the first iteration and $\{\Delta S_{i-1}\}$ for later iteration $i$. The `ResultDistance` function simply returns current `out_key`'s corresponding `out_value` set $v_i$'s size. The `fixedPointThreshold` is set to 1 at line 9 in `Main`. The `maxNumOfIteration` is set to 2. Thus, the loop termination condition is that either $\Delta S_i$ is empty or two iterations pass. Since the fixpoint evaluation does not compare results from two iterations, we disable reducer output cache option. Other parts in the `Main` function are similar to the corresponding parts in Figure 17.

## 9.4 $K$-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 cache, because the input data to mappers at each iteration are invariant, while the reducer input data keep changing. Also, since the output from each iteration has a very small size, there is no need to enable reducer output cache.

We sketch the code for this application in Figure 19. There is only one map-reduce step in the program: `Map_Kmeans` and `Reduce_Kmeans`. `Map_Kmeans` assigns an input tuple to the nearest cluster (based on the distances between the tuple and every cluster's mean), outputs the cluster ID as the key, and the tuple as value, while `Reduce_Kmeans` calculates the means of all tuples in one cluster. We only output cluster means as the result of each iteration. There is one extra MapReduce job to finally determine and output every tuple's cluster membership after the loop is completed. For simplicity, we omit this extra job here. `IterationInput` returns a constant (the HDFS path to the dataset), such that each iteration `Map_Kmeans` reads the same input files. Each mapper also loads the latest cluster means from HDFS in mapper hook function `Map_Kmeans_Configure` before the mapper function `Map_Kmeans` is called. The `ResultDistance` measures the dissimilarity between two clusters produced from different iterations but with the same cluster ID. The distance measure is the Manhattan distance[3] between two cluster means. The `fixedPointThreshold` is set to 0.01 at line 5 in `Main`, while the `maxNumOfIteration` is set to 12 at the next line. At line 8 of `Main`, the mapper input cache is enabled.

## 9.5 Higher-Level Query Language

We observe that the general form of the recursive queries we support has a basic structure similar to recursive queries as defined in the SQL standard.

Recall that our recursive programs have the form:

$$R_{i+1} = R_0 \cup (R_i \bowtie L)$$

**Descendant Query in SQL using WITH**. To illustrate how this formulation relates to a recursive query expressed in SQL using the `WITH` syntax, consider a simple descendant query as an example:

```
WITH descendants (parent, child) AS (
 -- R0: base case
 SELECT parent, child FROM parentof
 WHERE parent = 'Eric'
    UNION ALL
 -- R ⋈ L: step case
 SELECT d.parent, e.child
 FROM descendants d, parentof e
 WHERE d.child = e.parent
)
-- Ri+1 = R0 ∪ (Ri ⋈ L)
SELECT DISTINCT * FROM descendants
```

This query computes the transitive closure of the `parentof` table by repeatedly joining an initial result set (records with parent = 'Eric') with an invariant relation (the entire `parentof` relation), and (optionally) appending the results. The last line removes duplicates and returns all results.

We find this formulation to be very general; SQL queries using the `WITH` clause are sufficient to express a variety of iterative applications, including complex analytics that are not typically implemented in SQL.

**$K$-means in SQL using WITH**. We now show how to express $k$-means clustering as a recursive query. Assume there are two relations $points(pid, point)$, $means(kid, center)$. The $points$ relation holds data values for which we wish to compute the $k$ clusters. The $means$ relation holds an initial estimate of the means, usually randomized.

[3] http://en.wikipedia.org/wiki/Manhattan_distance

| 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 |

**Figure 20: Dataset Descriptions**

```
-- find minimum dist for each point
CREATE VIEW dmin SELECT pid,
min(dist(pp.point, kk.mean)) AS dist,center
FROM points pp, means kk
GROUP BY pid

-- find mean for each pid
CREATE VIEW assign_cluster
SELECT pid, point, kid
  FROM points p, means k, dmin d
WHERE dist(p.point, k.mean) = d.dist

-- update step
CREATE VIEW newmeans AS
SELECT kid, avg(point)
  FROM assign_cluster
GROUP BY kid

-- put it all together
WITH means AS (
SELECT kid, mean, 0 FROM initial_means
UNION ALL
SELECT kid, avg(point), level + 1
  FROM points p, means k
 WHERE dist(p.point, k.center) =
     (select min(dist(p.point, m.center))
       FROM means m)
   AND k.level = (select max(level) FROM means)
   AND dist(k.center, d.center) < $threshold
GROUP BY kid
);
SELECT * FROM means
```

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 map-reduce joins in Hive [9], 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. Further, with a modest extension to high-level languages such as Hive, common table expressions could be supported directly and optimized using HaLoop, and then programmers' implementation effort could be greatly reduced.

## 9.6 Hardware and Dataset Descriptions

This section presents additional details about our experimental design, for both reducer (input/output) cache evaluation and mapper input cache evaluation.

### *9.6.1 Settings for Reducer Cache Evaluations*

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.

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 as an edge. The filtered Triples dataset is 120GB in size.

Freebase is another real-world dataset[7], where a large amount of 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 of ⟨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 data set statistics are in Figure 20.

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.

### *9.6.2 Settings for Mapper Input Cache Evaluations*

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

## 9.7 Discussion

Here we compare some other design alternatives with HaLoop.

- *Disk Cache vs. Memory Cache*. To cache loop-invariant data, one can use either disk or memory. HaLoop only caches data to disk. The reason is that in a commodity machine cluster, a slave node does not have sufficient memory to hold the cache, especially when there are a large number of tasks that have to run on the node.
- *Synchronized Iteration vs. Asynchronous Iteration*. HaLoop only utilizes partitioned parallelism. There could be some dataflow parallelism if iterations are not strictly synchronized. However, dataflow parallelism is not the goal of MapReduce, and it is also out of this work's scope.
- *Loop Body: Single Pipeline vs. DAGs*. Currently, HaLoop only supports articulated map-reduce pairs with a single pipeline in the loop body, rather than DAGs. Although DAGs are a more general form of loop body, we believe the current design can meet the requirements of many iterative data analysis applications.

[4]http://aws.amazon.com/ec2/instance-types/
[5]http://snap.stanford.edu/data/index.html
[6]http://challenge.semanticweb.org/
[7]http://www.freebase.com/
[8]http://nuage.cs.washington.edu/benchmark/astro-nbody/dataset.php