HaLoop: Efficient Iterative Data Processing on Large Clusters

Yingyi Bu, Bill Howe, Magdalena Balazinska, Michael D.Ernest
Department of Computer Science and Engineering
University of Washington, Seattle, WA, U.S.A

발표자: 김현하, 김주희

CONTENTS

Abstract

- New highly scalable data-intensive computing platform needed

  -> Growing Demand for large-scale data mining and data analysis apps

  ex) MapReduce

- But they lack built-in support for iterative programs

- **HaLoop** : Modified version of Hadoop MapReduce

  (1) Iterative apps (2) Efficiency – Loop aware

Introduction

- MapReduce: a framework to perform large-scale data processing in a single pass
- Hadoop : an open-source MapReduce implementation

-> MapReduce does not support iterative programs.

-> Many data analysis techniques require " Iterative Computations"

ex) *PageRank*

Introduction

- 2 Problems of not supporting iteration:

(1) The data must be re-loaded and re-processed at each iteration

-> even though data is unchanged

(2) The termination condition may involve detecting when a *fixpoint* has been reached.

(when output doesn't change for 2 consecutive iteration)

➡ 2 Examples

Introduction

- Example 1 – PageRank

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

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

$$MR_1 \begin{cases} T_1 = R_i \bowtie_{url=url\_source} L \\ T_2 = \gamma_{url,rank,\frac{rank}{\text{COUNT}(url\_dest)} \rightarrow new\_rank}(T_1) \\ T_3 = T_2 \bowtie_{url=url\_source} L \end{cases}$$

$$MR_2 \left\{ R_{i+1} = \gamma_{url\_dest \rightarrow url, \text{SUM}(new\_rank) \rightarrow rank}(T_3) \right.$$

(c) Loop Body

L is invariant.

However, L is processed and

Shuffled at each iteration

(MapReduce framework)

-> extra work

Introduction

- Example 2 – Descendant Query

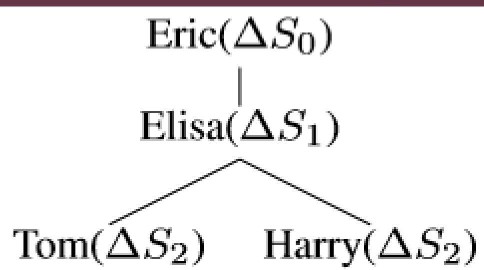
name1	name2
Tom	Bob
Tom	Alice
Elisa	Tom
Elisa	Harry
Sherry	Todd
Eric	Elisa
Todd	John
Robin	Edward

(a) Friend Table F

$$MR_1 \begin{cases} T_1 = \Delta S_i \bowtie_{\Delta S_i.\mathrm{name2}=F.\mathrm{name1}} F \\ T_2 = \pi_{\Delta S_i.\mathrm{name1},F.\mathrm{name2}}(T_1) \end{cases}$$

$$MR_2 \begin{cases} T_3 = \bigcup_{0 \le j \le (i-1)} \Delta S_j \\ \Delta S_{i+1} = \delta(T_2 - T_3) \end{cases}$$

(b) Loop Body



(c) Result Generating Trace

name1	name2
Eric	Elisa
Eric	Tom
Eric	Harry

(d) Result Table ΔS

F is also constant.

But it still gets processed and shuffled at each Iteration.

Introduction

- L(Linkage Table), F(Friend Table) =

  Significant fractions of the processed data

  -> remains invariant across iterations

- HaLoop is needed

  -> Ex) Iterative algorithms, web/graph ranking algorithms,

  recursive graph or network queries

Introduction

- HaLoop extends MapReduce

(1) A MapReduce cluster can cache the invariant data in the first iteration, and then reuse them in later iterations.

(2) A MapReduce cluster can cache reducer outputs, which makes checking for a fixpoint more efficient, without an extra job.

HaLoop Overview

- Architecture

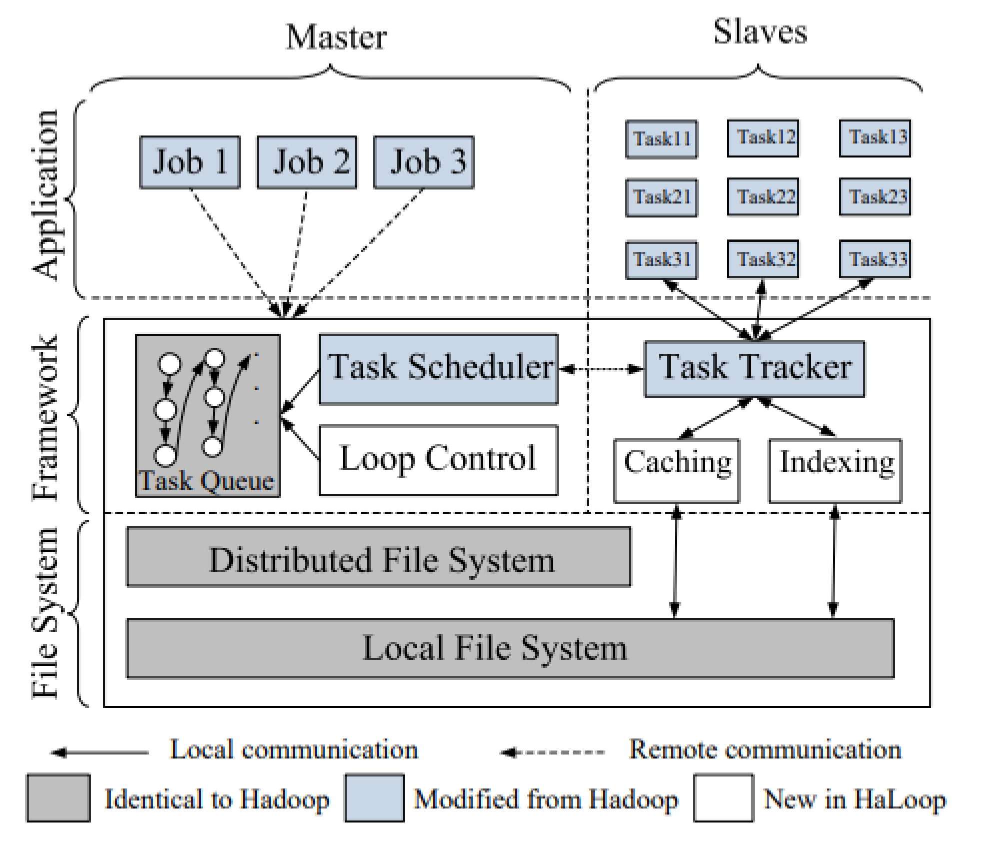


Figure 3: The HaLoop framework, a variant of Hadoop MapReduce framework.

- HDFS: Distributed File System
  -> divided into 2 parts
  (1) A master node
  (2) Many slave nodes

- Client
  -> Master node
  (Schedules parallel tasks)
  -> Slave node
  (has task tracker daemon)
  -> Task
  (map task or reduce task)

HaLoop Overview

- Architecture

- 4 changes from Hadoop MapReduce

(1) New app programming interface for users
that simplifies the expression of iterative MapReduce programs

(2) Master node has a new loop control module
that repeatedly starts new map-reduce steps

(3) New task scheduler for iterative apps
that leverages data locality in these apps

(4) HaLoop caches and indexes app data on slave nodes

HaLoop Overview

- Programming Model

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

R0 = Initial Result
L = Invariant Relation

- HaLoop core construct
  -> can express recursive programs
- It terminates when a fixpoint is reached
  -- when the result does not change from one iteration to the next

$$R_{i+1} = R_i$$

- Fixpoint: defined by exact equality between iterations
  + *Approximate Fixpoint*

HaLoop Overview

- Programming Model -API

- HaLoop API(Application Programming Interface)

(1) To specify the loop body:

-> *Map, Reduce, AddMap, AddReduce*

(2) To terminate the computation:

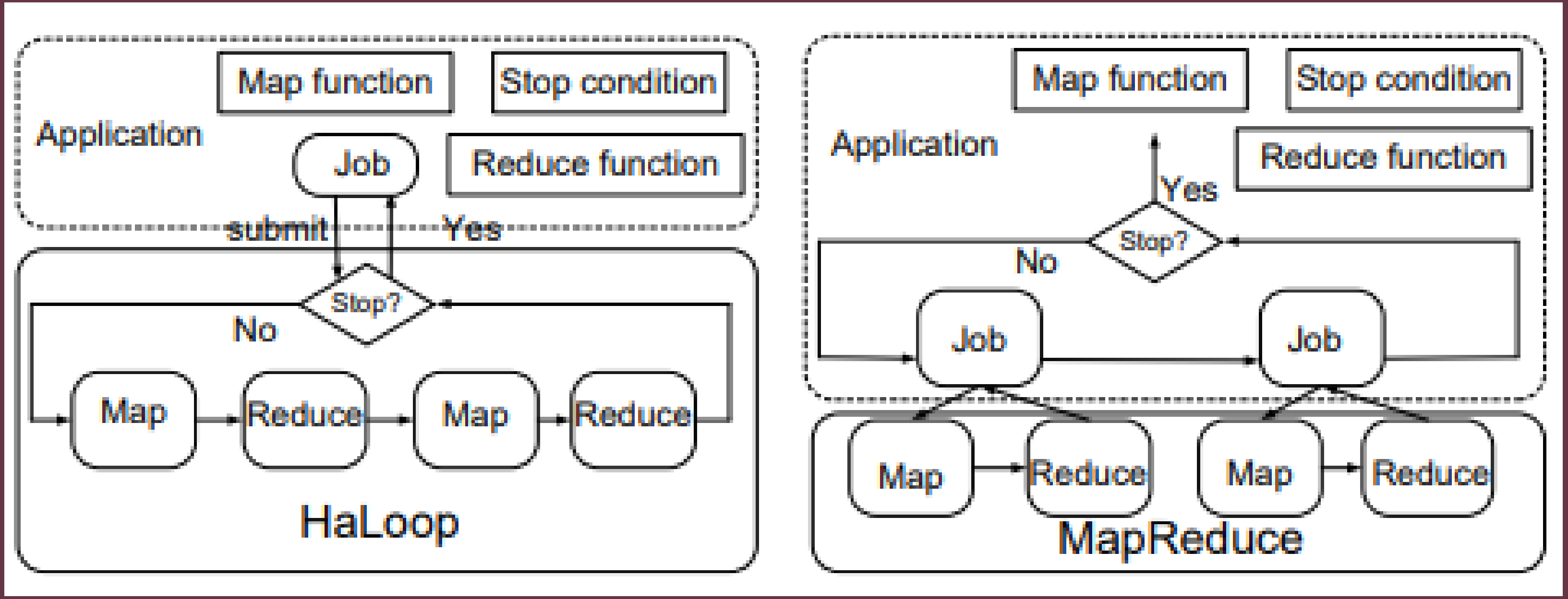
-> *SetFixedPointTreshold, ResultDistance, SetMaxNumofIteration*

(3) To specify and control inputs:

-> *SetIterationInput, AddStepInput, AddInvariantTable*

HaLoop Overview

- Programming Model



Which part controls the Loop?

Loop-Aware Task Scheduling

- Inter-Iteration Locality

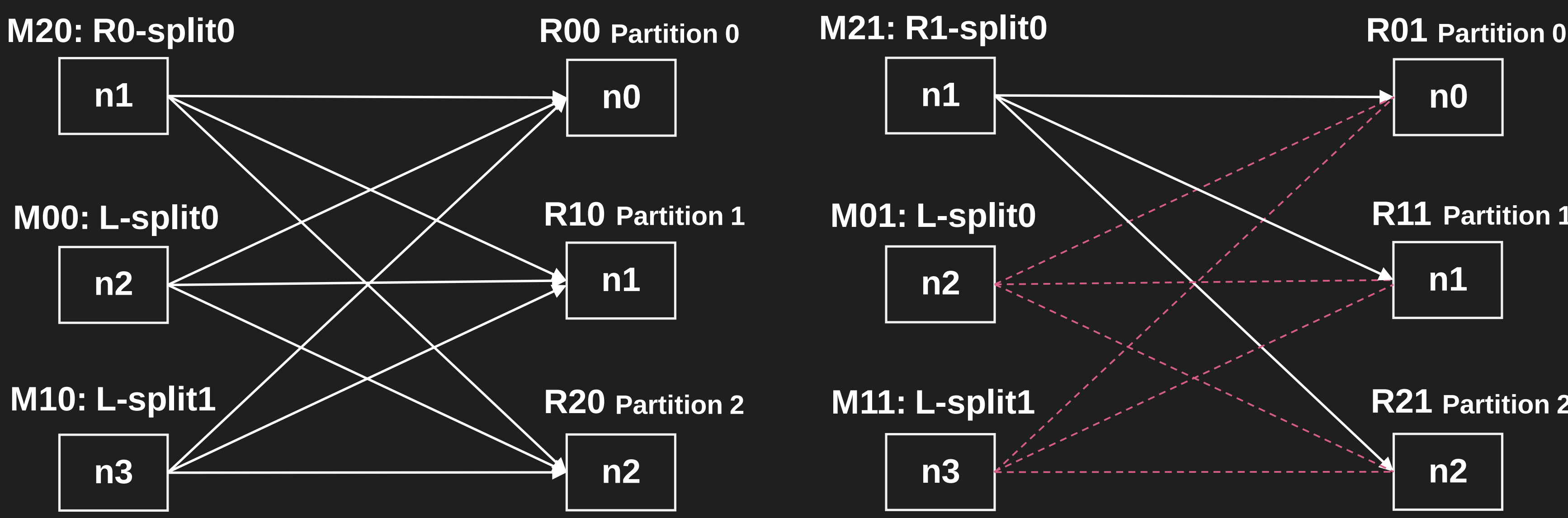
- HaLoop task scheduler: Provides better schedules for iterative program

  -> Goal :
  To place map and reduce tasks
  that occur in different iterations but access the same data
  on the same physical machine

  -> Data can more easily be cached and re-used between iterations

Loop-Aware Task Scheduling

- Inter-Iteration Locality



- Provides the feasibility to reuse loop-invariant data : L
- Inter-iteration locality : the number of reduce tasks should be invariant

Loop-Aware Task Scheduling

- Scheduling Algorithm

- Work Routine of Scheduler:

  (1) Master node receives a heartbeat from a slave node

  (2) Master node tries to give an unassigned task to slave node

  (3) To support, the master node maintains a mapping
  from each slave node to the data partitions that this node processed
  in the previous iterations.

  (4) If the slave node has a full load, the master re-assigns its
  tasks to a nearby slave node.

Loop-Aware Task Scheduling

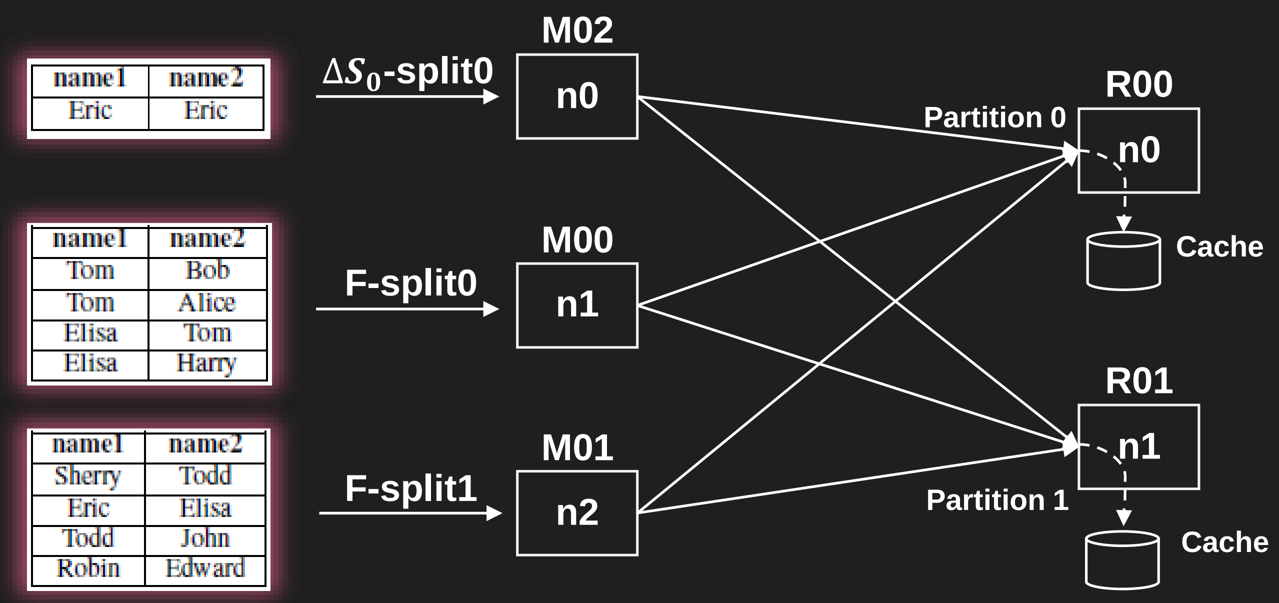
- Scheduling Algorithm

```
// 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
if iteration == 0 then
    Partition part = hadoopSchedule(node);
    current.get(node).add(part);
else
    if node.hasFullLoad() then
        Node substitution = findNearestIdleNode(node);
        previous.get(substitution).addAll(previous.remove(node));
        return;
    end if
    if previous.get(node).size()>0 then
        Partition part = previous.get(node).get(0);
        schedule(part, node);
        current.get(node).add(part);
        previous.remove(part);
    end if
end if
```

- Pseudocode for the scheduling algorithm

Reducer Input Cache

- overall structure

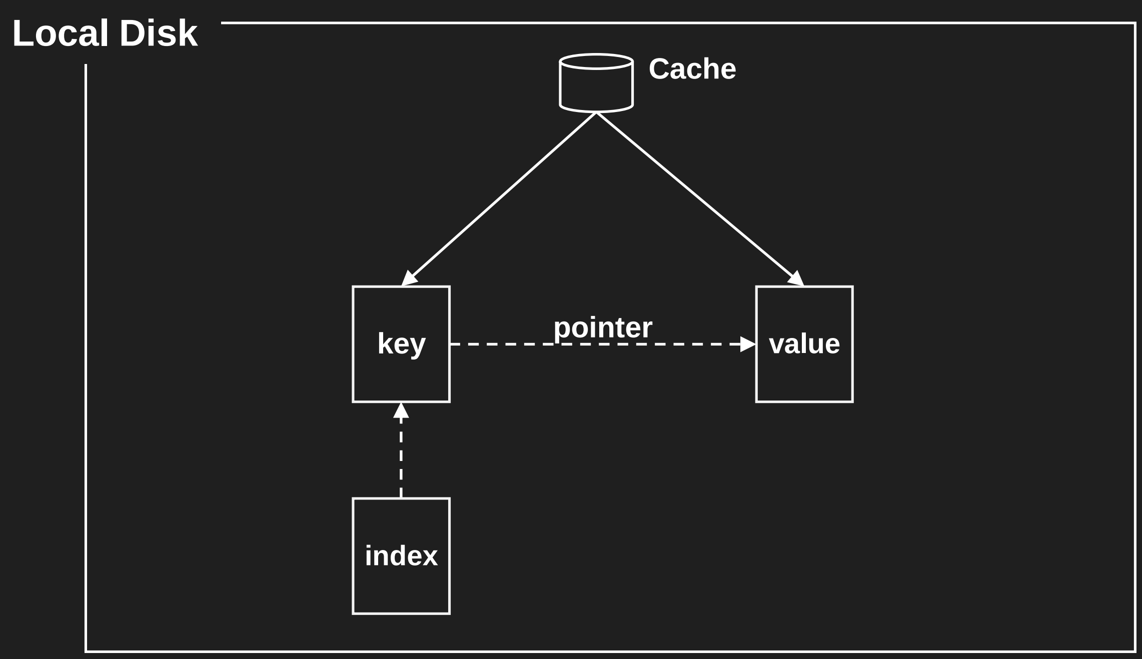


In example 2, F table is loop-invariant.
So, this data is cached as reducer input cache.

Also, mapper outputs in the first iteration are cached in the corresponding mapper's local disk for future reducer cache reloading.

Reducer Input Cache

- physical layout of the cache

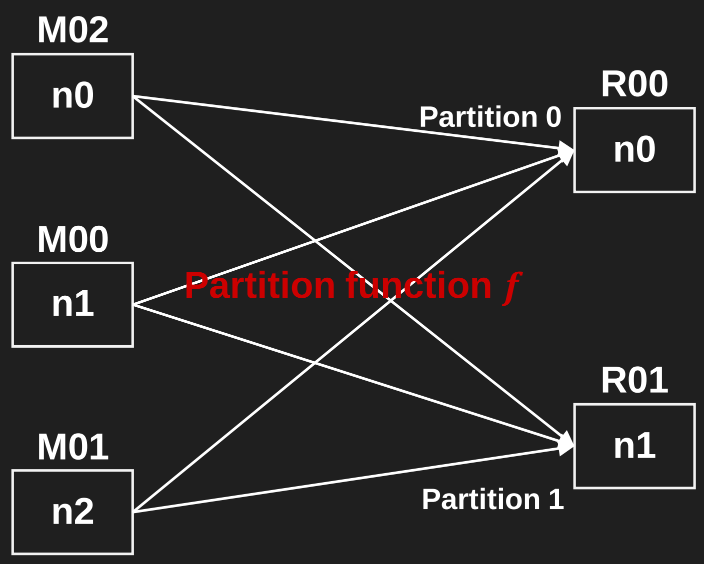


Keys and values are separated into two files.

Sometimes the selectivity in the cached loop-invariant data is low.
Thus, HaLoop creates an index over the keys.

Reducer Input Cache

- Requirements

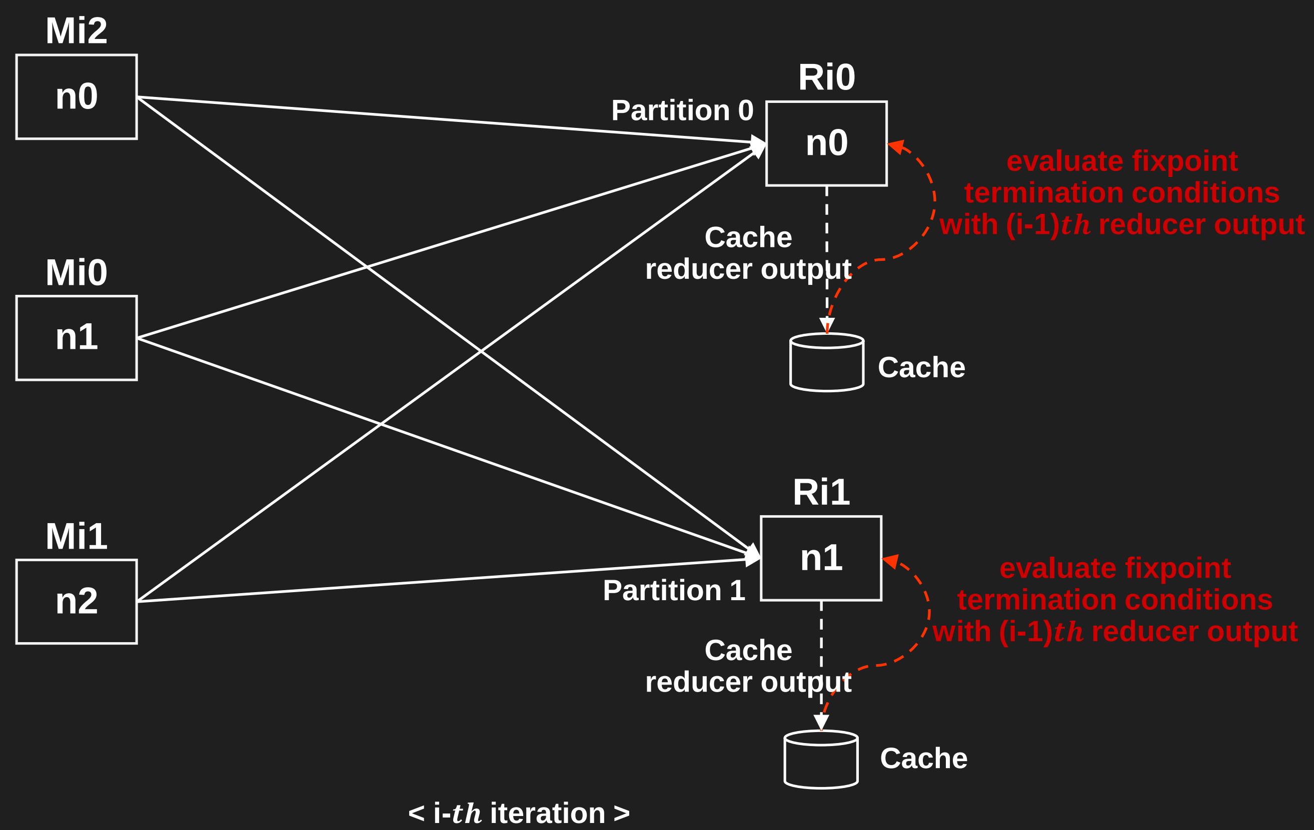


Partition function f should satisfies that :

- f must be deterministic
- f must remain the same across iterations
- f must not take any inputs other than the mapper output tuple

Reducer Output Cache

- overall structure



Reducer Output Cache

- Requirements

In the last map-reduce pair of the loop body, the mapper output partition function f and the reduce function satisfy the following conditions :

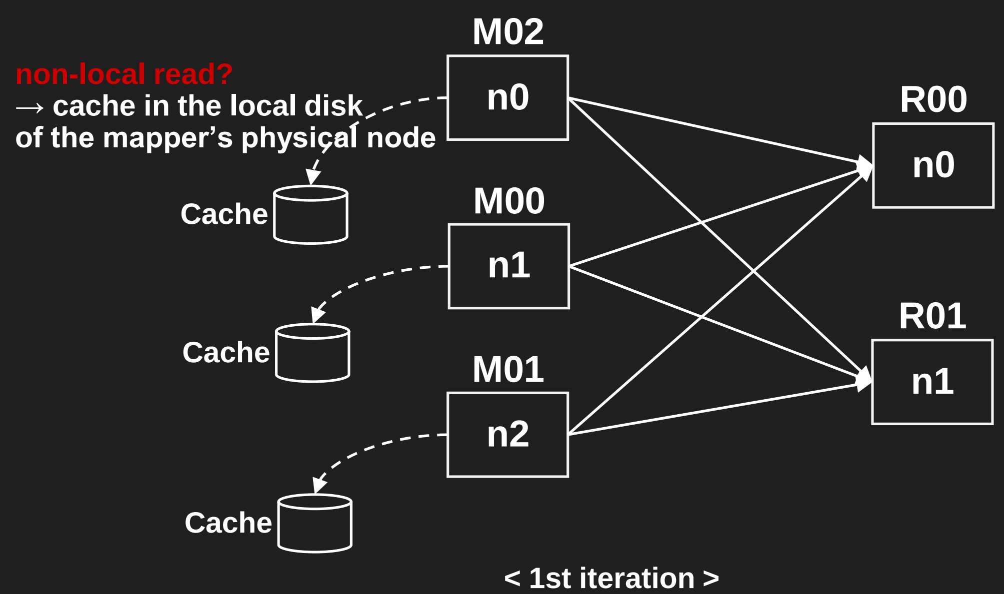
if (k_{o1}, v_{o1})∈reduce(k_i, v_i), (k_{o2}, v_{o2})∈reduce(k_j, v_j), and k_{o1}=k_{o2},
then $f(k_i)$ = $f(k_j)$

k : key
v : value
$i,\ j$: each different iteration

This requirement ensures the usefulness of reducer output cache and the correctness of the local fixpoint evaluation

Mapper Input Cache

- Overall structure



In later iterations, all mappers read data only from local disks.

The mapper input cache can be used by model-fitting applications such a iterative algorithm consuming mapper inputs that do not change across iterations.

Cache Reloading

- Cases

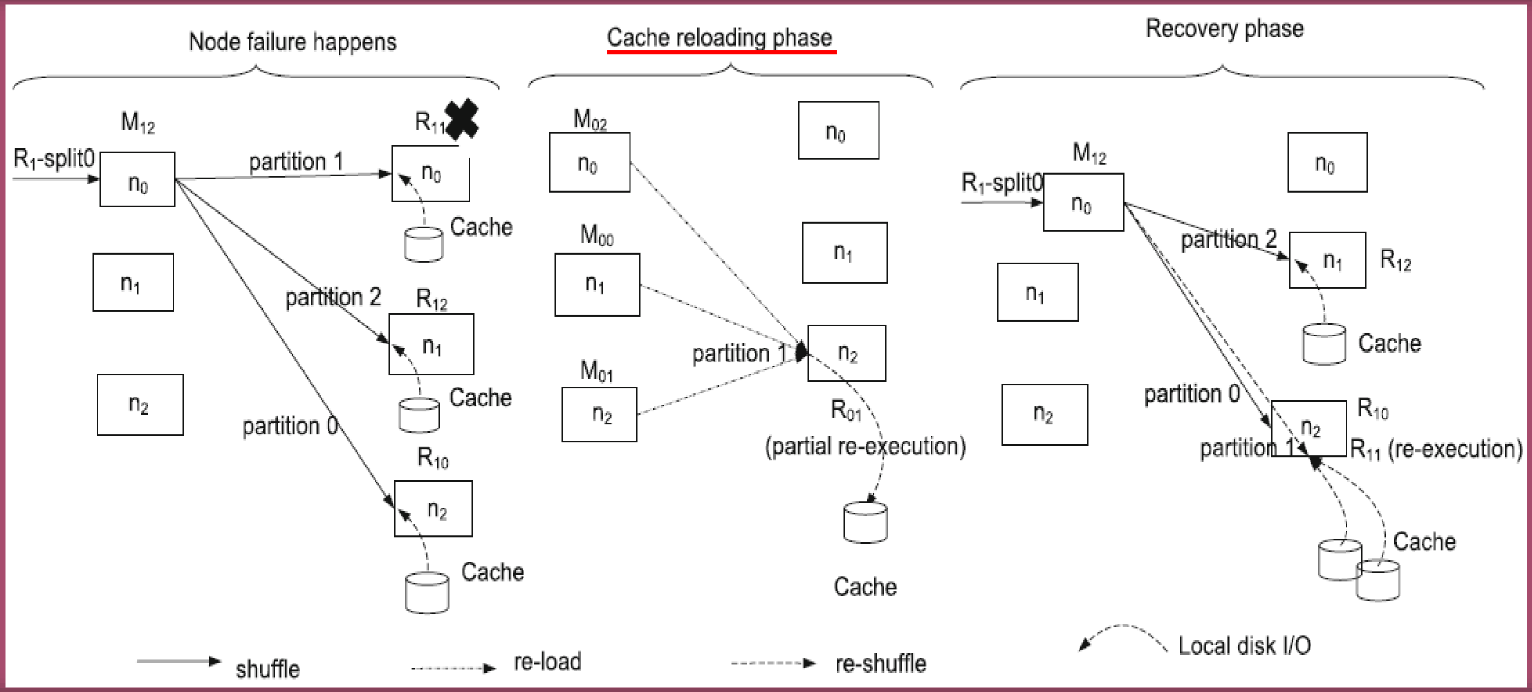
A few cases where the cache must be re-constructed :

- The hosting node fails
- The hosting node has a full load and a map or reduce task must be scheduled on a different substitution node

Cache Reloading

- How to reload cache?

- Reducer Input Cache
  → by copying the desired partition from all first-iteration mapper outputs



Cache Reloading

- How to reload cache?

- Mapper Input Cache and Reducer Output Cache

  → mapper/reducer only needs to read the corresponding chunks from the distributed file system, where replicas of the cached data are stored

Experimental evaluation

- Environment

- Experimental environment

  → used virtual machine cluster of 50 and 90 slave nodes
  in Amazon EC2

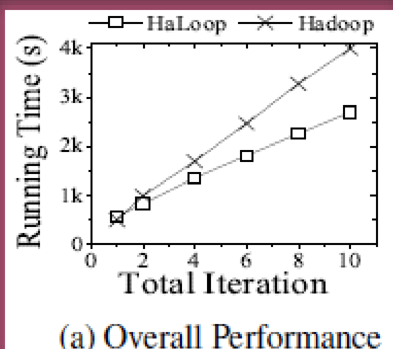
  → used both semi-synthetic and real-world datasets :
  - Livejournal (18GB, social network data) for PageRank, descendent
  - Triples (120GB, semantic web data) for descendant query
  - Freebase (12GB, concept linkage graph) for PageRank

Experimental evaluation

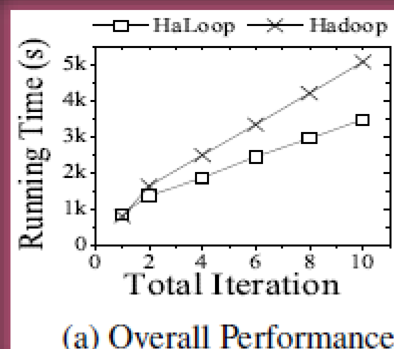
- Evaluation of Reducer Input Cache

Overall Run Time

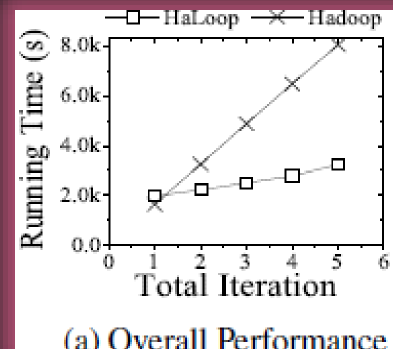
→ used SetMaxNumOfIterations to specify the loop termination condition



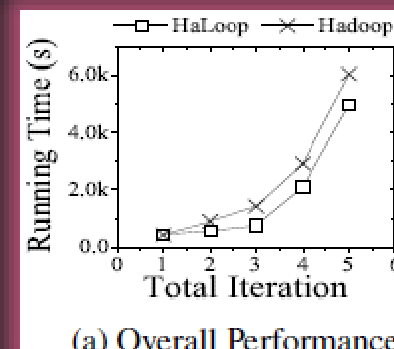
A. PageRank (Livejournal, 50nodes)



B. PageRank (Freebase, 90nodes)



C. Descendant Query (Triples, 90nodes)

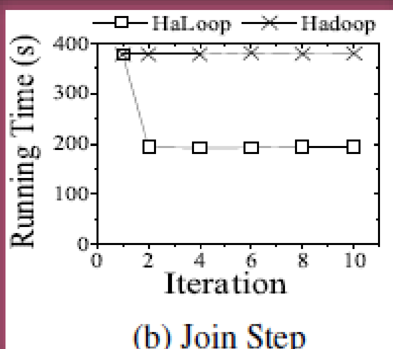


D. Descendant Query (Livejournal, 50nodes)

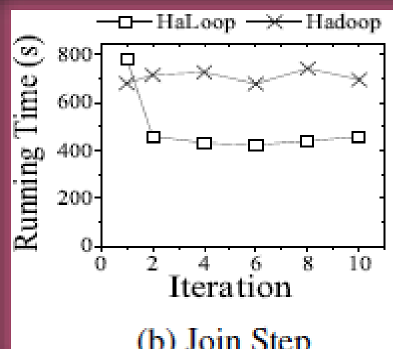
Experimental evaluation

- Evaluation of Reducer Input Cache

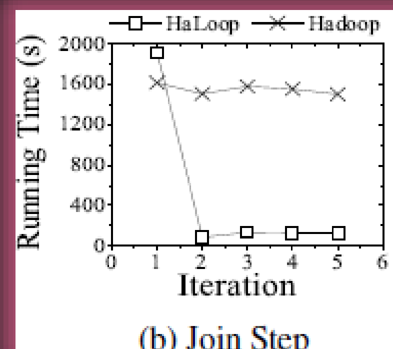
Join Step Run Time



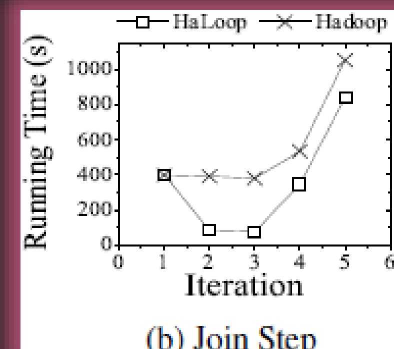
A. PageRank
(Livejournal, 50nodes)



B. PageRank
(Freebase, 90nodes)



C. Descendant Query
(Triples, 90nodes)

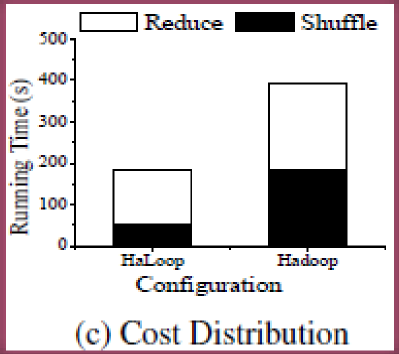


D. Descendant Query
(Livejournal, 50nodes)

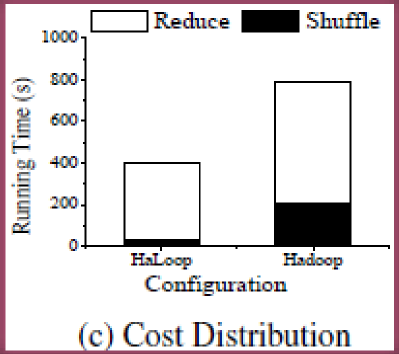
Experimental evaluation

- Evaluation of Reducer Input Cache

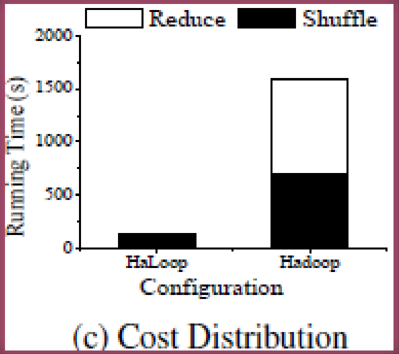
Cost Distribution for Join Step



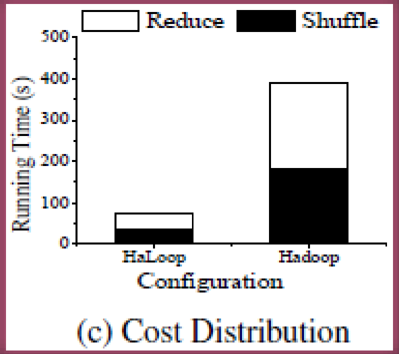
A. PageRank (Livejournal, 50nodes)



B. PageRank (Freebase, 90nodes)



C. Descendant Query (Triples, 90nodes)

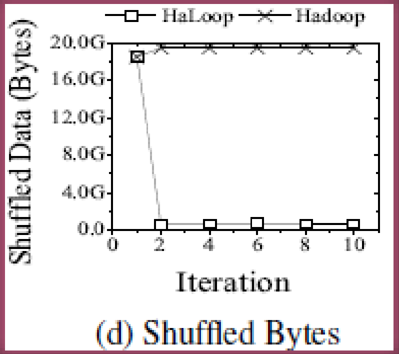


D. Descendant Query (Livejournal, 50nodes)

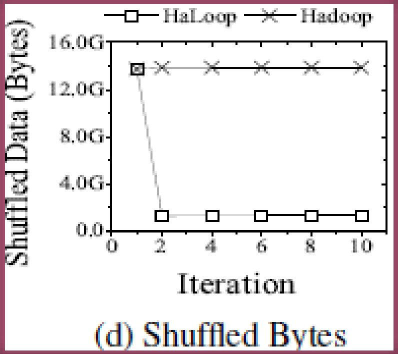
Experimental evaluation

- Evaluation of Reducer Input Cache

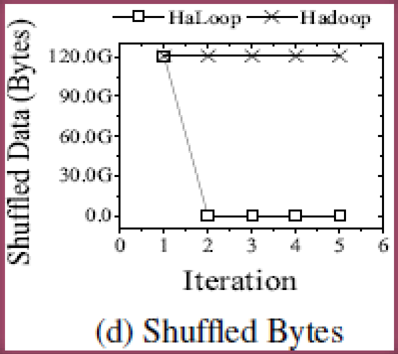
I/O in Shuffle Phase of Join Step



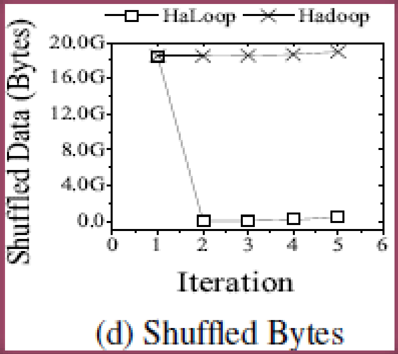
A. PageRank (Livejournal, 50nodes)



B. PageRank (Freebase, 90nodes)



C. Descendant Query (Triples, 90nodes)

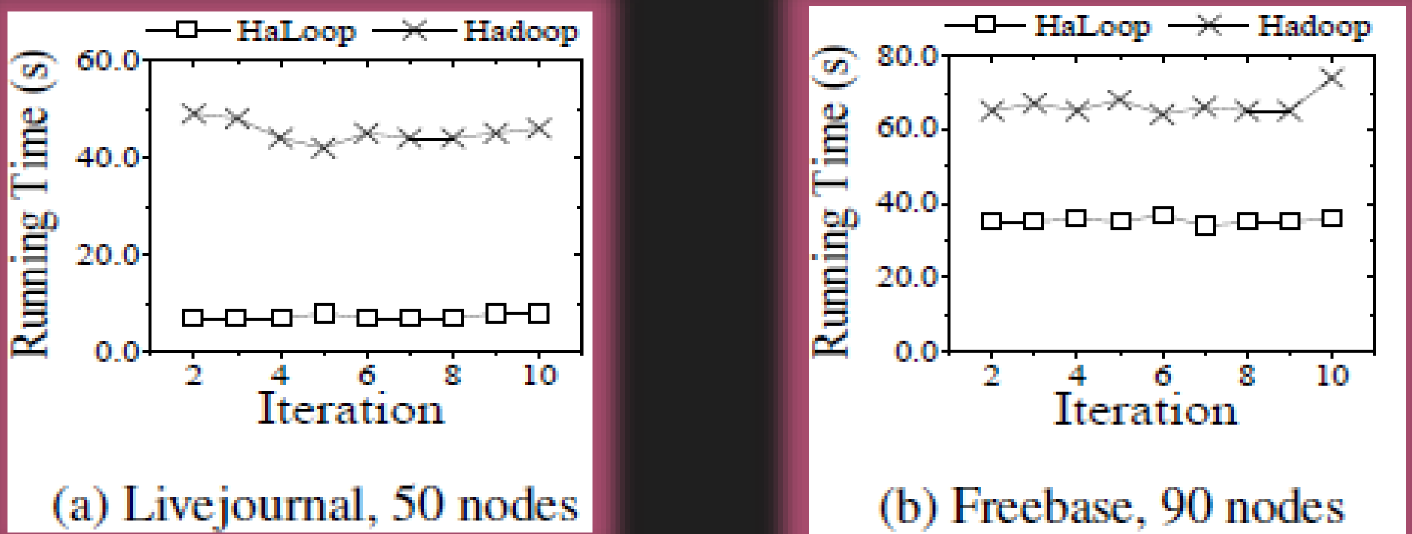


D. Descendant Query (Livejournal, 50nodes)

Experimental evaluation

- Evaluation of Reducer Output Cache

On average, compared with Hadoop, HaLoop reduces the cost of the fixpoint evaluation to 40%

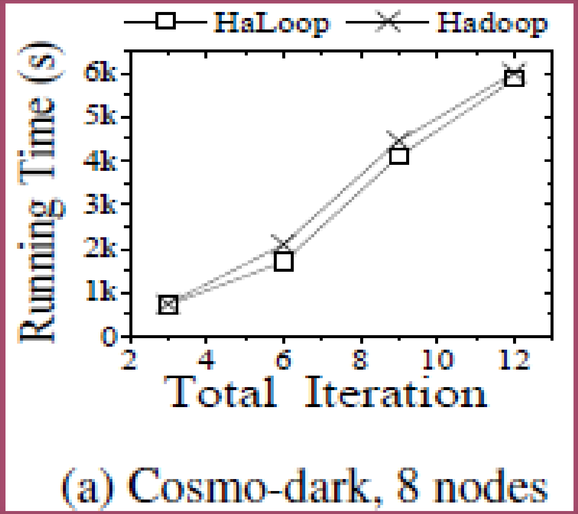


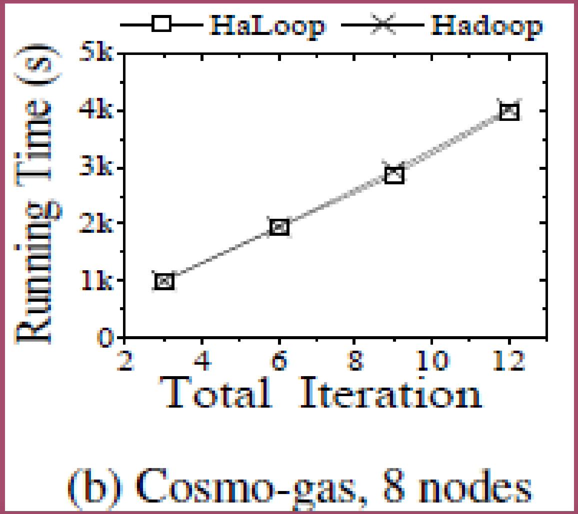
Fixpoint Evaluation Overhead in PageRank : HaLoop vs. Hadoop

Experimental evaluation

- Evaluation of Mapper Input Cache

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.





Performance of k-means : HaLoop vs. Hadoop

Conclusion

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 scheduler

(2) loop-invariant data caching

(3) caching for efficient fixpoint verification

With these features,
HaLoop improves the overall performance of iterative data analysis applications

THANK YOU