Data-Intensive Computing with MapReduce

Session 11: Beyond MapReduce

Jimmy Lin
University of Maryland
Thursday, April 11, 2013

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details
Today’s Agenda

- Making Hadoop more efficient
- Tweaking the MapReduce programming model
- Beyond MapReduce
Hadoop is slow...
A Major Step Backwards?

- MapReduce is a step backward in database access:
  - Schemas are good
  - Separation of the schema from the application is good
  - High-level access languages are good

- MapReduce is poor implementation
  - Brute force and only brute force (no indexes, for example)

- MapReduce is not novel

- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions…

- MapReduce is incompatible with DMBS tools

Source: Blog post by DeWitt and Stonebraker
Hadoop vs. Databases: Grep

Figure 4: Grep Task Results – 535MB/node Data Set

SELECT * FROM Data WHERE field LIKE ‘%XYZ%’;

Figure 5: Grep Task Results – 1TB/cluster Data Set

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
Hadoop vs. Databases: Select

**Figure 6: Selection Task Results**

```
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;
```
Hadoop vs. Databases: Aggregation

**Figure 7:** Aggregation Task Results (2.5 million Groups)

**Figure 8:** Aggregation Task Results (2,000 Groups)

```sql
SELECT sourceIP, SUM(adRevenue) 
FROM UserVisits GROUP BY sourceIP;
```

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
Hadoop vs. Databases: Join

Figure 9: Join Task Results

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
Schemas are a good idea!

- Parsing fields out of flat text files is slow
- Schemas define a contract, decoupling logical from physical
Thrift

- Originally developed by Facebook, now an Apache project
- Provides a DDL with numerous language bindings
  - Compact binary encoding of typed structs
  - Fields can be marked as optional or required
  - Compiler automatically generates code for manipulating messages
- Provides RPC mechanisms for service definitions
- Alternatives include protobufs and Avro
struct Tweet {
    1: required i32 userId;
    2: required string userName;
    3: required string text;
    4: optional Location loc;
}

struct Location {
    1: required double latitude;
    2: required double longitude;
}
Row vs. Column Stores

Row store

Column store
Row vs. Column Stores

- **Row stores**
  - Easy to modify a record
  - Might read unnecessary data when processing

- **Column stores**
  - Only read necessary data when processing
  - Tuple writes require multiple accesses
OLTP/OLAP Architecture

OLTP

ETL
(Extract, Transform, and Load)

OLAP
Advantages of Column Stores

- Read efficiency
- Better compression
- Vectorized processing
- Opportunities to operate directly on compressed data
Why not in Hadoop? No reason why not

III. THE DESIGN AND IMPLEMENTATION OF RCFile

In this section, we present RCFile (Record Columnar File), a fast and space-efficient data placement structure designed for MapReduce-based warehouse systems, such as Hive. RCFile applies the concept of "first horizontally-partition, then vertically-partition" from PAX. It combines the advantages of both row-store and column-store. First, as row-store, RCFile guarantees that data in the same row are located in the same node, thus it has low cost of tuple reconstruction. Second, as column-store, RCFile can exploit a column-wise data compression and skip unnecessary column reads.

A. Data Layout and Compression

RCFile is designed and implemented on top of the Hadoop Distributed File System (HDFS). As demonstrated in the example shown in Figure 3, RCFile has the following data layout to store a table:

1) According to the HDFS structure, a table can have multiple HDFS blocks.
2) In each HDFS block, RCFile organizes records with the basic unit of a row group. That is, all the records stored in an HDFS block are partitioned into row groups. For a table, all row groups have the same size. Depending on the row group size and the HDFS block size, an HDFS block can have only one or multiple row groups.

Fig. 3: An example to demonstrate the data layout of RCFile in an HDFS block.

3) A row group contains three sections. The first section is a sync marker that is placed at the beginning of the row group. The sync marker is mainly used to separate two continuous row groups in an HDFS block. The second section is a metadata header for the row group. The metadata header stores the information items on how many records are in this row group, how many bytes are in each column, and how many bytes are in each field in a column. The third section is the table data section that is actually a column-store. In this section, all the fields in the same column are stored continuously together. For example, as shown in Figure 3, the section first stores all fields in column A, and then all fields in column B, and so on.

We now introduce how data is compressed in RCFile. In each row group, the metadata header section and the table data section are compressed independently as follows.

- First, for the whole metadata header section, RCFile uses the RLE (Run Length Encoding) algorithm to compress data. Since all the values of the field lengths in the same column are continuously stored in this section, the RLE algorithm can find long runs of repeated data values, especially for fixed field lengths.
- Second, the table data section is not compressed as a whole unit. Rather, each column is independently compressed with the Gzip compression algorithm. RCFile uses the heavy-weight Gzip algorithm in order to get better compression ratios than other light-weight algorithms. For example, the RLE algorithm is not used since the column data is not already sorted. In addition, due to the lazy decompression technology to be discussed next, RCFile does not need to decompress all the columns when processing a row group. Thus, the relatively high decompression overhead of the Gzip algorithm can be reduced.

Though currently RCFile uses the same algorithm for all columns in the table data section, it allows us to use different algorithms to compress different columns. One future work related to the RCFile project is to automatically select the best compression algorithm for each column according to its data type and data distribution.

B. Data Appending

RCFile does not allow arbitrary data writing operations. Only an appending interface is provided for data writing in RCFile because the underlying HDFS currently only supports data writes to the end of a file. The method of data appending in RCFile is summarized as follows.

1) RCFile creates and maintains an in-memory column holder for each column. When a record is appended, all its fields will be scattered, and each field will be appended into its corresponding column holder. In addition, RCFile will record corresponding metadata of each field in the metadata header.
2) RCFile provides two parameters to control how many records can be buffered in memory before they are...
“Trojan Layouts”

We propose Trojan Layouts as our solution to decrease the wait time of data-intensive jobs when accessing data from HDFS.

2. OVERVIEW

HDFS, and (3) from both PAX and Row Layouts. In summary, we make the following contributions to future distributed systems as emphasized in the conclusion section.

1.3 Contributions

The core idea of Trojan Layouts is to internally organize data. We exploit existing data block scheduling aspects of our approach. Given a query workload $W$, we seamlessly wrap the input to other Trojan Layouts. An important feature of our approach is that we keep the Hadoop MapReduce interface intact by providing a (per-replica) Trojan Layout.

We use di↵erent Trojan Layouts for di↵erent data block replicas, grouping as well, i.e. for clustering queries in a workload across all data block replicas. We illustrate Trojan Layout for each data block replica. We then store each data block replica in a di↵erent Trojan Layout. We provide the details on how we compute Trojan Layouts in Section 3. Then, in Section 4, we describe how we enable HDFS to create different Trojan Layouts for di↵erent data block replicas.

MapReduce, users only care about the right data to Column Group 1, and so on. Each column group in turn contains attribute A. The second attribute pointer would then point to the beginning of the column group that contains that attribute. For instance, in Replica 1, the first attribute pointer would point to Column Group 1, which contains attribute A.

At query time, we transparently adapt an incoming MapReduce job to query the data block replica that minimizes the data access time of a data block. As a result, we do not require any change to other Trojan Layouts. An important feature of our approach is that it hides all messy details from the user. We do not change the outside HDFS unit, i.e. a Trojan Layout for each data block replica. We use this algorithm as a basis to determine which Layout is the best for a given query workload.
Key Ideas

- Separate logical from physical
- Preserve HDFS block structure
- Hide physical storage layout behind InputFormats
Hadoop vs. Databases: Grep (Revisited)

Figure 4: Grep Task Results – 535MB/node Data Set

Figure 5: Grep Task Results – 1TB/cluster Data Set

SELECT * FROM Data WHERE field LIKE ‘%XYZ%’;

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
Hadoop vs. Databases: Select (Revisited)

**Figure 6: Selection Task Results**

\[
\text{SELECT pageURL, pageRank} \\
\text{FROM Rankings WHERE pageRank > X;}
\]

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
Final consideration: Cost
Indexes are a good thing!

Source: Wikipedia (Card Catalog)
Why not in Hadoop? No reason why not

- Non-invasive: requires no changes to Hadoop infrastructure
- Useful for speeding up selections on joins
- Indexing building itself can be performed using MapReduce

Source: Dittrich et al. (2010) Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). VLDB.
A Major Step Backwards?

- MapReduce is a step backward in database access:
  - Schemas are good
  - Separation of the schema from the application is good
  - High-level access languages are good

- MapReduce is poor implementation
  - Brute force and only brute force (no indexes, for example)

- MapReduce is not novel

- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions...

- MapReduce is incompatible with DMBS tools

Source: Blog post by DeWitt and Stonebraker
Pig!

Source: Wikipedia (Pig)
### Pig: Example

**Task: Find the top 10 most visited pages in each category**

<table>
<thead>
<tr>
<th>Visits</th>
<th>Url Info</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User</strong></td>
<td><strong>Url</strong></td>
</tr>
<tr>
<td>Amy</td>
<td>cnn.com</td>
</tr>
<tr>
<td>Amy</td>
<td>bbc.com</td>
</tr>
<tr>
<td>Amy</td>
<td>flickr.com</td>
</tr>
<tr>
<td>Fred</td>
<td>cnn.com</td>
</tr>
</tbody>
</table>

Pig Slides adapted from Olston et al. (SIGMOD 2008)
**Pig Query Plan**

- **Load** Visits
- **Group** by url
  - **Foreach** url
    - **generate** count
- **Load** Url Info
- **Join** on url
- **Group** by category
- **Foreach** category
  - **generate** top10(urls)

Pig Slides adapted from Olston et al. (SIGMOD 2008)
Pig Script

visits = load '../data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);
urlInfo = load '../data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into '../data/topUrls';
Pig Script in Hadoop

Map\(_1\)

Load Visits

Reduce\(_1\)

Group by url

Reduce\(_2\)

Load Url Info

Reduce\(_3\)

Foreach url generate count

Join on url

Foreach category generate top10(urls)

Pig Slides adapted from Olston et al. (SIGMOD 2008)
Pig: Basics

- Sequence of statements manipulating relations (aliases)
- Data model
  - atoms
  - tuples
  - bags
  - maps
  - json
Pig: Common Operations

- LOAD: load data
- FOREACH … GENERATE: per tuple processing
- FILTER: discard unwanted tuples
- GROUP/COGROUP: group tuples
- JOIN: relational join
Pig: GROUPing

A = LOAD 'myfile.txt' AS (f1: int, f2: int, f3: int);

(1, 2, 3)
(4, 2, 1)
(8, 3, 4)
(4, 3, 3)
(7, 2, 5)
(8, 4, 3)

X = GROUP A BY f1;

(1, {(1, 2, 3)})
(4, {(4, 2, 1), (4, 3, 3)})
(7, {(7, 2, 5)})
(8, {(8, 3, 4), (8, 4, 3)})
Pig: COGROU ping

A:
(1, 2, 3)
(4, 2, 1)
(8, 3, 4)
(4, 3, 3)
(7, 2, 5)
(8, 4, 3)

B:
(2, 4)
(8, 9)
(1, 3)
(2, 7)
(2, 9)
(4, 6)
(4, 9)

X = COGROUP A BY f1, B BY $0;

(1, {(1, 2, 3)}, {(1, 3)})
(2, {}, {(2, 4), (2, 7), (2, 9)})
(4, {(4, 2, 1), (4, 3, 3)}, {(4, 6), (4, 9)})
(7, {(7, 2, 5)}, {}}
(8, {(8, 3, 4), (8, 4, 3)}, {(8, 9)})
Pig UDFs

- User-defined functions:
  - Java
  - Python
  - JavaScript
  - Ruby

- UDFs make Pig arbitrarily extensible
  - Express “core” computations in UDFs
  - Take advantage of Pig as glue code for scale-out plumbing
PageRank in Pig

previous_pagerank = LOAD ‘$docs_in’ USING PigStorage()
    AS (url: chararray, pagerank: float,
        links:{link: (url: chararray)});

outbound_pagerank = FOREACH previous_pagerank
    GENERATE pagerank / COUNT(links) AS pagerank,
    FLATTEN(links) AS to_url;

new_pagerank =
    FOREACH ( COGROUP outbound_pagerank
        BY to_url, previous_pagerank BY url INNER )
    GENERATE group AS url,
        (1 – $d) + $d * SUM(outbound_pagerank.pagerank) AS pagerank,
        FLATTEN(previous_pagerank.links) AS links;

STORE new_pagerank INTO ‘$docs_out’ USING PigStorage();
from org.apache.pig.scripting import *
P = Pig.compile(""" Pig part goes here """")

params = { 'd': '0.5', 'docs_in': 'data/pagerank_data_simple' }

for i in range(10):
    out = "out/pagerank_data_" + str(i + 1)
    params["docs_out"] = out
    Pig.fs("rmr " + out)
    stats = P.bind(params).runSingle()
    if not stats.isSuccessful():
        raise 'failed'
    params["docs_in"] = out

Oh, the iterative part too…
Why write MapReduce code in Java ever?
Hadoop + DBs = HadoopDB

- Why not have the best of both worlds?
  - Parallel databases focused on performance
  - Hadoop focused on scalability, flexibility, fault tolerance

- Key ideas:
  - Co-locate a RDBMS on every slave node
  - To the extent possible, “push down” operations into the DB

Source: Abouzeid et al. (2009) HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. VLDB.
HadoopDB Architecture

Source: Abouzeid et al. (2009) HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. VLDB.
The optimizer restructures the query plan to create a more executable MapReduce job for a simple GroupBy-Aggregation query. The join operation can be pushed entirely into the database layer as some tables are colocated and if partitioned on the same attribute. The HadoopDB by pushing most of the query processing logic into the database layer ranges from none (each operator forwards a data tuple to the meta information such as the Deserializer and InputFormat classes required to scan the table to retrieve the schema of the sales table. It also provides information about the table schemas and required Deserializer and InputFormat classes to the MetaStore. We implemented these specialized classes.

The hashing functions used by both the Global Hasher and the Local Hasher are pushed into the database layer. SMS uses a rule-based SQL generator to recreate references to our database tables. Hive allows tables to exist externally and (iii) finally, outside HDFS. The HadoopDB catalog, Section 5.2.2, populates different data structures with meta information such as

HadoopDB: Query Plans

SELECT YEAR(saleDate), SUM(revenue) FROM sales GROUP BY YEAR(saleDate);
MapReduce Sucks: Iterative Algorithms

- Java verbosity
- Hadoop task startup time
- Stragglers
- Needless data shuffling
- Checkpointing at each iteration
HaLoop: MapReduce + Iteration

Programming Model

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

(a) Initial Rank Table \( R_0 \)

<table>
<thead>
<tr>
<th>url</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.b.com">www.b.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.d.com">www.d.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td>1.0</td>
</tr>
</tbody>
</table>

(b) Linkage Table \( L \)

<table>
<thead>
<tr>
<th>url_source</th>
<th>url_dest</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td><a href="http://www.b.com">www.b.com</a></td>
</tr>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td><a href="http://www.c.com">www.c.com</a></td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td><a href="http://www.a.com">www.a.com</a></td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td><a href="http://www.d.com">www.d.com</a></td>
</tr>
<tr>
<td><a href="http://www.d.com">www.d.com</a></td>
<td><a href="http://www.b.com">www.b.com</a></td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td><a href="http://www.e.com">www.e.com</a></td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td><a href="http://www.c.com">www.c.com</a></td>
</tr>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td><a href="http://www.d.com">www.d.com</a></td>
</tr>
</tbody>
</table>

(c) Loop Body

MR_1 \( \left\{ \begin{align*}
T_1 &= R_i \bowtie_{\text{url} = \text{url}_\text{source}} L \\
T_2 &= \gamma_{\text{url}, \text{rank}} \cdot \frac{\text{rank}}{\text{COUNT}(\text{url}_\text{dest})} \rightarrow \text{new_rank}(T_1)
\end{align*} \right. \)

MR_2 \( \left\{ \begin{align*}
T_3 &= T_2 \bowtie_{\text{url} = \text{url}_\text{source}} L \\
R_{i+1} &= \gamma_{\text{url}_\text{dest} \rightarrow \text{url}, \text{SUM}(\text{new_rank})} \rightarrow \text{rank}(T_3)
\end{align*} \right. \)

(d) Rank Table \( R_3 \)

<table>
<thead>
<tr>
<th>url</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td>2.13</td>
</tr>
<tr>
<td><a href="http://www.b.com">www.b.com</a></td>
<td>3.89</td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td>2.60</td>
</tr>
<tr>
<td><a href="http://www.d.com">www.d.com</a></td>
<td>2.60</td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td>2.13</td>
</tr>
</tbody>
</table>

Source: Bu et al. (2010) HaLoop: Efficient Iterative Data Processing on Large Clusters. VLDB.
The iterative programs that HaLoop supports can be distilled into a general form:

\[ R_{i+1} = f(R_i) \]

where \( R_0 \) is an initial result and \( R \) is an invariant relation. A fixpoint is reached when the result does not change from one iteration to the next, i.e., \( R_{i+1} = R_i \). The loop terminates when a fixpoint is reached — where the distance between the current and previous iteration's outputs is less than a user-specified threshold. For example, in Example 1, at each iteration, \( R_{i+1} \) is an approximation of the result. The programmer uses an approximate fixpoint termination condition, the programmer uses the following functions.

- **SetFixedPointThreshold**: Sets the approximate fixpoint threshold.
- **SetMaxNumOfIterations**: Sets the maximum number of iterations.
- **AddMap**: Adds a map function.
- **AddReduce**: Adds a reduce function.
- **SetIterationInput**: Associates an input source with a specific key.
- **ResultDistance**: Calculates the distance between the reducer outputs of the current iteration and the previous iteration.

In order to accommodate the requirements of iterative data analytics, HaLoop leverages data locality in these applications (Section 3). Fourth, HaLoop uses a new task scheduler for iterative applications that starts new map-reduce steps that compose the loop body, until a user-specified stopping condition is met (Section 2.2). Third, HaLoop not only manages task execution, but also manages caches and indexes application data on slave nodes (Section 2.2). Second, HaLoop's programming interface to users that simplifies the expression of iterative MapReduce programs; we posit that a variety of high-level operations besides SQL such as including complex analytics that are not typically implemented in a high-level declarative language for expressing recursive queries. Rather, we focus on providing an efficient foundation API for iterative programs we intend to support, this work does not develop a detailed API.

To specify and control inputs, the programmer uses:

- **SetMaxNumOfIterations**: Sets the maximum number of iterations.
- **AddMap**: Adds a map function.
- **AddReduce**: Adds a reduce function.
- **SetIterationInput**: Associates an input source with a specific key.

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.

1. **Job Introduction**: A job is defined as an input to HaLoop. Each job is a multi-step MapReduce computation, and a job can be executed concurrently with other jobs.
2. **Job Definition**: A job is specified by the programmer and consists of a loop body and a termination condition.
3. **Job Execution**: The master node contains a new loop control module that repeatedly executes the loop body until a termination condition is met.
4. **Job Control**: The master node not only manages task execution, but also manages caches and indexes application data on slave nodes.
HaLoop: Loop Aware Scheduling

This section introduces the HaLoop task scheduler. The scheduling of iteration 6 is no different than in Hadoop. In that case, in iteration 8 only one mapper will copy intermediate data from one reducer to another node.

The high-level goal of HaLoop's scheduler is to place the mapper output key in the output of the next map-reduce pair, and the local disk. The output of the mapper input is no different, but it now comes from two sources—a file and a file splited.

Three map tasks are executed, each of which loads a partition of the join step of the first iteration. The input tables are two iterations and three slave nodes involved in the job. Figure 6x of the PageRank application from Example 6x shows the difference between HaLoop and Hadoop.

The mapper output key is hashed using this programming interface. Figure 0 is a sample schedule for the join step with iteration locality. Tasks pair.

AddStepInput

AddInvariantTable

This programming interface is sufficient to express a variety of representations are equal. Thanks to the inter-iteration locality offered by the task scheduler, the feasibility to reuse loop-invariant data from past iterations can be achieved.

More specifically, the HaLoop scheduler works as follows. Upon submitting a job, the master remembers the association between the slave node where processes a specific data partition and the current job. To support this assignment, the master node maintains a hashmap.

At any node, the master association between the slave node and the current job must be stored. The master node checks if the node already has a task that needs to be launched, and thus the three reducers are assigned to the slave node. To support this assignment, the master node maintains a hashmap.

The mapper output key is hashed using this programming interface. Figure 0 is a sample schedule for the join step with iteration locality. Tasks pair.

AddStepInput

AddInvariantTable

This programming interface is sufficient to express a variety of representations are equal. Thanks to the inter-iteration locality offered by the task scheduler, the feasibility to reuse loop-invariant data from past iterations can be achieved.

More specifically, the HaLoop scheduler works as follows. Upon submitting a job, the master remembers the association between the slave node where processes a specific data partition and the current job. To support this assignment, the master node maintains a hashmap.

At any node, the master association between the slave node and the current job must be stored. The master node checks if the node already has a task that needs to be launched, and thus the three reducers are assigned to the slave node. To support this assignment, the master node maintains a hashmap.

The mapper output key is hashed using this programming interface. Figure 0 is a sample schedule for the join step with iteration locality. Tasks pair.

AddStepInput

AddInvariantTable

This programming interface is sufficient to express a variety of representations are equal. Thanks to the inter-iteration locality offered by the task scheduler, the feasibility to reuse loop-invariant data from past iterations can be achieved.

More specifically, the HaLoop scheduler works as follows. Upon submitting a job, the master remembers the association between the slave node where processes a specific data partition and the current job. To support this assignment, the master node maintains a hashmap.

At any node, the master association between the slave node and the current job must be stored. The master node checks if the node already has a task that needs to be launched, and thus the three reducers are assigned to the slave node. To support this assignment, the master node maintains a hashmap.

The mapper output key is hashed using this programming interface. Figure 0 is a sample schedule for the join step with iteration locality. Tasks pair.

AddStepInput

AddInvariantTable

This programming interface is sufficient to express a variety of representations are equal. Thanks to the inter-iteration locality offered by the task scheduler, the feasibility to reuse loop-invariant data from past iterations can be achieved.

More specifically, the HaLoop scheduler works as follows. Upon submitting a job, the master remembers the association between the slave node where processes a specific data partition and the current job. To support this assignment, the master node maintains a hashmap.

At any node, the master association between the slave node and the current job must be stored. The master node checks if the node already has a task that needs to be launched, and thus the three reducers are assigned to the slave node. To support this assignment, the master node maintains a hashmap.

The mapper output key is hashed using this programming interface. Figure 0 is a sample schedule for the join step with iteration locality. Tasks pair.

AddStepInput

AddInvariantTable

This programming interface is sufficient to express a variety of representations are equal. Thanks to the inter-iteration locality offered by the task scheduler, the feasibility to reuse loop-invariant data from past iterations can be achieved.

More specifically, the HaLoop scheduler works as follows. Upon submitting a job, the master remembers the association between the slave node where processes a specific data partition and the current job. To support this assignment, the master node maintains a hashmap.

At any node, the master association between the slave node and the current job must be stored. The master node checks if the node already has a task that needs to be launched, and thus the three reducers are assigned to the slave node. To support this assignment, the master node maintains a hashmap.

The mapper output key is hashed using this programming interface. Figure 0 is a sample schedule for the join step with iteration locality. Tasks pair.

AddStepInput

AddInvariantTable

This programming interface is sufficient to express a variety of representations are equal. Thanks to the inter-iteration locality offered by the task scheduler, the feasibility to reuse loop-invariant data from past iterations can be achieved.

More specifically, the HaLoop scheduler works as follows. Upon submitting a job, the master remembers the association between the slave node where processes a specific data partition and the current job. To support this assignment, the master node maintains a hashmap.

At any node, the master association between the slave node and the current job must be stored. The master node checks if the node already has a task that needs to be launched, and thus the three reducers are assigned to the slave node. To support this assignment, the master node maintains a hashmap.

The mapper output key is hashed using this programming interface. Figure 0 is a sample schedule for the join step with iteration locality. Tasks pair.

AddStepInput

AddInvariantTable

This programming interface is sufficient to express a variety of representations are equal. Thanks to the inter-iteration locality offered by the task scheduler, the feasibility to reuse loop-invariant data from past iterations can be achieved.

More specifically, the HaLoop scheduler works as follows. Upon submitting a job, the master remembers the association between the slave node where processes a specific data partition and the current job. To support this assignment, the master node maintains a hashmap.

At any node, the master association between the slave node and the current job must be stored. The master node checks if the node already has a task that needs to be launched, and thus the three reducers are assigned to the slave node. To support this assignment, the master node maintains a hashmap.

The mapper output key is hashed using this programming interface. Figure 0 is a sample schedule for the join step with iteration locality. Tasks pair.

AddStepInput

AddInvariantTable

This programming interface is sufficient to express a variety of representations are equal. Thanks to the inter-iteration locality offered by the task scheduler, the feasibility to reuse loop-invariant data from past iterations can be achieved.

More specifically, the HaLoop scheduler works as follows. Upon submitting a job, the master remembers the association between the slave node where processes a specific data partition and the current job. To support this assignment, the master node maintains a hashmap.

At any node, the master association between the slave node and the current job must be stored. The master node checks if the node already has a task that needs to be launched, and thus the three reducers are assigned to the slave node. To support this assignment, the master node maintains a hashmap.
HaLoop: Optimizations

- Loop-aware scheduling
- Caching
  - Reducer input for invariant data
  - Reducer output speeding up convergence checks

Source: Bu et al. (2010) HaLoop: Efficient Iterative Data Processing on Large Clusters. VLDB.
HaLoop: Performance

(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)

Source: Bu et al. (2010) HaLoop: Efficient Iterative Data Processing on Large Clusters. VLDB.
Beyond MapReduce
Pregel: Computational Model

- Based on Bulk Synchronous Parallel (BSP)
  - Computational units encoded in a directed graph
  - Computation proceeds in a series of supersteps
  - Message passing architecture

- Each vertex, at each superstep:
  - Receives messages directed at it from previous superstep
  - Executes a user-defined function (modifying state)
  - Emits messages to other vertices (for the next superstep)

- Termination:
  - A vertex can choose to deactivate itself
  - Is “woken up” if new messages received
  - Computation halts when all vertices are inactive

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.
Pregel

superstep $t$

superstep $t+1$

superstep $t+2$

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.
Pregel: Implementation

- Master-Slave architecture
  - Vertices are hash partitioned (by default) and assigned to workers
  - Everything happens in memory

- Processing cycle:
  - Master tells all workers to advance a single superstep
  - Worker delivers messages from previous superstep, executing vertex computation
  - Messages sent asynchronously (in batches)
  - Worker notifies master of number of active vertices

- Fault tolerance
  - Checkpointing
  - Heartbeat/revert

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.
class PageRankVertex : public Vertex<double, void, double> {
  public:
    virtual void Compute(MessageIterator* msgs) {
      if (superstep() >= 1) {
        double sum = 0;
        for (; !msgs->Done(); msgs->Next())
          sum += msgs->Value();
        *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
      }

      if (superstep() < 30) {
        const int64 n = GetOutEdgeIterator().size();
        SendMessageToAllNeighbors(GetValue() / n);
      } else {
        VoteToHalt();
      }
    }
};

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.
Pregel: SSSP

class ShortestPathVertex : public Vertex<int, int, int> {
    void Compute(MessageIterator* msgs) {
        int mindist = IsSource(vertex_id()) ? 0 : INF;
        for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
        if (mindist < GetValue()) {
            *MutableValue() = mindist;
            OutEdgeIterator iter = GetOutEdgeIterator();
            for (; !iter.Done(); iter.Next())
                SendMessageTo(iter.Target(),
                               mindist + iter.GetValue());
        }
        VoteToHalt();
    }
};

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.
Pregel: Combiners

class MinIntCombiner : public Combiner<int> {
    virtual void Combine(MessageIterator* msgs) {

        int mindist = INF;
        for (; !msgs->Done(); msgs->Next()) {
            mindist = min(mindist, msgs->Value());
            Output("combined_source", mindist);
        }
    }
};

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.
Dataflows

Map\textsubscript{1}

Load Visits

Group by url

Reduce\textsubscript{1}

Foreach url generate count

Load Url Info

Join on url

Reduce\textsubscript{2}

Map\textsubscript{2}

Reduce\textsubscript{3}

Map\textsubscript{3}

Group by category

Foreach category generate top10(urls)

Pig Slides adapted from Olston et al. (SIGMOD 2008)
Dataflows

- Pig targets a dataflow at the Hadoop query execution engine
  - Everything broken down into maps and reduces
- Surely, there’s a richer set of graph operators you can build!
Dryad: Graph Operators

Dryad: Architecture

Dryad: Cool Tricks

- **Channel**: abstraction for vertex-to-vertex communication
  - File
  - TCP pipe
  - Shared memory

- **Runtime graph refinement**
  - Size of input is not known until runtime
  - Automatically rewrite graph based on invariant properties

Dryad: Sample Program

```
GraphBuilder XSet = moduleX^N;
GraphBuilder DSet = moduleD^N;
GraphBuilder MSet = moduleM^(N*4);
GraphBuilder SSet = moduleS^(N*4);
GraphBuilder YSet = moduleY^N;
GraphBuilder HSet = moduleH^1;

GraphBuilder XInputs = (ugriz1 >= XSet) || (neighbor >= XSet);
GraphBuilder YInputs = ugriz2 >= YSet;

GraphBuilder XToY = XSet >= DSet >> MSet >= SSet;
for (i = 0; i < N*4; ++i)
{
    XToY = XToY || (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
}

GraphBuilder YToH = YSet >= HSet;
GraphBuilder HOutputs = HSet >= output;

GraphBuilder final = XInputs || YInputs || XToY || YToH || HOutputs;
```

DryadLINQ

- LINQ = Language INtegrated Query
  - .NET constructs for combining imperative and declarative programming
- Developers write in DryadLINQ
  - Program compiled into computations that run on Dryad

Sound familiar?

Source: Yu et al. (2008) DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. OSDI.
DryadLINQ: Word Count

PartitionedTable<LineRecord> inputTable =
    PartitionedTable.Get<LineRecord>(uri);

IQueryable<string> words = inputTable.SelectMany(x => x.line.Split(' '));
IQueryable<IGrouping<string, string>> groups = words.GroupBy(x => x);
IQueryable<Pair> counts = groups.Select(x => new Pair(x.Key, x.Count()));
IQueryable<Pair> ordered = counts.OrderByDescending(x => x.Count);
IQueryable<Pair> top = ordered.Take(k);

Compare:

a = load 'file.txt' as (text: chararray);
b = foreach a generate flatten(TOKENIZE(text)) as term;
c = group b by term;
d = foreach c generate group as term, COUNT(b) as count;
store d into 'cnt';

Compare and contrast...
Storm: Real-Time Data Processing
Storm

- Provides an abstraction for real-time computation:
  - Spouts represent (infinite) source of tuples
  - Bolts consume tuples and emit new tuples
  - A topology determines how spouts and bolts are connected

- Runtime handles:
  - Process management
  - Guaranteed message delivery
  - Fault-tolerance
Hadoop Cluster Architecture

- namenode
  - namenode daemon

- job submission node
  - jobtracker

- tasktracker
  - datanode daemon
  - Linux file system

- slave node
  - datanode daemon
  - Linux file system
YARN

- Hadoop limitations:
  - Can only run MapReduce
  - What if we want to run other distributed frameworks?

- YARN = Yet-Another-Resource-Negotiator
  - Provides API to develop any generic distribution application
  - Handles scheduling and resource request
  - MapReduce (MR2) is one such application in YARN
YARN: Architecture
Today’s Agenda

- Making Hadoop more efficient
- Tweaking the MapReduce programming model
- Beyond MapReduce
Questions?