Extreme Computing

Beyond MapReduce
Today’s agenda

• Making Hadoop more efficient
• Tweaking the MapReduce programming model
• Beyond MapReduce
MORE EXPRESSIVE PROCESSING USING MAPREDUCE
We’ve seen this before

- 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
Hadoop vs. RDBMS: grep

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

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;
Hadoop vs. RDBMS: aggregation

```
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. RDBMS: join

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.
Why?

• Schemas are a good idea
  – Parsing fields out of flat text files is slow
  – Schemas define a contract, decoupling logical from physical
• Schemas allow for building efficient auxiliary structures
  – Value indexes, join indexes, etc.
• Relational algorithms have been optimised for the underlying system
  – The system itself has complete control of performance-critical decisions
  – Storage layout, choice of algorithm, order of execution, etc.
Alleviating schema absence: thrift

• Originally developed by Facebook, now an Apache project
• Provides a Data Definition Language (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 Remote Procedure Call (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;
}
Storage layout: row vs. column stores

Row store

Column store
Storage layout: 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
Advantages of column stores

• Read efficiency
  – If only need to access a few columns, no need to drag around the rest of the values

• Better compression
  – Repeated values appear more frequently in a column than repeated rows appear

• Vectorised processing
  – Leveraging CPU architecture-level support

• Opportunities to operate directly on compressed data
  – For instance, when evaluating a selection; or when projecting a column
Why not in Hadoop?

RCFile layout to store a table: example shown in Figure 3, RCFile has the following data

- **A. Data Layout and Compression**
  - RCFile can exploit a column-wise data compression and skip unnecessary column reads.
  - Second, as column-store, in the same row are located in the same node, thus it has low cost of tuple reconstruction.
  - First, as row-store, RCFile guarantees that all the fields in the same column are continuously stored in this section, 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.

- **B. Data Appending**
  - According to the HDFS structure, a table can have multiple HDFS blocks.
  - In each HDFS block, RCFile organizes records with a row group size. Depending on the row group size and the HDFS block size, an HDFS block can have only one or multiple row groups. For a table, all row groups have the same size.
  - RCFile creates and maintains an in-memory data writes to the end of a file. The method of data appending RCFile because the underlying HDFS currently only supports only an appending interface is provided for data writing in HDFS. Though currently RCFile uses the same algorithm for all columns in the table data section, it allows us to use different algorithms. For example, the RLE algorithm is not used since it uses the heavy-weight Gzip algorithm in order to get better compression ratios than other light-weight algorithms. For example, the RLE algorithm can find long runs of repeated data values, but the column data is not already sorted. In addition, due to the limitation would not help our goal of fast query processing for a huge amount of disk scans on massively growing data sets.
  - Limited by the page-level data manipulation inside a processing system, such as Hive. RCFile applies the concept of a flat table, which is called a RCF file (Record Columnar File) from a PAX. It combines the advantages of both row-store and column-store. First, as row-store, RCFile guarantees that all the fields in the same column are continuously stored in this section, the RLE 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 it uses the heavy-weight Gzip algorithm in order to get better compression ratios than other light-weight algorithms.
  - First, for the whole metadata header section, RCFile uses the best compression algorithm for each column according to its data type and data distribution. If the data is highly-diverse range of data resource types of different sizes in large data processing systems, such as the one in Facebook.
  - No reason why not

Source: He et al. (2011) RCFile: A Fast and Space-Efficient Data Placement Structure in MapReduce-based Warehouse Systems. ICDE.
Some small steps forward

• 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
Digging further into 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: COGROUPing

**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)

\[ X = \text{COGROUP} \ A \ \text{BY} \ f1, \ B \ \text{BY} \ $0; \]

\[
\begin{align*}
(1, \{(1, 2, 3),\{(1, 3)\}) \\
(2, \{\}, \{(2, 4), (2, 7), (2, 9)\}) \\
(4, \{(4, 2, 1), (4, 3, 3)\}, \{(4, 6), (4, 9)\}) \\
(7, \{(7, 2, 5)\}, \{\}) \\
(8, \{(8, 3, 4), (8, 4, 3)\}, \{(8, 9)\})
\end{align*}
\]
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();
Iterative computation

#!/usr/bin/python
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

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.
MapReduce underperforms in iterative algorithms

• Java verbosity
• Hadoop task startup time
• Stragglers
• Needless data shuffling
• Checkpointing at each iteration
HaLoop architecture

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

Application

Map function  Stop condition

Stop?

Yes

Reduce function

No

Job

Job

Map Reduce Map Reduce

Hadoop MapReduce

Source: Bu et al. (2010) HaLoop: Efficient Iterative Data Processing on Large Clusters. VLDB.
HaLoop: loop-aware scheduling

Source: Bu et al. (2010) HaLoop: Efficient Iterative Data Processing on Large Clusters. VLDB.
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.
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.
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.
YARN: Hadoop version 2.0

• 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
Summary

• Making Hadoop more efficient
  – Leveraging lessons learned from database systems, or extending node-level functionality
• Tweaking the MapReduce programming model
  – Higher-level programming
• Beyond MapReduce
  – Catering for different data models and use cases
  – Extending the runtime for generality