Big Data Analytics Systems: What Goes Around Comes Around

Reynold Xin, CS186 guest lecture @ Berkeley
Apr 9, 2015
Who am I?

Co-founder & architect @ Databricks

On-leave from PhD @ Berkeley AMPLab

Current world record holder in 100TB sorting (Daytona GraySort Benchmark)
Transaction Processing (OLTP)

“User A bought item b”

Analytics (OLAP)

“What is revenue each store this year?”
Agenda

What is “Big Data” (BD)?

GFS, MapReduce, Hadoop, Spark

What’s different between BD and DB?

Assumption: you learned about parallel DB already.
What is “Big Data”?
Gartner’s Definition

“Big data” is high-volume, -velocity and -variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making.
3 Vs of Big Data

**Volume**: data size

**Velocity**: rate of data coming in

**Variety (most important V)**: data sources, formats, workloads
“Big Data” can also refer to the tech stack

Many were pioneered by Google
Why didn’t Google just use database systems?
Challenges

Data size growing (volume & velocity)
  – Processing has to scale out over large clusters

Complexity of analysis increasing (variety)
  – Massive ETL (web crawling)
  – Machine learning, graph processing
Examples

Google web index: 10+ PB

Types of data: HTML pages, PDFs, images, videos, ...

Cost of 1 TB of disk: $50

Time to read 1 TB from disk: 6 hours (50 MB/s)
The Big Data Problem

Semi-/Un-structured data doesn’t fit well with databases

Single machine can no longer process or even store all the data!

Only solution is to **distribute** general storage & processing over clusters.
The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google’s data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB
GFS Assumptions

“Component failures are the norm rather than the exception”

“Files are huge by traditional standards”

“Most files are mutated by appending new data rather than overwriting existing data”

- GFS paper
Files are composed of set of blocks
- Typically 64MB in size
- Each block is stored as a separate file in the local file system (e.g. NTFS)
**Block Placement**

**Example:**

- **Node 1**
  - Block 1
  - Block 3

- **Node 2**
  - Block 2
  - Block 1

- **Node 3**
  - Block 3
  - Block 2

- **Node 4**
  - Block 2
  - Block 3

- **Node 5**
  - Block 2

**e.g., Replication factor = 3**

**Default placement policy:**
- First copy is written to the node creating the file.
- Second copy is written to a data node within the same rack to minimize cross-rack network traffic.
- Third copy is written to a data node in a different rack to tolerate switch failures.

**Objectives:** load balancing, fast access, fault tolerance
GFS Architecture

NameNode  BackupNode

namespace backups

(heartbeat, balancing, replication, etc.)
Failure types:
- Disk errors and failures
- DataNode failures
- Switch/Rack failures
- NameNode failures
- Datacenter failures

GFS paper: “Component failures are the norm rather than the exception.”
GFS Summary

Store large, immutable (append-only) files

Scalability

Reliability

Availability
Google Datacenter

How do we program this thing?
Traditional Network Programming

Message-passing between nodes (MPI, RPC, etc)

Really hard to do at scale:

- How to split problem across nodes?
  - Important to consider network and data locality
- How to deal with failures?
  - If a typical server fails every 3 years, a 10,000-node cluster sees 10 faults/day!
- Even without failures: stragglers (a node is slow)

Almost nobody does this!
Data-Parallel Models

Restrict the programming interface so that the system can do more automatically

“Here’s an operation, run it on all of the data”
  – I don’t care where it runs (you schedule that)
  – In fact, feel free to run it twice on different nodes

Does this sound familiar?
MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to each key. This is exactly the form of a map followed by a reduce, which we call a map-reduce operation.

Given a large data set and a map-reduce computation to perform, the MapReduce system takes care of all the details of running the computation on a cluster of commodity machines. The programmer need only provide the functions to map and reduce, the data to be processed, and the name of the machine on which to run the job. The MapReduce system will automatically partition the input data, run the map function on each partition, merge the intermediate results, and then run the reduce function on the merged results.

The key idea is to provide a high-level abstraction that hides the details of parallelization, fault-tolerance, and load balancing. This allows programmers to focus on the problem at hand, rather than the details of how to implement a parallel program on a commodity cluster.

Given the simplicity of the programming model and the high-level abstraction provided by the MapReduce system, we have found it to be a powerful and easy-to-use tool for processing large data sets. We have used MapReduce to implement a wide variety of applications, including web search, data mining, and data analysis. In all cases, we have found that MapReduce is able to efficiently and effectively process large data sets on commodity clusters.
MapReduce

First widely popular programming model for data-intensive apps on clusters

Published by Google in 2004

- Processes 20 PB of data / day

Popularized by open-source Hadoop project
MapReduce Programming Model

Data type: key-value records

Map function:

\[(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})\]

Reduce function:

\[(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})\]
Hello World of Big Data: Word Count

Input
the quick brown fox
the fox ate the mouse
how now brown cow

Map
the, 1
brown, 1
fox, 1

Shuffle & Sort
the, 1
fox, 1
the, 1

Reduce
brown, 2
fox, 2
how, 1
now, 1
the, 3

Output
ate, 1
cow, 1
mouse, 1
quick, 1
MapReduce Execution

Automatically split work into many small tasks

Send map tasks to nodes based on data locality

Load-balance dynamically as tasks finish
MapReduce Fault Recovery

If a task fails, re-run it and re-fetch its input
  – Requirement: input is immutable

If a node fails, re-run its map tasks on others
  – Requirement: task result is deterministic & side effect is idempotent

If a task is slow, launch 2nd copy on other node
  – Requirement: same as above
MapReduce Summary

By providing a data-parallel model, MapReduce greatly simplified cluster computing:

- Automatic division of job into tasks
- Locality-aware scheduling
- Load balancing
- Recovery from failures & stragglers

Also flexible enough to model a lot of workloads…
Hadoop

Open-sourced by Yahoo!
   – modeled after the two Google papers

Two components:
   – Storage: Hadoop Distributed File System (HDFS)
   – Compute: Hadoop MapReduce

Sometimes synonymous with Big Data
MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)

[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here with a short essay to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm suggests constructing a data center by lining up a large number of processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to test a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce model.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a new data-intensive application. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the extent that MapReduce offers some benefits over conventional database systems, we have two quick observations:

1. A giant step backward in the programming paradigm for large-scale data intensive applications

2. A sub-optimal implementation, in that it uses brute force instead of indexing

3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago

4. Missing most of the features that are routinely included in current DBMS
Why didn’t Google just use databases?

Cost
  – database vendors charge by $/TB or $/core

Scale
  – no database systems at the time had been demonstrated to work at that scale (# machines or data size)

Data Model
  – A lot of semi-/un-structured data: web pages, images, videos

Compute Model
  – SQL not expressive (or “simple”) enough for many Google tasks (e.g. crawl the web, build inverted index, log analysis on unstructured data)

Not-invented-here
MapReduce Programmability

Most real applications require multiple MR steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. count clicks & top K): 2 – 5 steps
- Iterative algorithms (e.g. PageRank): 10’s of steps

Multi-step jobs create spaghetti code

- 21 MR steps -> 21 mapper and reducer classes
- Lots of boilerplate code per step
Higher Level Frameworks

SELECT count(*) FROM users

In reality, 90+% of MR jobs are generated by Hive SQL

A = load 'foo';
B = group A all;
C = foreach B generate COUNT(A);
SQL on Hadoop (Hive)
Problems with MapReduce

1. Programmability
   - We covered this earlier …

2. Performance
   - Each MR job writes all output to disk
   - Lack of more primitives such as data broadcast
Spark

Started in Berkeley AMPLab in 2010; addresses MR problems.

Programmability: DSL in Scala / Java / Python
- Functional transformations on collections
- 5 – 10X less code than MR
- Interactive use from Scala / Python REPL
- You can unit test Spark programs!

Performance:
- General DAG of tasks (i.e. multi-stage MR)
- Richer primitives: in-memory cache, torrent broadcast, etc
- Can run 10 – 100X faster than MR
Full Google WordCount:

```cpp
#include "mapreduce/mapreduce.h"

// User's map function
class SplitWords: public Mapper {
public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while (i < n && isspace(text[i])) i++;
      // Find word end
      int start = i;
      while (i < n && !isspace(text[i])) i++;
      if (start < i)
        Emit(text.substr(start, i - start), "1");
    }
  }
};
REGISTER_MAPPER(SplitWords);

// User's reduce function
class Sum: public Reducer {
public:
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }
    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCTER(Sum);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
  MapReduceSpecification spec;
  for (int i = 1; i < argc; i++) {
    MapReduceInput* in = spec.add_input();
    in->set_format("text");
    in->set_filepattern(argv[i]);
    in->set_mapper_class("SplitWords");
  }
  // Specify the output files
  MapReduceOutput* out = spec.output();
  out->set_format("text");
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_reducer_class("Sum");
  // Do partial sums within map
  out->set_combiner_class("Sum");
  // Tuning parameters
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);
  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();
  return 0;
}
```

Programmability
Spark WordCount:

```scala
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

counts.save("out.txt")
```
Performance

K-Means Clustering

- Hadoop MR: 155 seconds
- Spark: 4.1 seconds

Logistic Regression

- Hadoop MR: 110 seconds
- Spark: 0.96 seconds

Time per Iteration (s)
Performance
Time to sort 100TB

2013 Record: Hadoop
2100 machines
72 minutes

2014 Record: Spark
207 machines
23 minutes

Also sorted 1PB in 4 hours

Source: Daytona GraySort benchmark, sortbenchmark.org
Spark Ecosystem

- BlinkDB (Approximate SQL)
- Spark SQL
- Spark Streaming (Streaming)
- MLlib (Machine Learning)
- GraphX (Graph Computation)
- Spark R (R on Spark)

Spark Core Engine
Spark Summary

Spark generalizes MapReduce to provide:

- High performance
- Better programmability
- (consequently) a unified engine

The most active open source data project
Note: not a scientific comparison.
Beyond Hadoop Users

Spark early adopters

Users
Understands MapReduce & functional APIs

Data Engineers
Data Scientists
Statisticians
R users
PyData …
```scala
pdata.map(lambda x: (x.dept, [x.age, 1])) \
 .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
 .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
 .collect()

data.groupBy("dept").avg("age")
```
DataFrames in Spark

Distributed collection of data grouped into named columns (i.e. RDD with schema)

DSL designed for common tasks

- Metadata
- Sampling
- Project, filter, aggregation, join, ...
- UDFs

Available in Python, Scala, Java, and R (via SparkR)
DataFrames and SQL share the same optimization/execution pipeline

Maximize code reuse & share optimization efforts
Our Experience So Far

SQL is wildly popular and important
- 100% of Databricks customers use some SQL

Schema is very useful
- Most data pipelines, even the ones that start with unstructured data, end up having some implicit structure
- Key-value too limited
- That said, semi-/un-structured support is paramount

Separation of logical vs physical plan
- Important for performance optimizations (e.g. join selection)
Return of SQL

SQL is what’s next for Hadoop: Here’s who’s doing it

by Declan Hirst  Feb. 21, 2018 - 10:29 AM PDT

When we first began putting together the schedule for Structure: Data several months ago, we knew that running SQL queries on Hadoop would be a big deal — we just didn’t know how big a deal it would actually become. Fast-forward to today, a mere month away from the event (March 20-21 in New York), and the writing on the wall is a lot clearer. SQL support isn’t the end game for Hadoop, but it’s the feature that will...
Dremel: Interactive Analysis of Web-Scale Datasets

Tenzing
A SQL Implementation On The MapReduce Framework

ABSTRACT

Tenzing is a query engine for ad hoc analysis of mostly complete SQL queries combined with very high performance, low latency and structured data, and it is used internally at Google to serve 10,000+ queries per second. In this paper, we present the implementation of Tenzing, a typical analytical query engine.

1. INTRODUCTION

The MapReduce framework has quickly become the most popular distributed data processing framework. However, it was designed primarily for batch processing of large datasets, and does not provide an efficient way to process queries on large datasets. In this paper, we present the implementation of Tenzing, a typical analytical query engine.

ABSTRACT

Column-oriented database systems have been a real game changer for the industry in recent years. Highly tuned and performant systems have evolved that provide users with the possibility of answering ad hoc queries over large datasets in an interactive manner.

In this paper we present the column-oriented datastore developed as part of the central components of PowerDrill. It combines the advantages of columnar data layout with other known techniques (such as using composite range partitions) and extensive algorithmic engineering on key data structures. The main goal of the latter being to reduce the main memory footprint and to increase the efficiency in processing typical user queries. In this combination we achieve large speed-ups. These enable a highly interactive Web UI where it is common that a single mouse click leads to processing a trillion values in the underlying dataset.

1. INTRODUCTION

In the last decade, large companies have been placing an ever-increasing importance on mining their in-house databases; often recognizing them as one of their core assets. With this and with dataset sizes growing at an enormously fast pace, the need for a new generation of data analysis tools has been felt.

Background

Before diving into the subject matter, we give a little background about the PowerDrill system and how it is used for data analysis at Google. Its most visible part is an interactive Web UI making heavy use of AJAX with the help of the Google Web Toolkit [16]. It enables data visualization and exploration in a web browser.
Why SQL?

Almost everybody knows SQL

Easier to write than MR (even Spark) for analytic queries

Lingua franca for data analysis tools (business intelligence, etc)

Schema is useful (key-value is limited)
What’s really different?

SQL on BD (Hadoop/Spark) vs SQL in DB?

Two perspectives:

1. Flexibility in data and compute model

2. Fault-tolerance
Traditional Database Systems (Monolithic)

Applications

SQL

Physical Execution Engine (Dataflow)

Storage Manager

One way (SQL) in/out and data must be structured
Big Data Ecosystems (Layered)

Applications

- SQL
- DataFrame
- M.L.

Data-Parallel Engine (Spark, MR)

General Storage (HDFS, S3, etc)

Decoupled storage, low vs high level compute
Structured, semi-structured, unstructured data
Schema on read, schema on write
Evolution of Database Systems
Decouple Storage from Compute

Traditional

Applications

SQL

Physical Execution Engine (Dataflow)

Storage Manager

2014 - 2015

Applications

SQL

Physical Execution Engine (Dataflow)

General Storage (HDFS)

IBM Big Insight
Oracle
EMC Greenplum
...

support for nested data (e.g. JSON)
Perspective 2: Fault Tolerance

Database systems: coarse-grained fault tolerance
   – If fault happens, fail the query (or rerun from the beginning)

MapReduce: fine-grained fault tolerance
   – Rerun failed tasks, not the entire query
We were writing it to 48,000 hard drives (we did not use the full capacity of these disks, though), and every time we ran our sort, at least one of our disks managed to break (this is not surprising at all given the duration of the test, the number of disks involved, and the expected lifetime of hard disks).
MapReduce
Checkpointing-based Fault Tolerance

Checkpoint all intermediate output
- Replicate them to multiple nodes
- Upon failure, recover from checkpoints
- High cost of fault-tolerance (disk and network I/O)

Necessary for PBs of data on thousands of machines

What if I have 20 nodes and my query takes only 1 min?
Spark Unified Checkpointing and Rerun

Simple idea: remember the lineage to create an RDD, and recompute from last checkpoint.

When fault happens, query still continues.

When faults are rare, no need to checkpoint, i.e. cost of fault-tolerance is low.
What’s Really Different?

Monolithic vs layered storage & compute
- DB becoming more layered
- Although “Big Data” still far more flexible than DB

Fault-tolerance
- DB mostly coarse-grained fault-tolerance, assuming faults are rare
- Big Data mostly fine-grained fault-tolerance, with new strategies in Spark to mitigate faults at low cost
Convergence

DB evolving towards BD
- Decouple storage from compute
- Provide alternative programming models
- Semi-structured data (JSON, XML, etc)

BD evolving towards DB
- Schema beyond key-value
- Separation of logical vs physical plan
- Query optimization
- More optimized storage formats
Thanks & Questions?

Reynold Xin

rxin@databricks.com

@rxin
Acknowledgement

Some slides taken from:

Zaharia. Processing Big Data with Small Programs

Franklin. SQL, NoSQL, NewSQL? CS186 2013