Big Data Processing
(and Friends)

Peter Bailis
Stanford CS245
(with slides from Matei Zaharia + Mu Li)
Previous Outline

• Replication Strategies
• Partitioning Strategies
• AC & 2PC
• CAP
• Why is coordination hard?
• NoSQL
“NoSQL”

• Popular set of databases, largely built by web companies in the 2000s
  • Focus on scale-out and flexible schemas
  • Lots of hype, somewhat dying down
Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT
Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service-oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always small but significant number of server and network components that fail. We have seen (and continue to see) many different kinds of hardware failures, ranging from simple component power-issues to full datacenter outages. The ability to deal with failures is a differentiator for us compared to other cloud computing services. It is also a reason that we move more quickly to produce and refine the services that we offer to our customers. We believe the challenges we face are sufficiently interesting and important to be shared in a high level research document.
Figure 1: Service-oriented architecture of Amazon’s platform

Figure 3: Version evolution of an object over time.
“NoSQL”

• Popular set of databases, largely built by web companies in the 2000s
  • Focus on scale-out and flexible schemas
  • Lots of hype, somewhat dying down
• Amazon’s Dynamo was among the first
• Open source examples: MongoDB, Cassandra, Redis
Example API: MongoDB

```javascript
{
   "_id" : ObjectId("54c955492b7c8eb21818bd09"),
   "address" : {
      "street" : "2 Avenue",
      "zipcode" : "10075",
      "building" : "1480",
      "coord" : [ -73.9557413, 40.7720266 ]
   },
   "borough" : "Manhattan",
   "cuisine" : "Italian",
   "grades" : [
      {
         "date" : ISODate("2014-10-01T00:00:00Z"),
         "grade" : "A",
         "score" : 11
      },
      {
         "date" : ISODate("2015-01-01T00:00:00Z"),
         "grade" : "B",
         "score" : 9
      }
   ],
}

db.restaurants.find( { "borough": "Manhattan" } )

db.restaurants.find( { "address.zipcode": "10075" } )

db.restaurants.update(
   { "name" : "Juni" },
   {
      $set: { "cuisine": "American (New)" },
      $currentDate: { "lastModified": true }
   }
)`
“NoSQL”

• Popular set of databases, largely built by web companies in the 2000s
  • Focus on scale-out and flexible schemas
  • Lots of hype, somewhat dying down
• Amazon’s Dynamo was among the first
• Open source examples: MongoDB, Cassandra, Redis

• Newer: “NewSQL” – next-generation, with txns, sometimes SQL!
  • Spanner, CockroachDB, MemSQL
What couldn’t RDBMSs do well?

• Schema changes were (are?) a pain
  • Hard to add new columns, critical when building new applications quickly
• Auto-partition and re-partition (”shard”)
• Gracefully fail-over during failures
• Multi-partition operations
How much of “NoSQL” et al. is new?

• Basic algorithms for scale-out execution were known in 1980s
• Google’s Spanner: core algorithms published in 1993
• Reality: takes a lot of engineering to get right! (web & cloud drove demand)
  • Hint: adding distribution is much harder than building from the ground up!
How much of “NoSQL” et al. is new?

• Semi-structured data management is hugely useful for developers
  • Web and open source: shift from “DBA-first” to “developer-first” mentality
  • Not always a good thing for a mature products or services needing stability!
• Have less info for query optimization, but... people cost more than compute!
Lessons from “NoSQL”

• Scale drove 2000s technology demands
• Open source enabled adoption of less mature technology, experimentation
• Developers, not DBAs (“DevOps”)
• Exciting time for data infrastructure
Today’s Outline

• NoSQL overview
• Cloud Landscape
• System in focus: Spark
• Scale-out ML systems
Key Technology: The Web

• Application pull: how to make sense of the ‘net?
• Hardware push: commodity clusters
Berkeley Network of Workstations project (‘95)
Led to Inktomi (last lecture!)
Old CW: use mainframes // New CW: cheap, commodity storage!
In Huang!
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.
Google File System Big Ideas

• Store big chunks of data on a big, distributed cluster
• Sounds like a database...?
• Bedrock of Google’s entire data infrastructure
  • Can build a number of higher-level storage engines on top...
  • ...in addition to compute engines...
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.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

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...
Example:
word count!

map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
Key MapReduce Ideas

- Express parallel computation using free functional transformations
  - Can execute map, reduce in parallel
  - Side-effect free? Can restart jobs in event of failure
- Fault tolerant
  - Writes intermediate data to disk
  - Node failure? Recompute from upstream
- No SQL, no planner, no optimizer
  - User specifies number of “workers”
Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
Was MapReduce New?
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 our views on MapReduce. This is a good time to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of "jelly beans" rather than utilizing a much smaller 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 teach students how to program such clusters using a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce framework.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

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
Was MapReduce New?

• Stonebraker and Dewitt:
  • No!
  • Isn’t very flexible; user codes entire query plan
  • Doesn’t use indexes
  • Techniques known for decades
  • Kind of dumb: writes intermediate data to disk
A Comparison of Approaches to Large-Scale Data Analysis

Andrew Pavlo  
Brown University  
pavlo@cs.brown.edu

Erik Paulson  
University of Wisconsin  
epaulson@cs.wisc.edu

Alexander Rasin  
Brown University  
alexr@cs.brown.edu

Daniel J. Abadi  
Yale University  
dna@cs.yale.edu

David J. DeWitt  
Microsoft Inc.  
dewitt@microsoft.com

Samuel Madden  
M.I.T. CSAIL  
maddesen@csail.mit.edu

Michael Stonebraker  
M.I.T. CSAIL  
stonebraker@csail.mit.edu

ABSTRACT

There is currently considerable enthusiasm around the MapReduce (MR) paradigm for large-scale data analysis [17]. Although the basic control flow of this framework has existed in parallel SQL database management systems (DBMS) for over 20 years, some have called MR a dramatically new computing model [8, 17]. In this paper, we describe and compare both paradigms. Furthermore, we evaluate both kinds of systems in terms of performance and development complexity. To this end, we define a benchmark consisting of a collection of tasks that we have run on an open source version of MR as well as on two parallel DBMSs. For each task, we measure each system’s performance for various degrees of parallelism on a cluster of 100 nodes. Our results reveal some interesting trade-offs. Although the process to load data into and tune the execution of parallel DBMSs took much longer than the MR system, the observed performance of these DBMSs was strikingly better. We speculate about the causes of the dramatic performance difference and consider implementation concepts that future systems should take from both kinds of architectures.

Categories and Subject Descriptors

model through which users can express relatively sophisticated distributed programs, leading to significant interest in the educational community. For example, IBM and Google have announced plans to make a 1000 processor MapReduce cluster available to teach students distributed programming.

Given this interest in MapReduce, it is natural to ask “Why not use a parallel DBMS instead?” Parallel database systems (which all share a common architectural design) have been commercially available for nearly two decades, and there are now about a dozen in the marketplace, including Teradata, Aster Data, Netezza, DATALegro (and therefore soon Microsoft SQL Server via Project Madison), Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via the Database Partitioning Feature), and Oracle (via Exadata). They are robust, high performance computing platforms. Like MapReduce, they provide a high-level programming environment and parallelize readily. Though it may seem that MR and parallel databases target different audiences, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates to filter and combine data) or a set of MapReduce jobs. Inspired by this question, our goal is to understand the differences between the MapReduce approach
Was MapReduce New?

• Reality
  • Somewhere in-between
  • Ideas not necessarily new...
    • Dataflow: old idea
    • Map and Reduce: about as old
  • ...but where is fault-tolerant system that can index the internet?
    • Dean and Ghemawat just claim it’s a “useful tool!”
  • ...and what do programmers prefer to use?
Was MapReduce useful?

• Yes!
• 2006: Team at Yahoo! creates Hadoop, open source GFS+MapReduce
• 2008: Hadoop runs on 4000 nodes
• 2009: Hadoop sorts a petabyte of data in < 17 hours
• 2011: Hadoop v1 released...
Hadoop Ecosystem

• Around mid-2000s, open source exploded

• Build versus buy?
  • Many web companies, startups adopted/adapted open source
  • Yahoo!, Facebook, Twitter release, contribute back to open source
  • Apache Software Foundation becomes “home” for Hadoop ecosystem

• Simultaneously:
  • Cloud infrastructure (e.g., AWS) means easier than ever to get cluster
  • Can scale on-demand
Late 2000s: Continued Evolution, Pain Points

• Storage in HDFS
  • Problem: raw files waste space, are inefficient
  • Solution: impose flexible schemas (see: Parquet, RCFile)

• Faster serving from HDFS
  • Problem: flat files are slow to serve
  • Solution: HBase, open source clone of another Google project, called BigTable

• Hadoop is batch-oriented
  • Problem: want faster execution
  • Solution: streaming dataflow engines like Storm

• Hadoop is slow and has awkward APIs
  • Problem: intermediate materialization is slow, APIs are clunky
  • Solution: new interface; Apache Spark!
Today’s Outline

• NoSQL overview
• Cloud Landscape
• System in focus: Spark
• Scale-out ML systems
Matei Zaharia
(cool fact: he’s on our faculty!)
Original Spark Vision

1) Unified engine for big data processing
   • Combines batch, interactive, iterative, streaming

2) Concise, language-integrated API
   • Functional programming in Scala/Java/Python
Motivation: Unification

MapReduce

General batch processing

Pregel
Dremel
Impala
Storm

Specialized systems for new workloads

Giraph
Drill
Presto
S4 . . .

Hard to compose in pipelines
Motivation: Unification

MapReduce → Pregel → Giraph
Dremel → Drill → Presto
Impala → Storm → S4 → Spark

General batch processing → Specialized systems for new workloads → Unified engine
Motivation: Concise API

Much of data analysis is exploratory / interactive

Spark solution: Resilient Distributed Datasets (RDDs)
  • “Distributed collection” abstraction with simple functional API

```scala
lines = spark.textFile("hdfs://...")       // RDD[String]
points = lines.map(line => parsePoint(line)) // RDD[Point]
points.filter(p => p.x > 100).count()
```
Implementation idea

Execution similar to Hadoop: distribute to cluster

Store intermediate data in memory

Recover any failed partitions by re-running functional tasks (Trade-off with Hadoop/MapReduce?)
Hadoop poor fit for iterative ML

(a) Logistic Regression

(b) K-Means
How Did the Vision Hold Up?

Generally well!

Users really appreciate unification

Functional API causes some challenges, work in progress
Libraries Built on Spark

> SQL  Streaming  MLlib  GraphX

Spark Core (RDDs)

Largest integrated standard library for big data
Which Libraries Do People Use?

- Spark SQL: 69%
- DataFrames: 62%
- Spark Streaming: 58%
- MLlib + GraphX: 58%

75% of users use more than one component.
Top Applications

- Business Intelligence: 68%
- Data Warehousing: 52%
- Recommendation: 44%
- Log Processing: 40%
- User-Facing Services: 36%
- Faud Detection / Security: 29%
Main Challenge: Functional API

Looks high-level, but hides many semantics of computation

• Functions are arbitrary blocks of Java bytecode
• Data stored is arbitrary Java objects

Users can mix APIs in suboptimal ways
Example Problem

```scala
pairs = data.map(word => (word, 1))

groups = pairs.groupByKey()

groups.map((k, vs) => (k, vs.sum))
```

Materializes all groups as lists of integers

Then promptly aggregates them
Challenge: Data Representation

Java objects often many times larger than underlying fields

class User(name: String, friends: Array[Int])
User(“Bobby”, Array(1, 2))
DataFrame API

DataFrames hold rows with a known schema and offer relational operations on them through a DSL

```python
c = HiveContext()
users = c.sql("select * from users")
ma_users = users[users.state == "MA"]
ma_users.count()
ma_users.groupBy("name").avg("age")
ma_users.map(lambda row: row.user.toUpper())
```

Expression AST
Execution Steps

SQL -> Data Frames -> Logical Plan

Optimizer

Datasets

Physical Plan

Code Generator

RDDs

Catalog

Data Source API

elasticsearch, cassandra, HBase, HDFS, PostgreSQL, Hive, ...
API Details

Based on **data frame** concept in R, Python

- Spark is the first to make this a declarative API

Integrated with the rest of Spark

- ML library takes DataFrames as input & output
- Easily convert RDDs ↔ DataFrames

Google trends for “data frame”
What DataFrames Enable

1. Compact binary representation
   - Columnar, compressed format for caching; rows for processing

2. Optimization across operators (join reordering, pushdown, etc)

3. Runtime code generation
Performance

Time for aggregation benchmark (s)
Performance

Time for aggregation benchmark (s)
Data Sources

Now that we have an API for structured data, map it to data stores

- Spark apps should be able to migrate across Hive, Cassandra, JSON, ...
- Rich semantics of API allows query pushdown into data sources, something not possible with original Spark

```
users[users.age > 20]
select * from users
```
Data Source API

All data sources provide a schema given a connection string (e.g. JSON file, Hive table name)

Different interfaces for “smarter” federation

• **Table scan**: just read all rows → CSV, JSON
• **Pruned scan**: read specific columns → Cassandra, HBase
• **Filtered scan**: read rows matching an expression → JDBC, Parquet, Hive
Examples

JSON:

```
{  
  "text": "hi",
  "user": {  
    "name": "bob",
    "id": 15
  }
}
```

tweets.json

```
select user.id, text from tweets
```

JDBC:

```
select age from users where lang = "en"
```

Together:

```
select t.text, u.age
from tweets t, users u
where t.user.id = u.id
and u.lang = "en"
```
Hardware Trends

Storage

Network

CPU
## Hardware Trends

<table>
<thead>
<tr>
<th></th>
<th>2010</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>50+MB/s (HDD)</td>
<td>500+MB/s (SSD)</td>
</tr>
<tr>
<td>Network</td>
<td>1Gbps</td>
<td>10Gbps</td>
</tr>
<tr>
<td>CPU</td>
<td>~3GHz</td>
<td>~3GHz</td>
</tr>
</tbody>
</table>
## Hardware Trends

<table>
<thead>
<tr>
<th></th>
<th>2010</th>
<th>2015</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>50+MB/s (HDD)</td>
<td>500+MB/s (SSD)</td>
<td>10X</td>
</tr>
<tr>
<td>Network</td>
<td>1Gbps</td>
<td>10Gbps</td>
<td>10X</td>
</tr>
<tr>
<td>CPU</td>
<td>~3GHz</td>
<td>~3GHz</td>
<td>😞</td>
</tr>
</tbody>
</table>
Project Tungsten

Substantially speed up Spark by optimizing CPU efficiency, via:

1. Runtime code generation
2. Exploiting cache locality
3. Off-heap memory management
Tungsten’s Compact Encoding

(123, “data”, “bricks”)
Runtime Code Generation

DataFrame Code / SQL

`df.where(df("year") > 2015)`

Catalyst Expressions

`Greater Than(year#234, Literal(2015))`

Low-level bytecode

```java
bool filter(Object baseObject) {
    int offset = baseOffset + BitSetWidthInBytes + 3*8L;
    int value = Platform.getInt(baseObject, offset);
    return value34 > 2015;
}
```

JVM intrinsic JIT-ed to pointer arithmetic
Long-Term Vision

First stage out in Spark 1.5
Big Data in Production

Big data is moving from offline analytics to production use

- Incorporate new data in seconds (streaming)
- Power low-latency queries (data serving)

Currently very hard to build: separate streaming, serving & batch systems

Our goal: one engine for “continuous apps”
PB’s Punchlines

- Spark is de facto batch analytics processor today
  - Streaming: just run min—batches…
- Looks a lot like SQL data warehouse…
- …but can do a bunch more, too: ML, etc.
- Maybe the biggest lesson:
  - Building modular software enables modular usages
  - Compare: traditional data warehouses
    - Still slower than fast data warehouse, but more flexible!
- Humans win over hardware efficiency (for many cases)!
Today’s Outline

NoSQL overview
Cloud Landscape
System in focus: Spark
Scale-out ML systems
Machine learning is concerned with systems that can learn from data.
Idea:

- For “big models,” partition data and parameters
  - Called a “parameter server”
  - Asynchronous training can help!
- New systems like TensorFlow combine this idea with dataflow
Data and model partition

Training data
Data and model partition

Model

Server machines

Training data

Worker machines
Data and model partition

Model

Server machines

push

Training data

Worker machines
Data and model partition

Model

Server machines

push

Worker machines

pull

Training data
Example: distributed gradient descent

Server machines

Worker machines
Example: distributed gradient descent

Workers **pull** the working set of **model**

Server machines

Worker machines
Example: distributed gradient descent

Workers **pull** the working set of model
Iterate until stop

Server machines

Workers compute **gradients**

Worker machines
Example: distributed gradient descent

Workers pull the working set of model
Iterate until stop

Workers push gradients

Server machines

Worker machines
Example: distributed gradient descent

Workers **pull** the working set of model.
Iterate until stop.

Workers **compute gradients**.

Workers **push gradients**.

Update model.

Server machines

Worker machines
Example: distributed gradient descent

Workers pull the working set of model
Iterate until stop

Workers compute gradients
Workers push gradients
Update model
Workers pull updated model

Server machines

Worker machines
Costs/Benefits compared to Dataflow?
Costs/Benefits compared to Dataflow?

- Pro: Efficient; only access model parameters you need
- Con: No real “query optimization”
- Pro or Con?: Fault tolerance
- Pro: Allows asynchronous execution
Results for bounded delay

Ad click prediction

sequential

- computing
- waiting

Time (hour):
- 0.9
- 1.8

Bounded delay:
- 0
- 1
- 2
- 4
- 8
- 16
Results for bounded delay

Ad click prediction

sequential

computing

waiting

time (hour)

Bounded delay

0 1 2 4 8 16
Bonus from last time:

Does machine learning always need serializability?

- e.g., say I want to train a deep network on 1000s of GPUs
Bonus from last time:

Does machine learning always need serializability?

• No! Turns out asynchronous execution is \textit{provably} safe (for sufficiently small delays)
• Convex optimization routines (e.g., SGD) run faster on modern HW without locks
• Best paper name ever: HogWild!
TensorFlow: A system for large-scale machine learning

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Mannanath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng

Google Brain

Abstract

TensorFlow is a machine learning system that operates at large scale and in heterogeneous environments. TensorFlow uses dataflow graphs to represent computation, shared state, and the operations that mutate that state. It maps the nodes of a dataflow graph across many machines in a cluster, and within a machine across multiple computational devices, including multicore CPUs, general-purpose GPUs, and custom-designed ASICs known as Tensor Processing Units (TPUs). This architecture gives flexibility to the application developer: whereas in previous "parameter server" designs the management of shared state is built into the system, TensorFlow enables developers to experiment with novel optimizations and training algorithms. TensorFlow supports a variety of applications, with a focus on training and inference on deep neural networks. Several Google services use TensorFlow in production, we have released it as an open-source project, and datasets, and moving them into production. We have based TensorFlow on many years of experience with our first-generation system, DistBelief [20], both simplifying and generalizing it to enable researchers to explore a wider variety of ideas with relative ease. TensorFlow supports both large-scale training and inference: it efficiently uses hundreds of powerful (GPU-enabled) servers for fast training, and it runs trained models for inference in production on various platforms, ranging from large distributed clusters in a datacenter, down to running locally on mobile devices. At the same time, it is flexible enough to support experimentation and research into new machine learning models and system-level optimizations.

TensorFlow uses a unified dataflow graph to represent both the computation in an algorithm and the state on which the algorithm operates. We draw inspiration from the high-level programming models of dataflow systems [2, 21, 34] and the low-level efficiency of parameter servers [14, 20, 49]. Unlike traditional dataflow sys-
Figure 2: A schematic TensorFlow dataflow graph for a training pipeline, containing subgraphs for reading input data, preprocessing, training, and checkpointing state.
Punchlines

• Parameter server architecture is useful for training large models
  • Increasingly popular in “deep networks”

• Lots of noise about new systems (Graphs, Deep Learning)
  • Often need special adaptation for workloads (e.g., special joins, operators)
  • But basic computational patterns (dataflow with some shared state) same

• Asynchrony can help training time in distributed environment
Hidden Technical Debt in Machine Learning Systems

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips
{dsculley, gholt, dgg, edavydov, toddphillips}@google.com
Google, Inc.

Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, Dan Dennison
{ebner, vchaudhary, mwyoung, jfcrespo, dennison}@google.com
Google, Inc.
Next generation of systems:

Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

Post-database data management!!!