Big Data and Distributed Data Processing (Analytics)

Reynold Xin @rxin

2017-12-05, CS145 Guest Lecture, Stanford

databricks™

Who am I?

Co-founder & Chief Architect @ Databricks

- Day job: direction for data processing (including Spark)
- Night job: code contributor to Apache Spark, #1 committer

On-leave from PhD @ Berkeley AMPLab

Transaction Processing	Analytics
(OLTP)	(OLAP)
"User A bought item b"	"What is revenue each store this year?"

Agenda

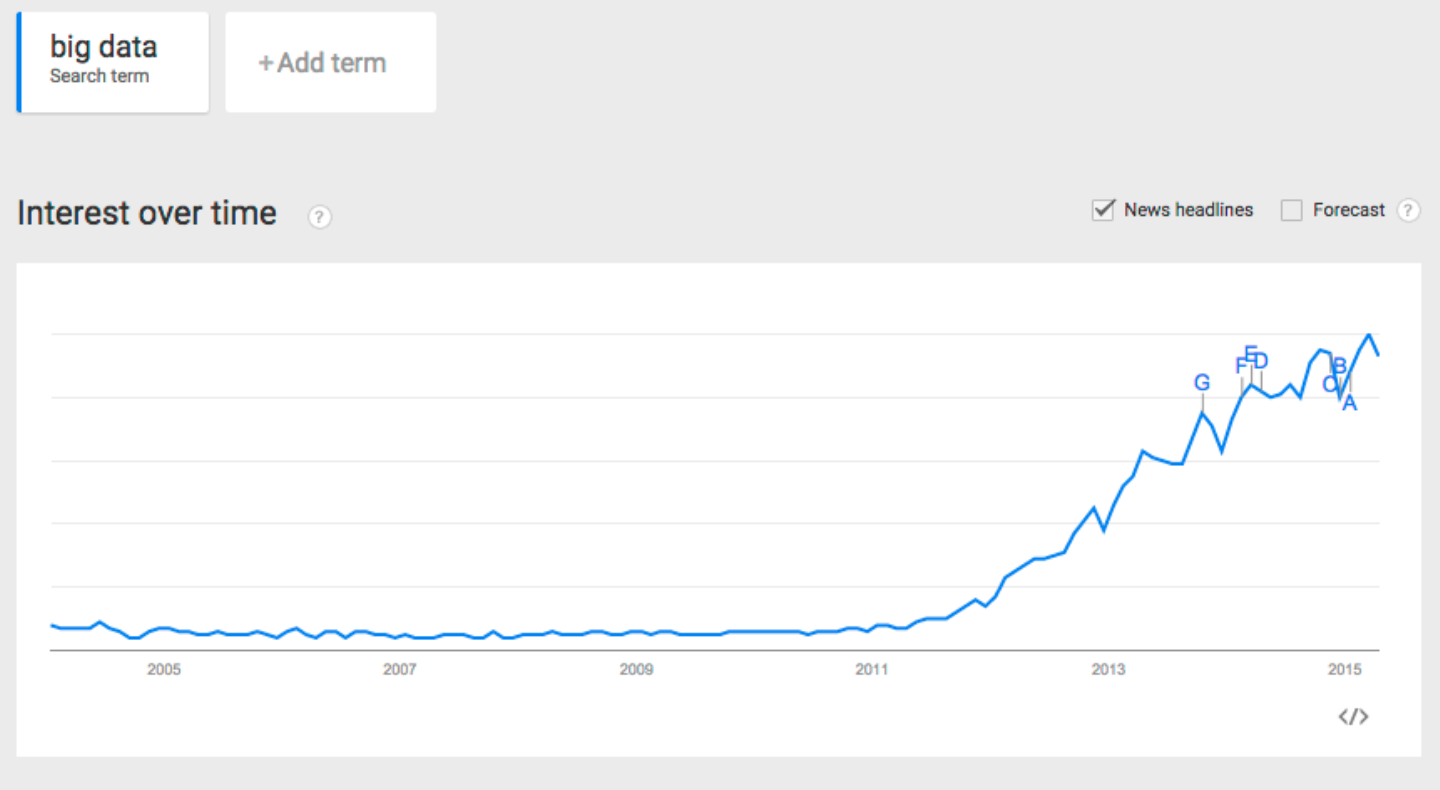
What is “Big Data” (BD)?

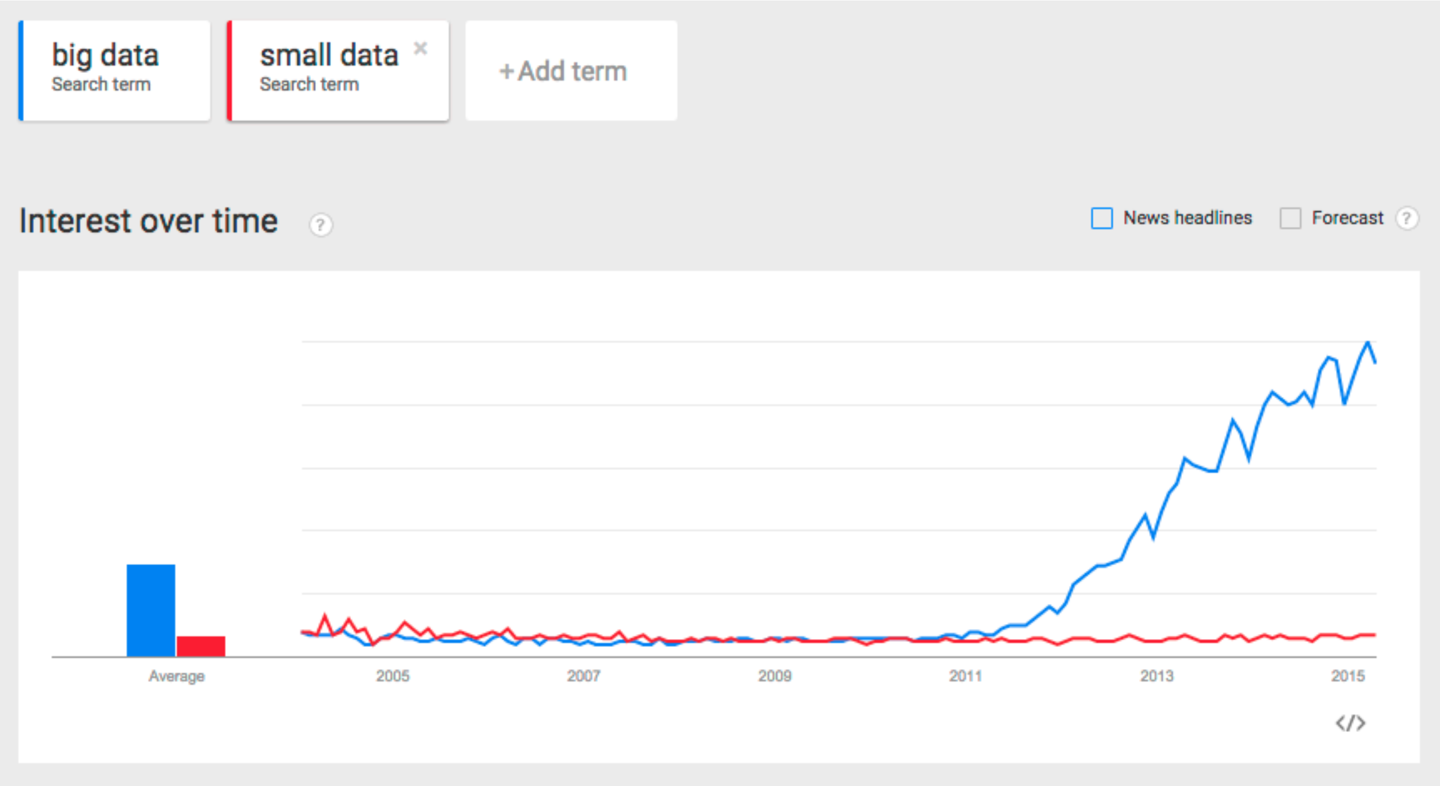
Distributed data processing / MPP databases

GFS, MapReduce, Hadoop

Spark

What’s different between BD and DB?


big data
Search term
+Add term
Interest over time
News headlines
Forecast
G
F
E
D
B
C
A
2005
2007
2009
2011
2013
2015



big data
Search term
small data
Search term
+Add term
Interest over time
News headlines
Forecast
Average
2005
2007
2009
2011
2013
2015

databricks

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

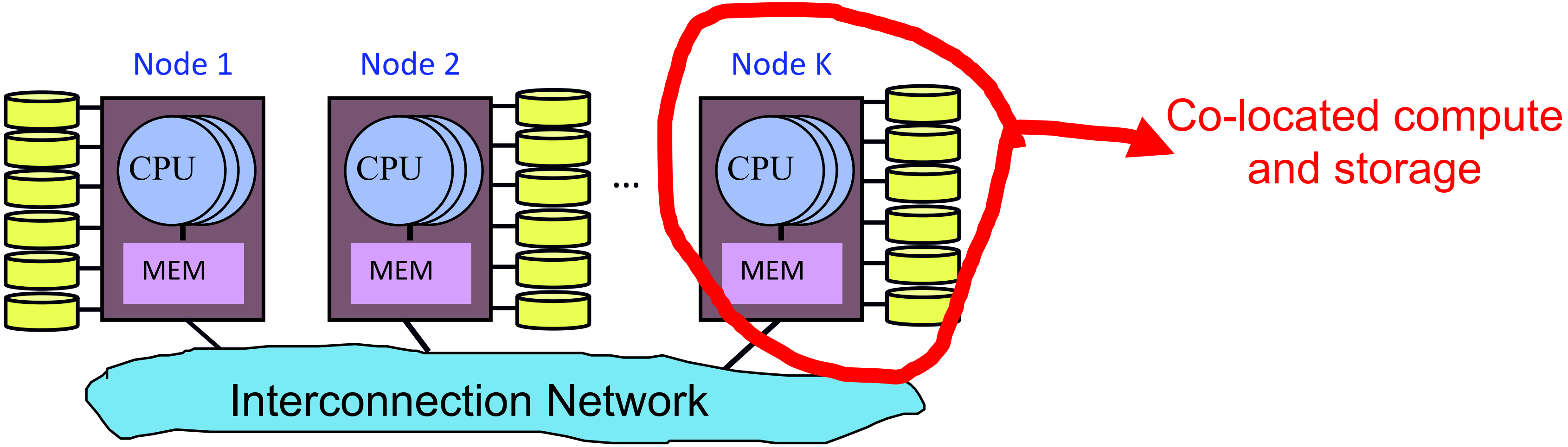
Some concepts pioneered by Google

Massively Parallel Processing Databases (MPP)

Shared nothing architecture

Commodity servers connected via commodity networking

Example: Teradata, Redshift



databricks

* Source: DeWitt, Data Warehousing in the Cloud, The End of "Shared-Nothing"

How does query processing work?

Embarrassingly parallel operators (each node doing their own work):

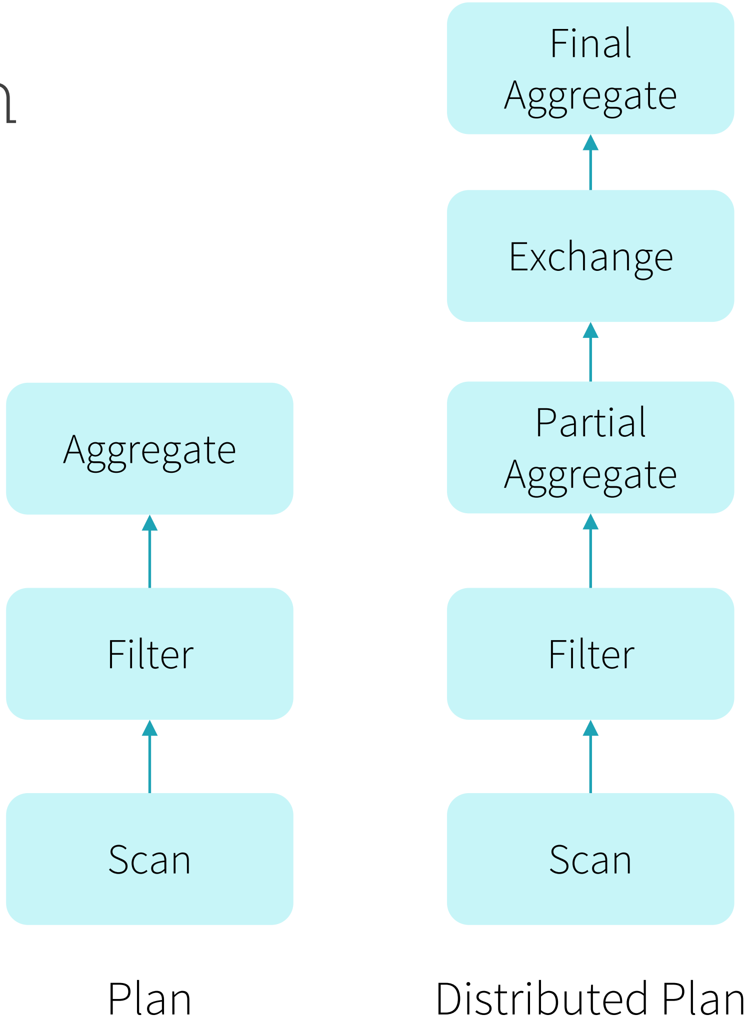
- Scan
- Filter

Aggregate?

Join?

Distributed Aggregation

```
select count(*) from store_sales
where ss_item_sk = 1000
```



Exchange Operator

"Shuffles" data to the right partition (node / thread), hash or range

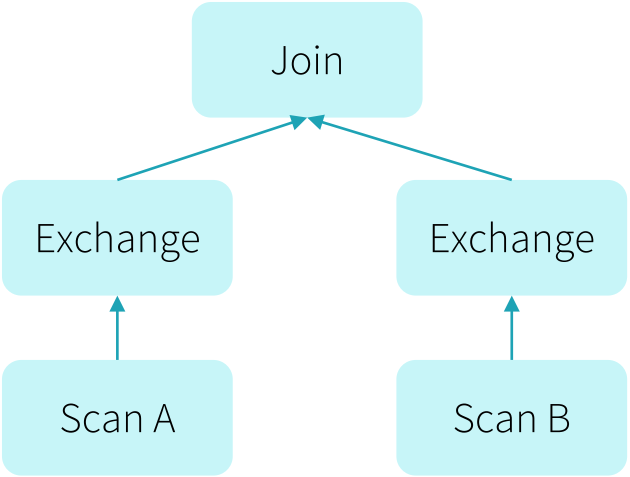
Hash partitioning: partition_id(row) = hash(key(row)) % N

Separation of concerns in distributed query processing

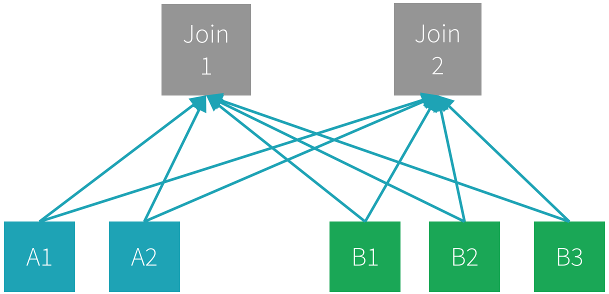
- Other operators are no different from single-threaded implementations (e.g. aggregate, scan, filter)

Distributed Joins - Shuffle Joins

A.k.a. copartitioned join



Plan



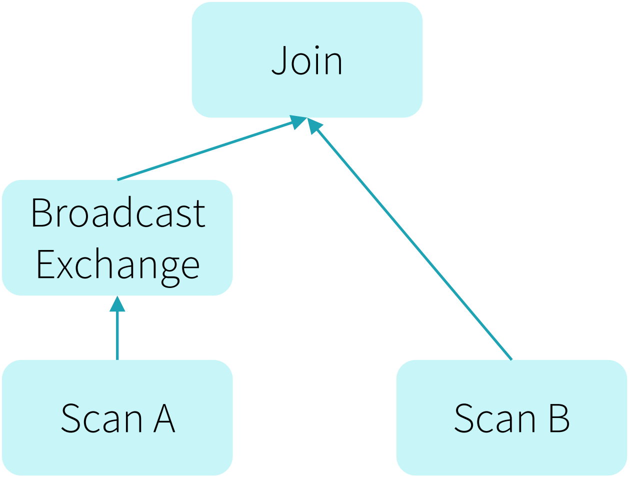
Data Flow

Network traffic: size(A) + size(B)

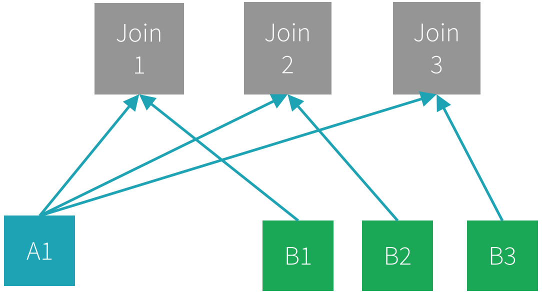
databricks

Distributed Joins – Broadcast Joins

If size(A) << size(B), e.g. A is 1MB, and B is 1TB, can we avoid shuffling all the data?



Plan



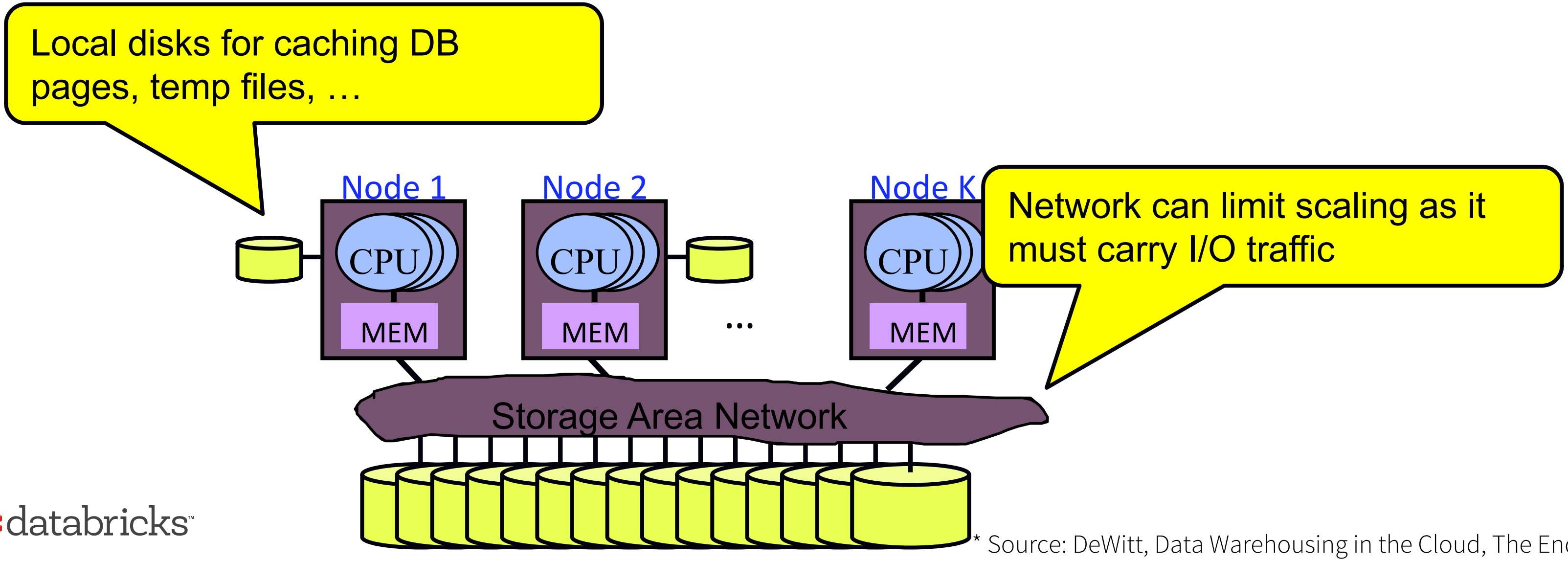
Data Flow

Network traffic: size(A) * N

Shared storage architecture

Can scale compute / storage separately.

"Cloud" model. Examples: Hadoop, Spark.



* Source: DeWitt, Data Warehousing in the Cloud, The End of "Shared-Nothing"

Why didn't Google just use database systems?

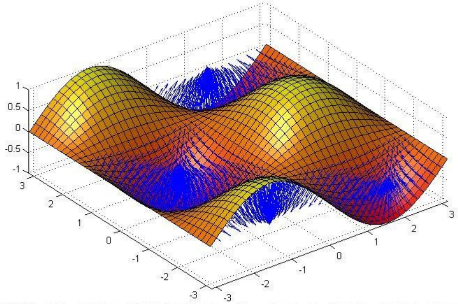
Challenges Google faced

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



MACHINE LEARNING

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

File Splits

Example:

Large File

1100101010011100101010011100101010011100101010011100110010101001110010101001110010101001110010101001110010101001110010101001
1100101010011100101010011100101010011100101010011100110010101001110010101001110010101001110010101001110010101001110010101001
1100101010011100101010011100101010011100101010011100110010101001110010101001110010101001110010101001110010101001110010101001
1100101010011100101010011100101010011100101010011100110010101001110010101001110010101001110010101001110010101001110010101001
1100101010011100101010011100101010011100101010011100110010101001110010101001110010101001110010101001110010101001110010101001
1100101010011100101010011100101010011100101010011100110010101001110010101001110010101001110010101001110010101001110010101001
1100101010011100101010011100101010011100101010011100110010101001110010101001110010101001110010101001110010101001110010101001
1100101010011100101010011100101010011100101010011100110010101001110010101001110010101001110010101001110010101001110010101001
...

6440MB

Let's color-code them

Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	...	Block 100	Block 101
64MB	64MB	64MB	64MB	64MB	64MB		64MB	40MB

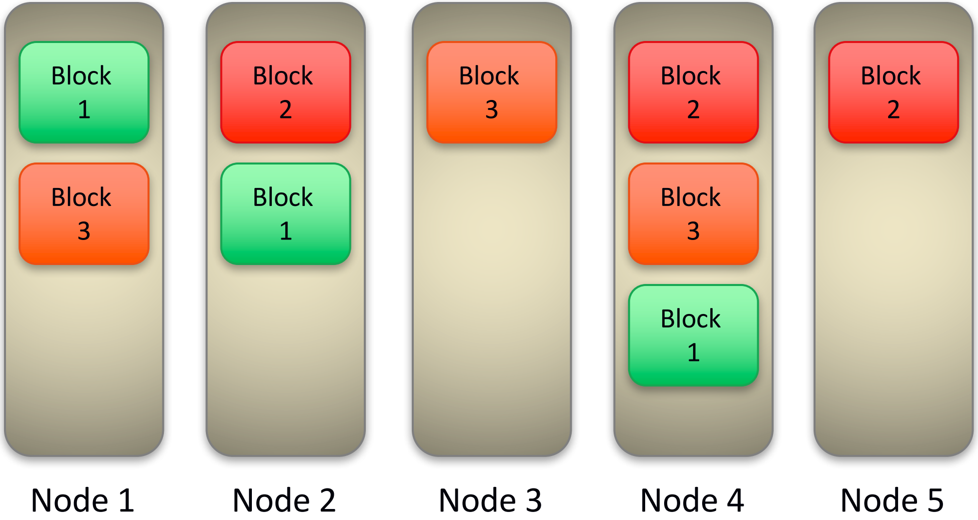
e.g., Block Size = 64MB

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:



e.g., Replication factor = 3

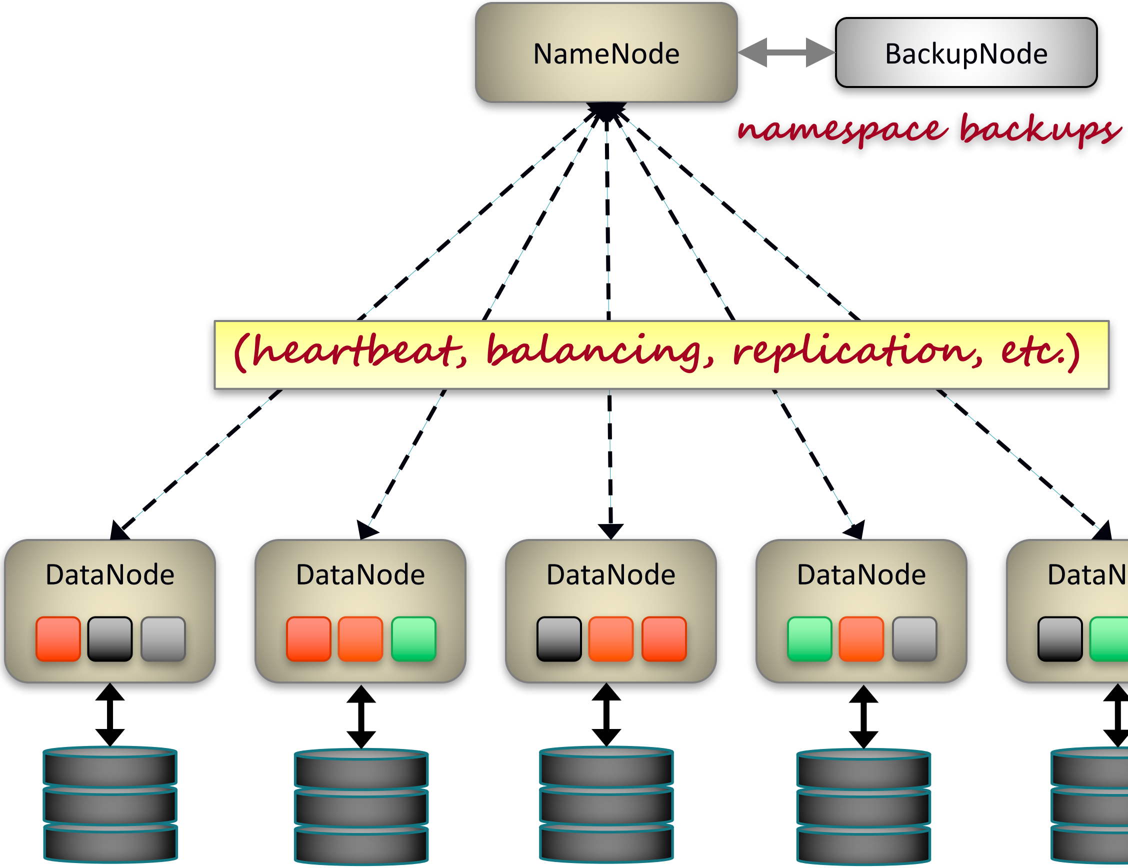
Default placement policy:

- First copy is written to the node creati... ...y)
- Second copy is writt... ...the same rack
  (...ross-rack network traffic)
- ...y is written to a data node in a different rack
  (*to tolerate switch failures*)

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

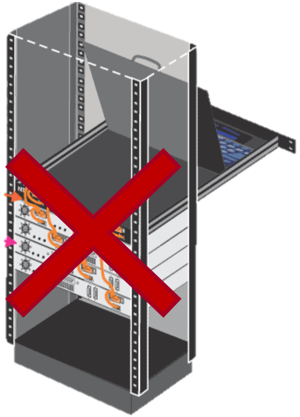
databricks™

GFS Architecture



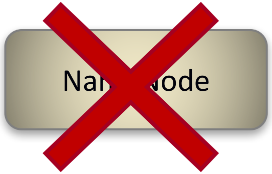
Failures, Failures, Failures

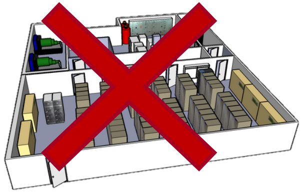
GFS paper: “Component failures are the norm rather than the exception.”



Failure types:

- Disk errors and failures
- DataNode failures
- Switch/Rack failures
- NameNode failures
- Datacenter failures





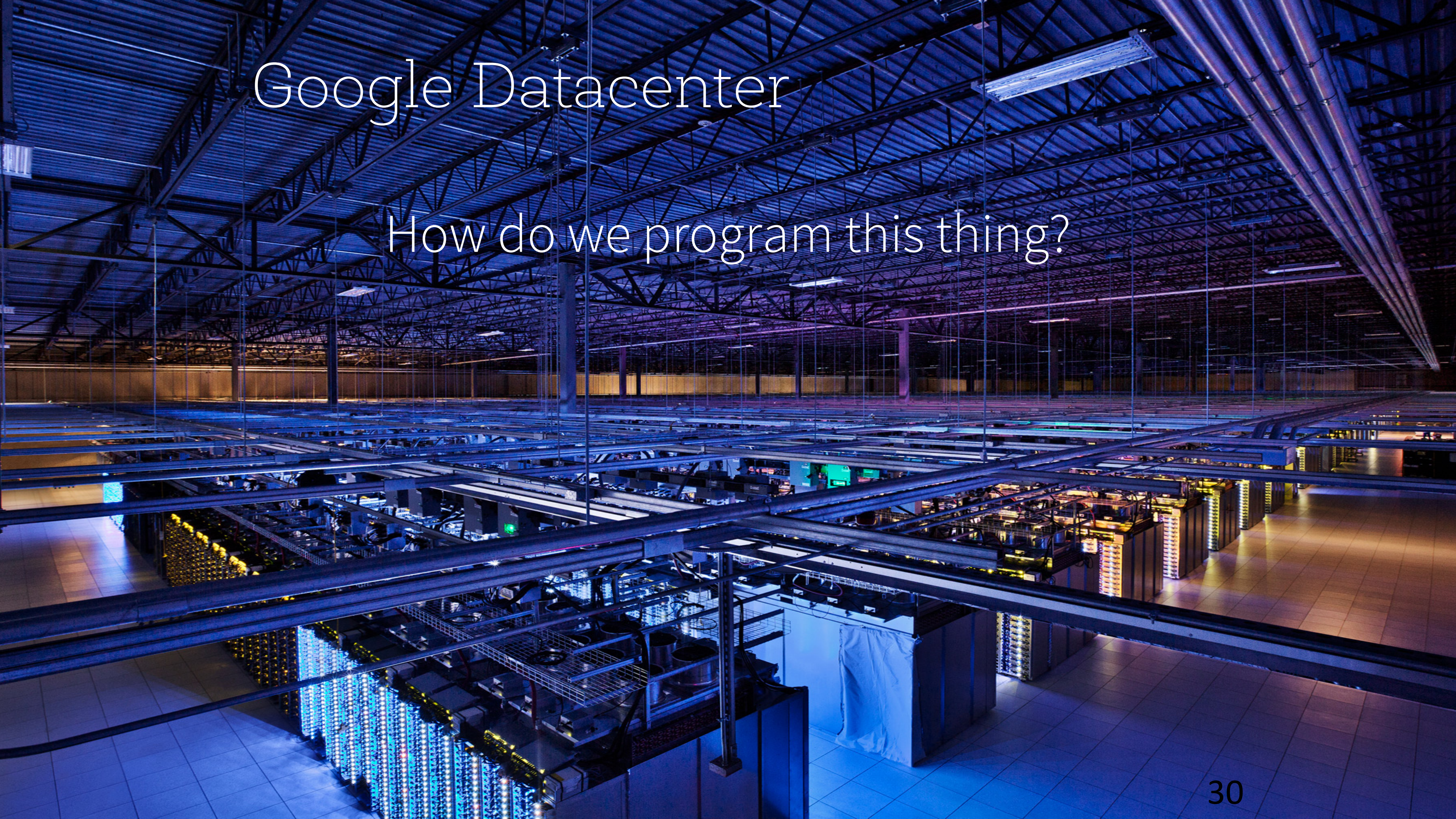
GFS Summary

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
- Simlar to "declarative programming" in databases

MapReduce Programming Model

Data type: key-value *records*

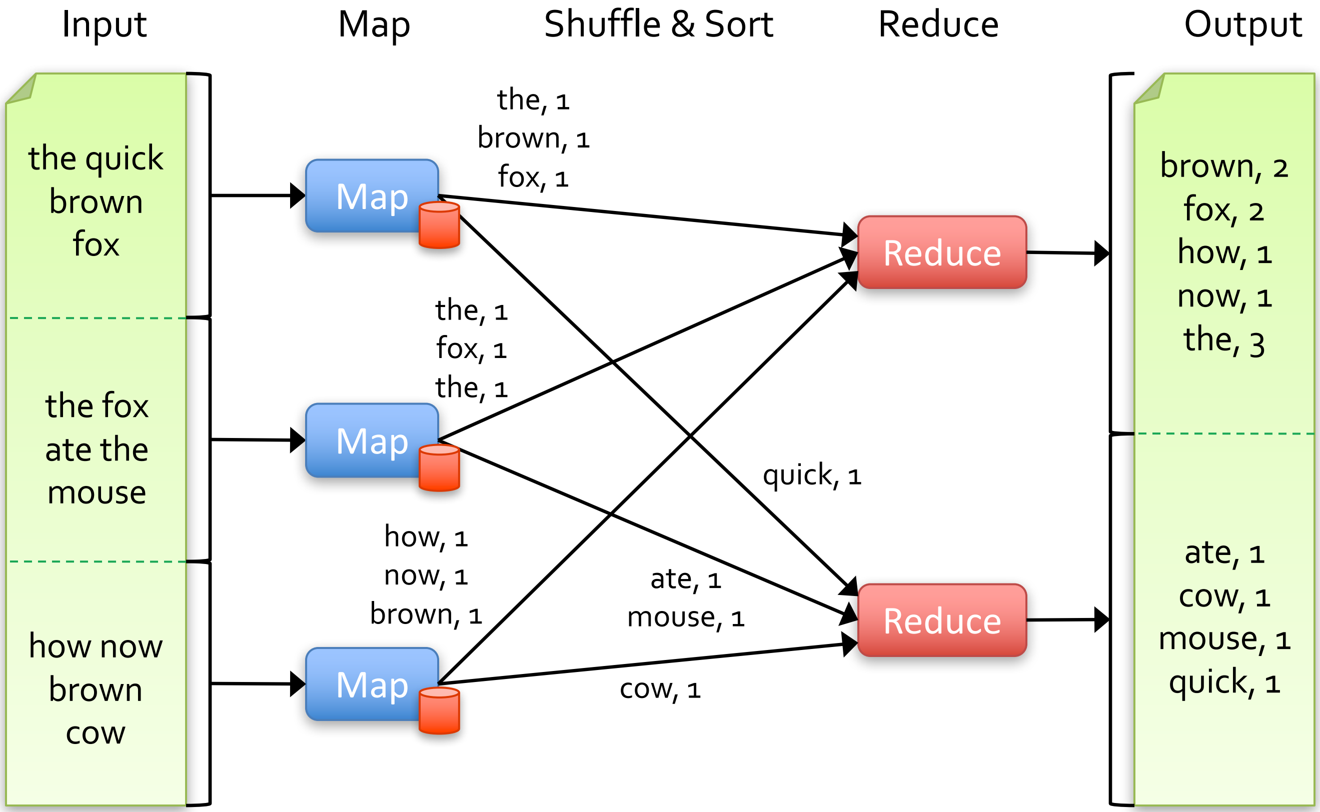
Map function:

$$(K_{in}, V_{in}) \text{ -> } list(K_{inter}, V_{inter})$$

Reduce function:

$$(K_{inter}, list(V_{inter})) \text{ -> } list(K_{out}, V_{out})$$

Hello World of Big Data: Word Count



databricks

MapReduce Execution

Automatically split work into many small tasks

Send map tasks to nodes based on data locality

Load-balance dynamically as tasks finish

Shuffle (remember Exchange?) to handle cross-task communication

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 v
to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigi
processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a larg
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 te
software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the Ma

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represen
data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the

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

Programming 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);
```

Problems with MapReduce

1. Programmability
   - We covered this earlier …

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

Spark

Started in Berkeley in 2010; donated to Apache Software Foundation in 2013

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

Programmability

Full Google WordCount:

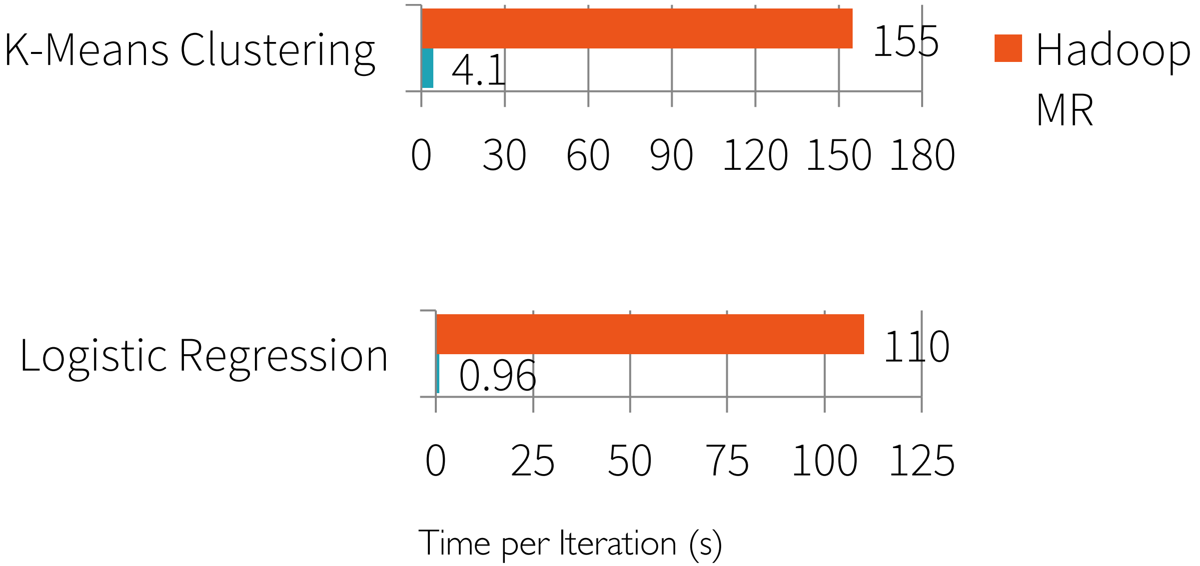
```
#include "mapreduce/mapreduce.h"

// 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_REDUCER(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_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  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

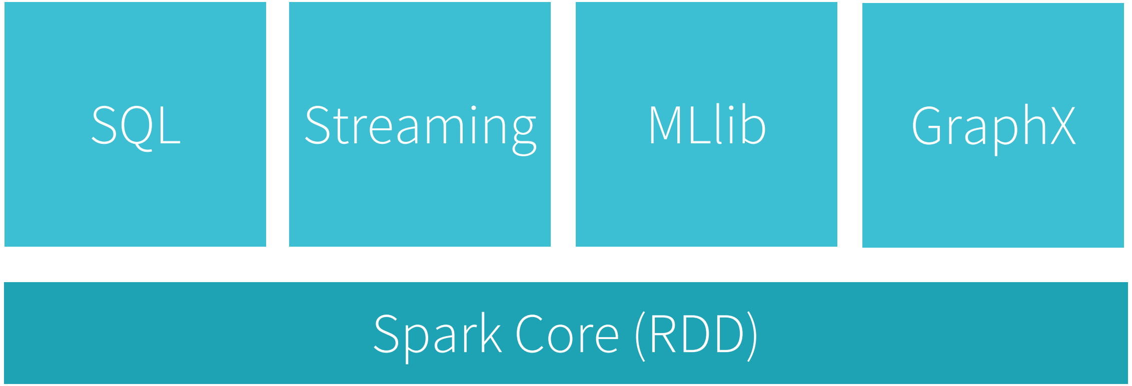
Spark WordCount:

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

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

Performance



Spark stack diagram



Spark Summary

Spark generalizes MapReduce to provide:

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

The most active open source data project with over 1000 contributors

How is Spark different from MPP databases?

Use cases: ETL, log analysis, advanced analytics (beyond SQL)

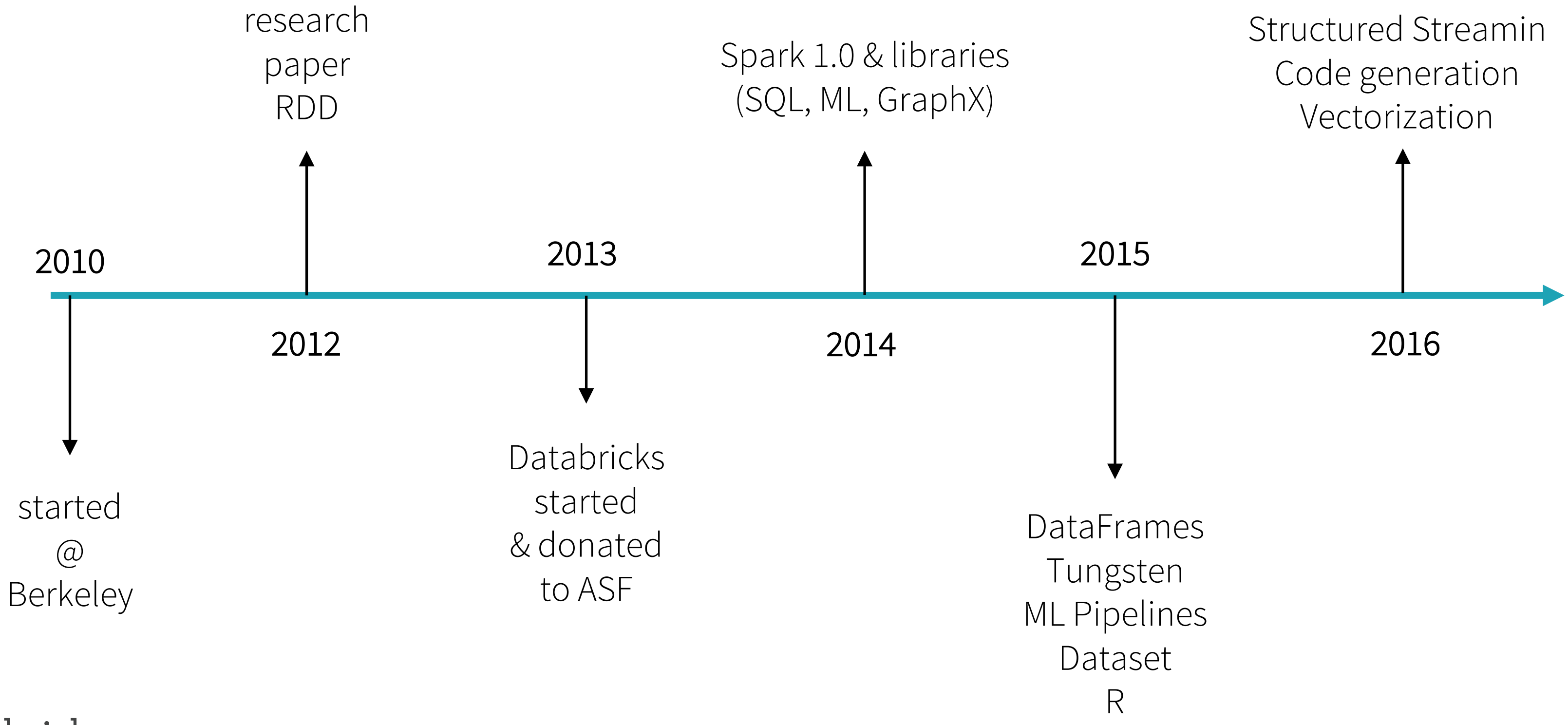
Interfaces: SQL and programmatic access (Scala, Java, Python)

Architecture: “shared nothing” vs “decoupled storage from compute”

“Spark is the Taylor Swift of big data software.”

- Derrick Harris, Fortune

Spark history



databricks

Scaling Spark users

Early adopters

hadoop

Users

Understands
MapReduce
& functional APIs

Data Scientists
Statisticians
R users
PyData
…

```
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")
```

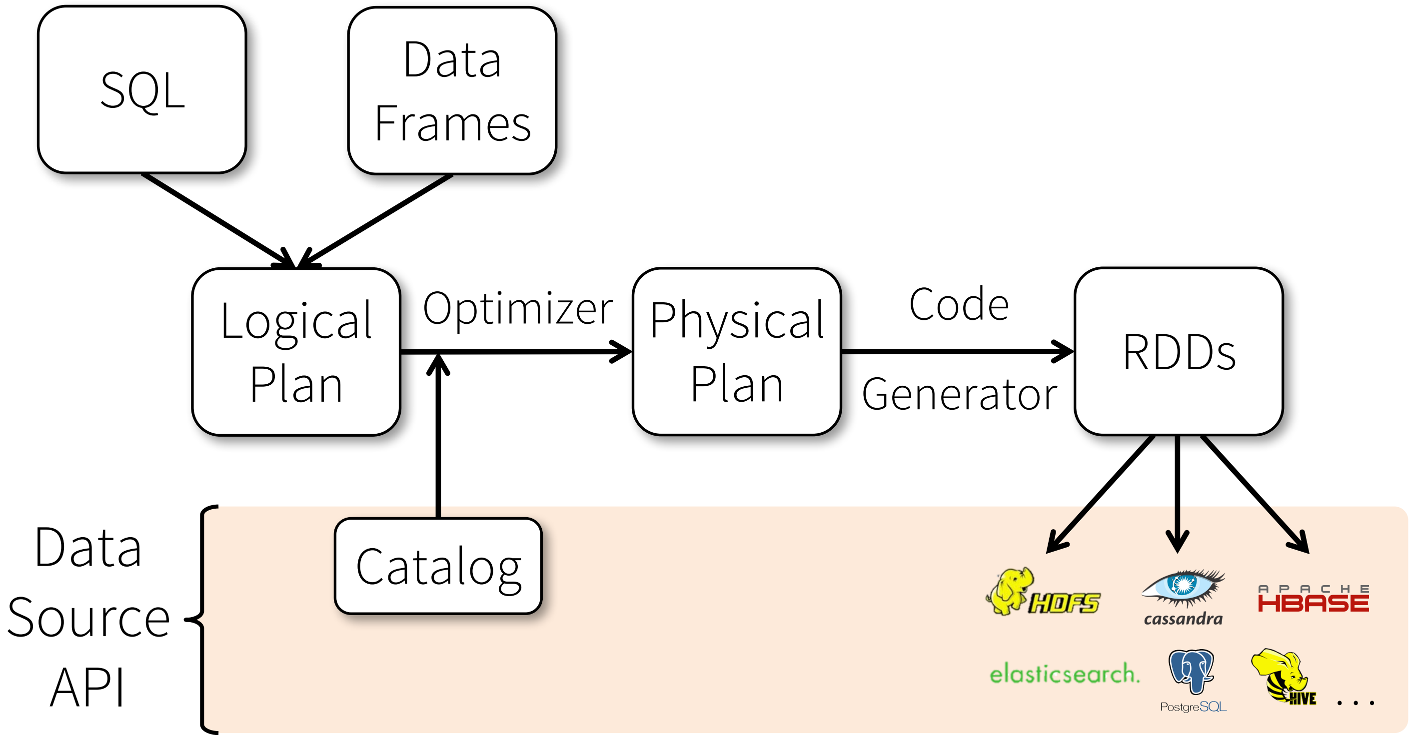
DataFrames in Spark

Distributed data frame abstraction for Java, Python, R, Scala

Similar APIs as single-node tools (Pandas, dplyr), i.e. easy to learn

```
> head(filter(df, df$waiting < 50))  # an example in R
##  eruptions waiting
##1     1.750      47
##2     1.750      47
##3     1.867      48
```

Spark SQL



SQL
Data Frames
Logical Plan
Optimizer
Physical Plan
Code Generator
RDDs
Catalog
Data Source API
HDFS
cassandra
APACHE HBASE
elasticsearch.
PostgreSQL
HIVE
...

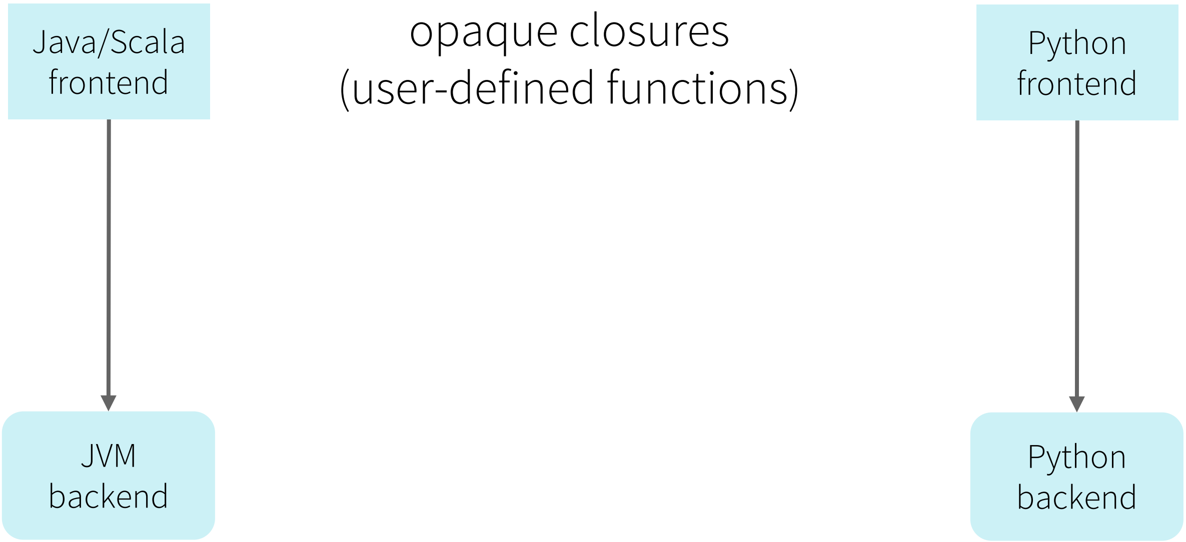
DataFrame API

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

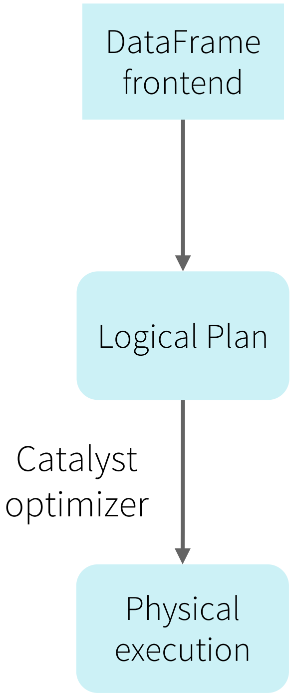
```
val users = spark.sql("select * from users")

val massUsers = users(users("country") === "ES")

massUsers.count()

massUsers.groupBy("name").avg("age")
```

Expression AST

Spark RDD Execution

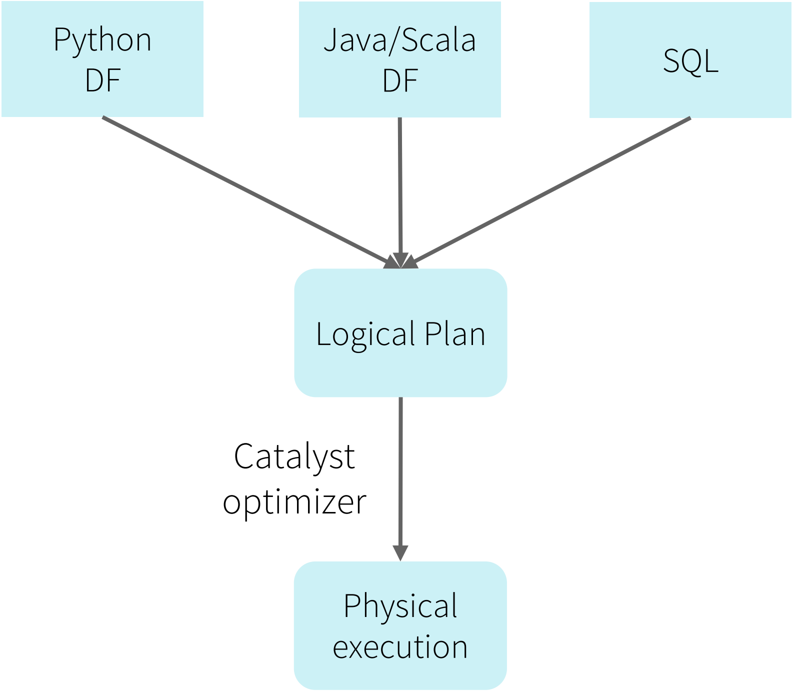


Spark DataFrame Execution



Intermediate representation for computation

Spark DataFrame Execution



Simple wrappers to create logical plan

Intermediate representation for computation

Can we improve Spark performance by an order of magnitude?

databricks™

Performance

How do we get fast distributed query processing?

Fast single-node query processing + fast exchange + good query plans (query optimizations)

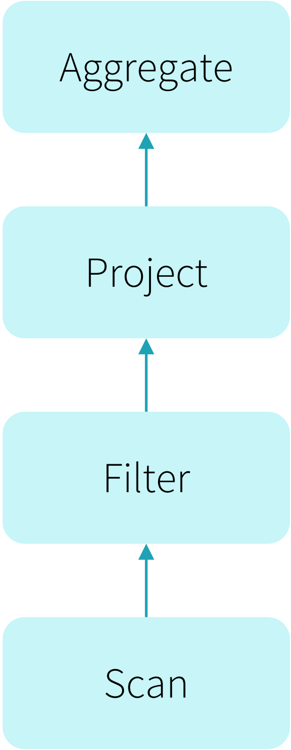
Going back to the fundamentals

Difficult to get order of magnitude performance speed ups with profiling techniques

- For 10x improvement, would need of find top hotspots that add up to 90% and make them instantaneous
- For 100x, 99%

Instead, look bottom up, how fast should *it* run?

```
select count(*) from store_sales
where ss_item_sk = 1000
```



Volcano Iterator Model

Standard for 30 years: almost all databases do it

Each operator is an "iterator" that consumes records from its input operator

```
class Filter {
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }

  def fetch(): InternalRow = {
    child.fetch()
  }
  …
}
```

What if we hire a college freshman to implement this query in Java in 10 mins?

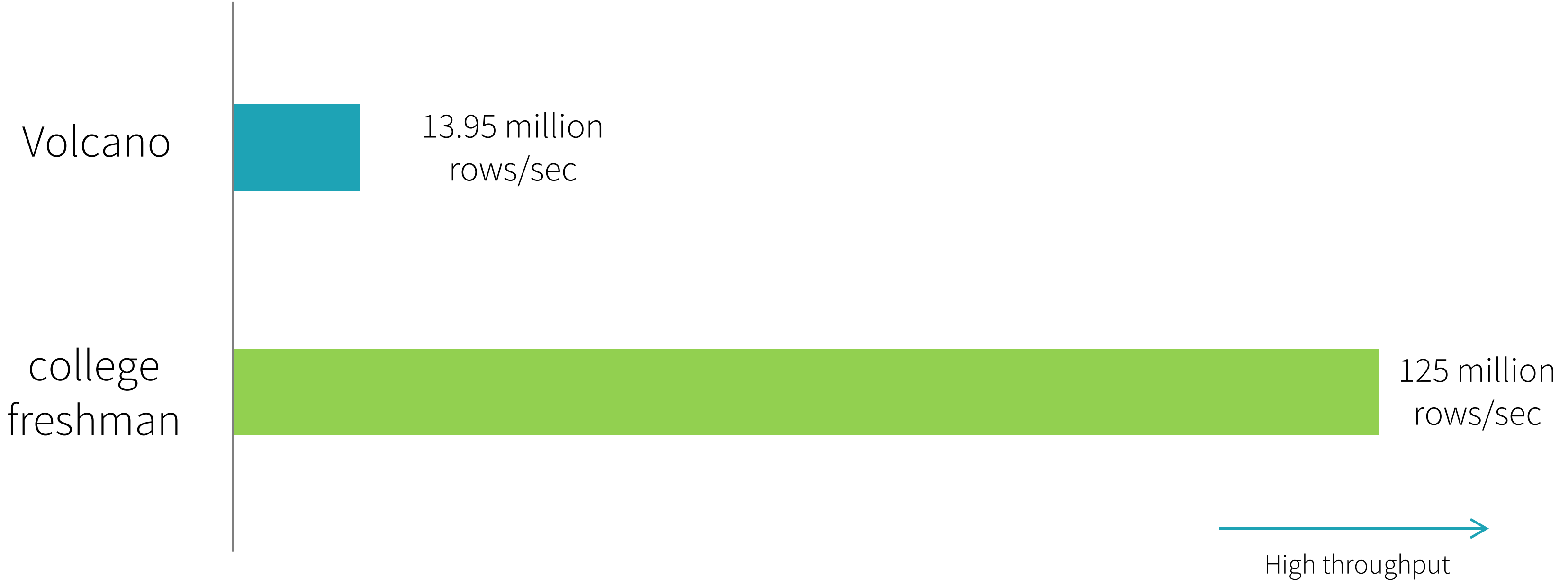
```
select count(*) from store_sales
where ss_item_sk = 1000
```

```
var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

Volcano model
30+ years of database research

vs

college freshman
hand-written code in 10 mins



Note: End-to-end, single thread, single column, and data originated in Parquet on disk

How does a student beat 30 years of research?

Volcano	hand-written code
1. Many virtual function calls	1. No virtual function calls
2. Data in memory (or cache)	2. Data in CPU registers
3. No loop unrolling, SIMD, pipelining	3. Compiler loop unrolling, SIMD, pipelining

Take advantage of all the information that is known **after** query compilation

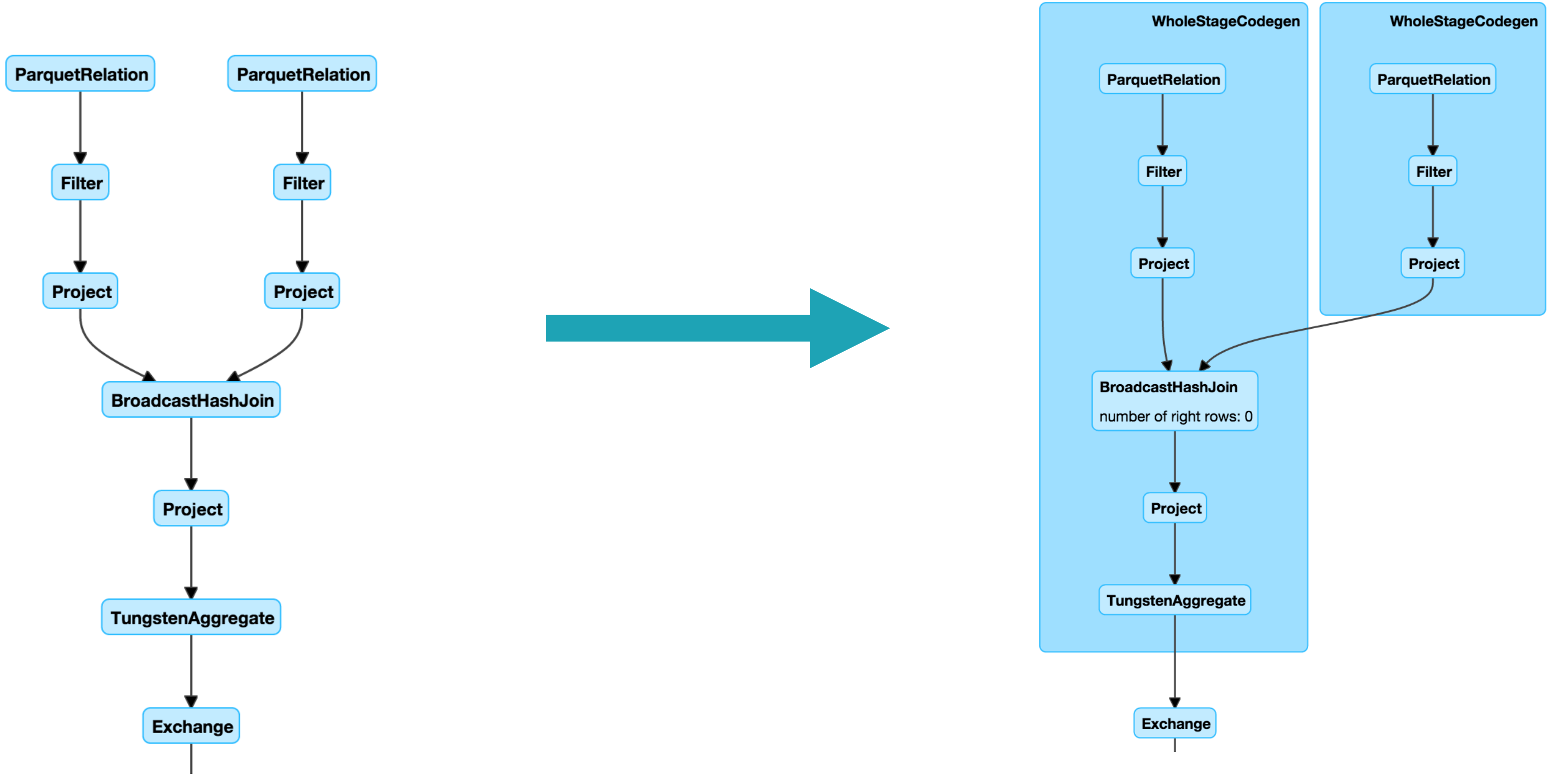
Whole-stage Codegen

Fusing operators together so the generated code looks like hand optimized code:

- Identity chains of operators (“stages”)
- Compile each stage into a single function
- Functionality of a general purpose execution engine; performance as if hand built system just to run your query

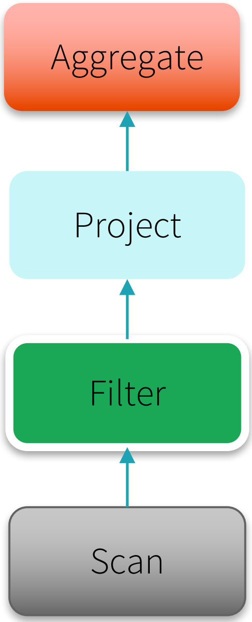
* Source: Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware.

Whole-stage Codegen: Planner



databricks™

Whole-stage Codegen: Spark as a "Compiler"




```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

The new APIs made this possible

DataFrame specifies high-level “intent”, similar to SQL

Spark understands the intent, and then optimizes the execution

API principle: Sufficiently abstracted to allow automatic optimization

Two interesting directions for Spark

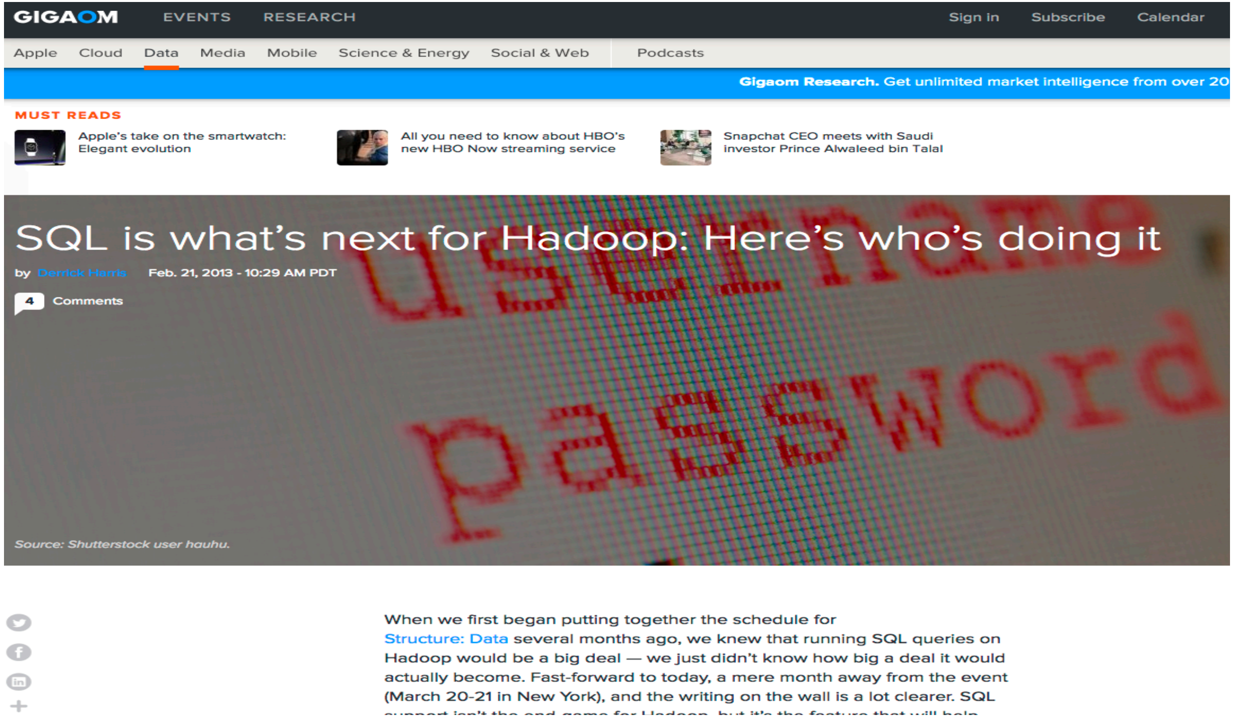
Multi-core scalability

- Machine with 128 cores start to look remarkably similar to distributed systems
- Spark runs reasonably well on a single laptop

Continuous (streaming) applications

- Very often a production data pipeline runs continuously against infinite data

Return of SQL



GIGAOM EVENTS RESEARCH Sign in Subscribe Calendar

Apple Cloud Data Media Mobile Science & Energy Social & Web Podcasts

Gigaom Research. Get unlimited market intelligence from over 20

MUST READS

Apple's take on the smartwatch: Elegant evolution

All you need to know about HBO's new HBO Now streaming service

Snapchat CEO meets with Saudi investor Prince Alwaleed bin Talal

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

by Derrick Harris Feb. 21, 2013 - 10:29 AM PDT

4 Comments

Source: Shutterstock user hauhu.

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

Dremel: Interactive Analysis of Web-Scale Datasets

ABSTR

Dremel is a
sis of read-
trees and co
tion queries
to thousand
of users at
and implem
MapReduce
age represe
few-thousan

1. INT

Large-scale
web compa
storage tha
data. Putti
has grown
ten make a
ing, online

Tenzing
A SQL Implementation On The MapReduce Framework

Biswap
Chattopa
biswape

Prathy
Arago
prathyus

ABSTRACT

Tenzing is a query engi
for ad hoc analysis of
mostly complete SQL in
sions) combined with sev
erogeneity, high perform
data awareness, low late
and structured data, and
rently used internally a
serves 10000+ queries p
pressed data. In this p
and implementation of T
typical analytical querie

1. INTRODUCTI

The MapReduce [9] fra
both inside and outside
has quickly become the f

Processing a Trillion Cells per Mouse Click

Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, Marc Nunkesser
Google, Inc.
{alexhall, olafb, buessow, silviu, marcnunkesser}@google.com

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 one of the central components of PowerDrill[1]. 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 enormous

relevant columns. Obviously, in denormalized datasets with often several thousands of columns this can make a huge difference compared to the the row-wise storage used by most database systems. Moreover, columnar formats compress very well, thus leading to less I/O and main memory usage.

At Google multiple frameworks have been developed to support data analysis at a very large scale. Best known and most widely used are MapReduce [13] and Dremel [23]. Both are highly distributed systems processing requests on thousands of machines. The latter is a column-store providing interactive query speeds for ad hoc SQL-like queries.

In this paper we present an alternative column-store developed at Google as part of the PowerDrill project. For typical user queries originating from an interactive Web UI (developed as part of the same project) it gives a performance boost of 10–100x compared to traditional column-stores which do full scans of the data.

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

databricks

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)

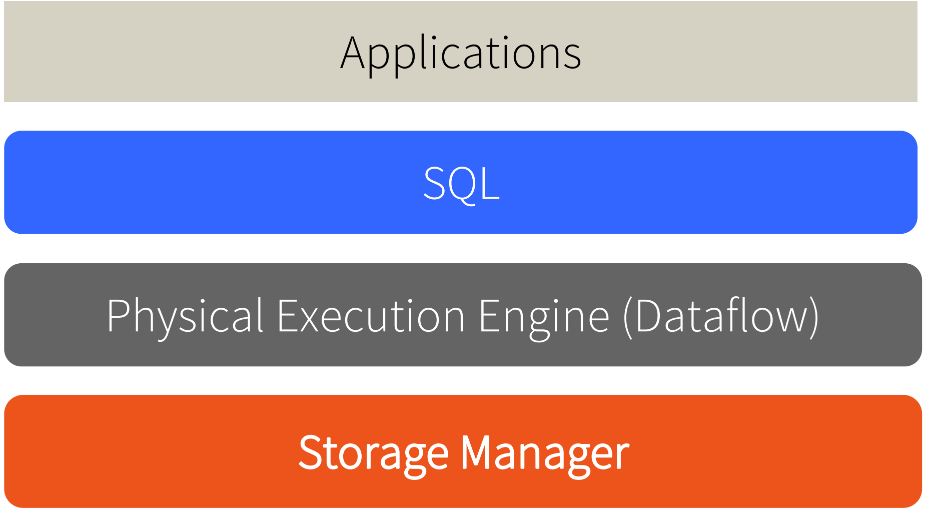
What's really different?

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

Two perspectives:

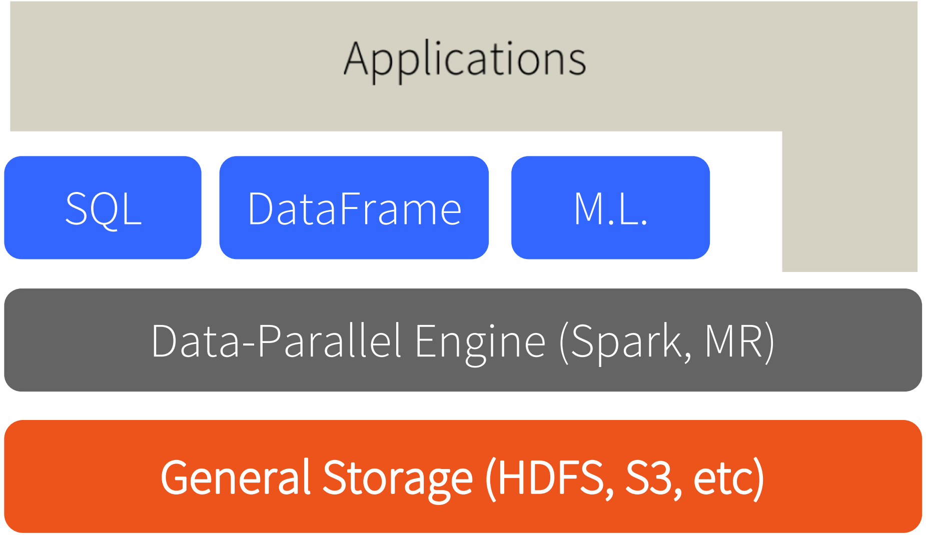
1. Flexibility in data and compute model
2. Fault-tolerance

Traditional Database Systems (Monolithic)



One way (SQL) in/out and data must be structured

Big Data Systems (Layered)



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	2014 - 2017
Applications	Applications
SQL	SQL
Physical Execution Engine (Dataflow)	Physical Execution Engine (Dataflow)
Storage Manager	**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

Google | Official Blog

Insights from Googlers into our products, technology, and the Google culture

Sorting 1PB with MapReduce

Posted: Friday, November 21, 2008

g+1 53 Tweet 38 Like 73

At Google we are fanatical about organizing the world's information. As a result, we spend a lot of time finding better ways to sort information using MapReduce, a key component of our software infrastructure that allows us to run multiple processes simultaneously. MapReduce is a perfect solution for many of the computations we run daily ...

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).

information.

databricks

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

What did we talk about today?

What is “Big Data” (BD)?

Distributed data processing / MPP databases

GFS, MapReduce, Hadoop

Spark

What’s different between BD and DB?

Thanks! Questions?

(And yes we are hiring)

rxin@databricks.com

databricks™

Acknowledgement

Some materials taken from:

Zaharia. Processing Big Data with Small Programs

Franklin. SQL, NoSQL, NewSQL? CS186 2013

DeWitt. Data Warehousing in the Cloud, The End of Shared Nothing