Data Intensive Computing Frameworks

Amir H. Payberah
Swedish Institute of Computer Science

amir@sics.se
1393/10/17

Big Data

small data

big data

- Big Data refers to datasets and flows large enough that has outpaced our capability to store, process, analyze, and understand.

Where Does Big Data Come From?

Big Data Market Driving Factors

The number of web pages indexed by Google, which were around one million in 1998, have exceeded one trillion in 2008, and its expansion is accelerated by appearance of the social networks.*

*"Mining big data: current status, and forecast to the future" [Wei Fan et al., 2013]

Big Data Market Driving Factors

The amount of mobile data traffic is expected to grow to 10.8 Exabyte per month by 2016.∗

∗ "Worldwide Big Data Technology and Services 2012-2015 Forecast" [Dan Vesset et al., 2013]

Big Data Market Driving Factors

More than 65 billion devices were connected to the Internet by 2010, and this number will go up to 230 billion by 2020.*

*"The Internet of Things Is Coming" [John Mahoney et al., 2013]

Big Data Market Driving Factors

Many companies are moving towards using Cloud services to access Big Data analytical tools.

Big Data Market Driving Factors

Open source communities

How To Store and Process Big Data?

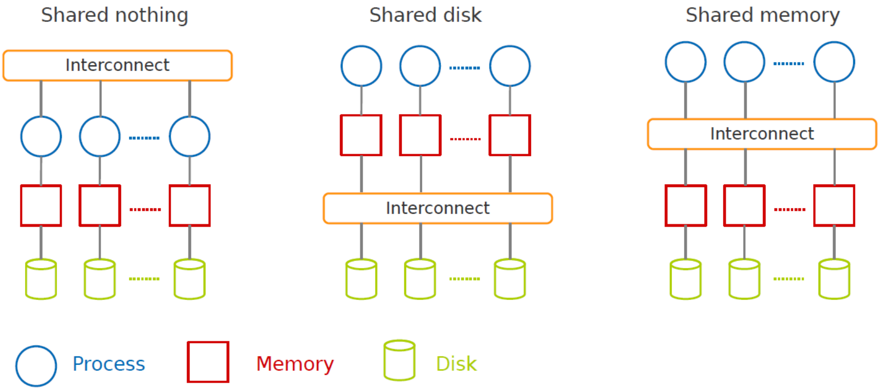
Scale Up vs. Scale Out (1/2)

- Scale up or scale vertically: adding resources to a single node in a system.
- Scale out or scale horizontally: adding more nodes to a system.

Scale Up vs. Scale Out (2/2)

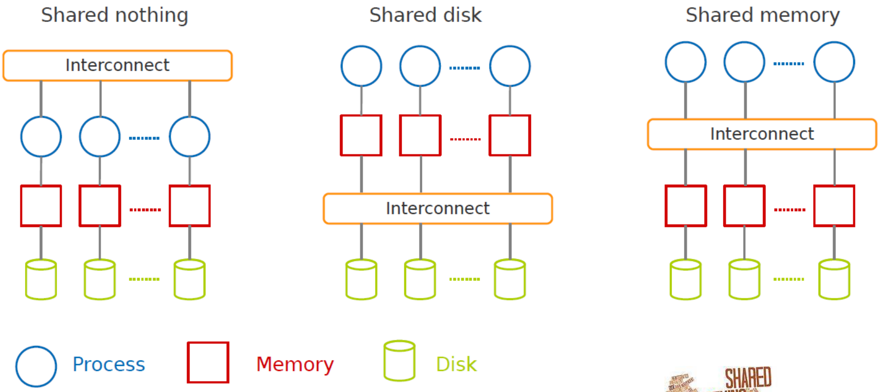
- Scale up: more expensive than scaling out.
- Scale out: more challenging for fault tolerance and software development.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. "Parallel database systems: the future of high performance database systems". ACM Communications, 35(6), 85-98, 1992.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. "Parallel database systems: the future of high performance database systems". ACM Communications, 35(6), 85-98, 1992.

APACHE HBASE

mahout

GraphX

hadoop

StratoSphere
Above the Clouds

Storm

GraphLab

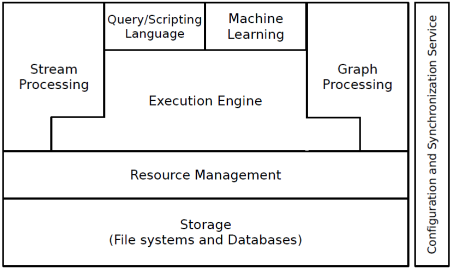
S4 distributed stream computing platform

HIVE

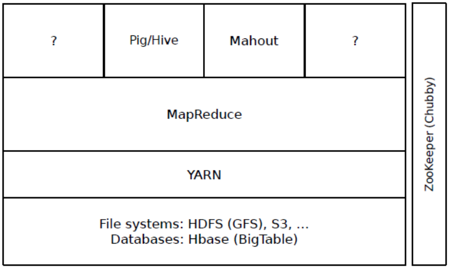
Spark

cassandra

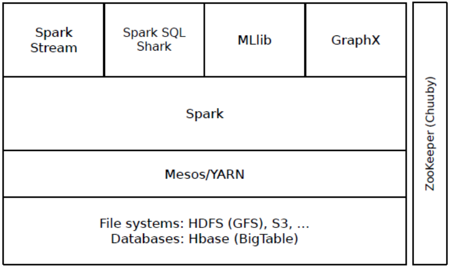
Big Data Analytics Stack



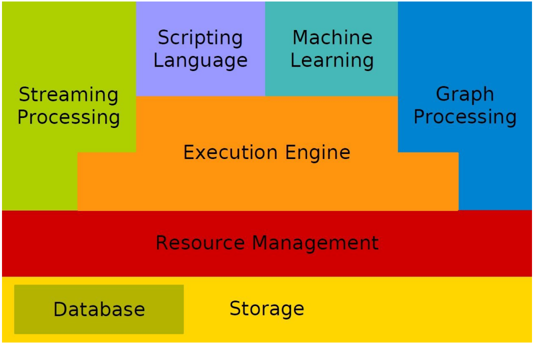
Hadoop Big Data Analytics Stack



Spark Big Data Analytics Stack

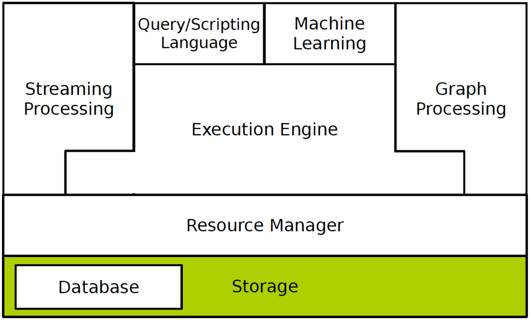


Outline



Streaming Processing
Scripting Language
Machine Learning
Graph Processing
Execution Engine
Resource Management
Database
Storage

Outline


Streaming Processing
Query/Scripting Language
Machine Learning
Graph Processing
Execution Engine
Resource Manager
Database
Storage

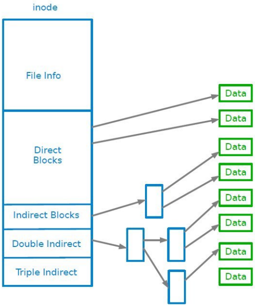
hadoop
HDFS

What is Filesystem?

- Controls how data is stored in and retrieved from disk.

What is Filesystem?

- Controls how data is stored in and retrieved from disk.



Distributed Filesystems

- When data outgrows the storage capacity of a single machine: partition it across a number of separate machines.

- Distributed filesystems: manage the storage across a network of machines.

HDFS

- Hadoop Distributed FileSystem
- Appears as a single disk
- Runs on top of a native filesystem, e.g., ext3
- Fault tolerant: can handle disk crashes, machine crashes, ...
- Based on Google's filesystem GFS

HDFS is Good for ...

- Storing large files
  - Terabytes, Petabytes, etc...
  - 100MB or more per file.
- Streaming data access
  - Data is written once and read many times.
  - Optimized for batch reads rather than random reads.
- Cheap commodity hardware
  - No need for super-computers, use less reliable commodity hardware.

HDFS is Not Good for ...

- Low-latency reads
  - High-throughput rather than low latency for small chunks of data.
  - HBase addresses this issue.

- Large amount of small files
  - Better for millions of large files instead of billions of small files.

- Multiple writers
  - Single writer per file.
  - Writes only at the end of file, no-support for arbitrary offset.

HDFS Daemons (1/2)

- HDFS cluster is manager by three types of processes.

HDFS Daemons (1/2)

- HDFS cluster is manager by three types of processes.

- Namenode
  - Manages the filesystem, e.g., namespace, meta-data, and file blocks
  - Metadata is stored in memory.

HDFS Daemons (1/2)

- HDFS cluster is manager by three types of processes.

- Namenode
  - Manages the filesystem, e.g., namespace, meta-data, and file blocks
  - Metadata is stored in memory.

- Datanode
  - Stores and retrieves data blocks
  - Reports to Namenode
  - Runs on many machines

HDFS Daemons (1/2)

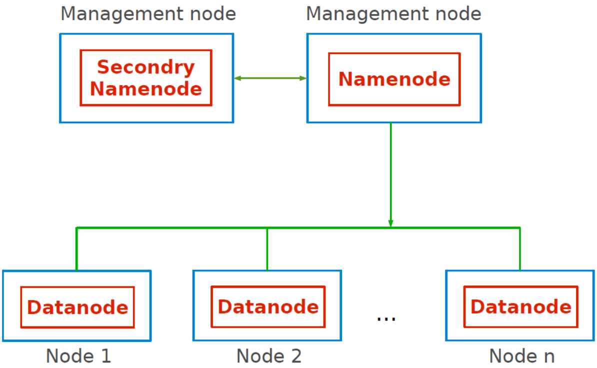
- HDFS cluster is manager by three types of processes.

- Namenode
  - Manages the filesystem, e.g., namespace, meta-data, and file blocks
  - Metadata is stored in memory.

- Datanode
  - Stores and retrieves data blocks
  - Reports to Namenode
  - Runs on many machines

- Secondary Namenode
  - Only for checkpointing.
  - Not a backup for Namenode

HDFS Daemons (2/2)



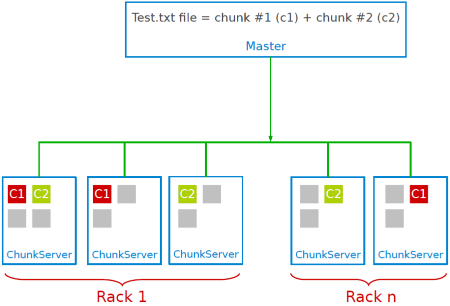
Files and Blocks (1/2)

- Files are split into blocks.
- Blocks
  - Single unit of storage: a contiguous piece of information on a disk.
  - Transparent to user.
  - Managed by Namenode, stored by Datanode.
  - Blocks are traditionally either 64MB or 128MB: default is 64MB.

Files and Blocks (2/2)

- Same block is replicated on multiple machines: default is 3
  - Replica placements are rack aware.
  - 1st replica on the local rack.
  - 2nd replica on the local rack but different machine.
  - 3rd replica on the different rack.

- Namenode determines replica placement.

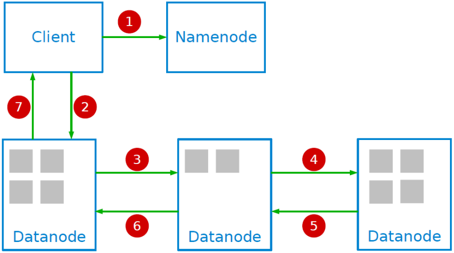


HDFS Client

- Client interacts with Namenode
  - To update the Namenode namespace.
  - To retrieve block locations for writing and reading.
- Client interacts directly with Datanode
  - To read and write data.
- Namenode does not directly write or read data.

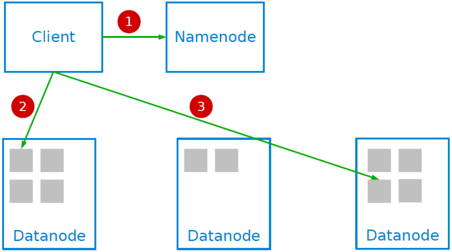
HDFS Write

- 1. Create a new file in the Namenode's Namespace; calculate block topology.
- 2, 3, 4. Stream data to the first, second and third node.
- 5, 6, 7. Success/failure acknowledgment.

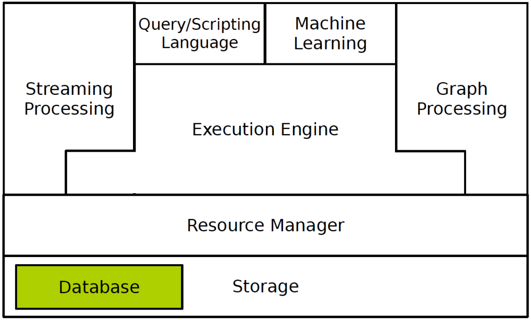


HDFS Read

- 1. Retrieve block locations.
- 2, 3. Read blocks to re-assemble the file.



Outline



Streaming Processing
Query/Scripting Language
Machine Learning
Graph Processing
Execution Engine
Resource Manager
Database
Storage

Database and Database Management System

- Database: an organized collection of data.

Database and Database Management System

- Database: an organized collection of data.

- Database Management System (DBMS): a software that interacts with users, other applications, and the database itself to capture and analyze data.

Relational Databases Management Systems (RDMBSs)

- RDMBSs: the dominant technology for storing structured data in web and business applications.

- SQL is good
  - Rich language
  - Easy to use and integrate
  - Rich toolset
  - Many vendors

- They promise: ACID

ACID Properties

- Atomicity
  - All included statements in a transaction are either executed or the whole transaction is aborted without affecting the database.

ACID Properties

- Atomicity
  - All included statements in a transaction are either executed or the whole transaction is aborted without affecting the database.

- Consistency
  - A database is in a consistent state before and after a transaction.

ACID Properties

- Atomicity
  - All included statements in a transaction are either executed or the whole transaction is aborted without affecting the database.
- Consistency
  - A database is in a consistent state before and after a transaction.
- Isolation
  - Transactions can not see uncommitted changes in the database.

ACID Properties

- Atomicity
  - All included statements in a transaction are either executed or the whole transaction is aborted without affecting the database.
- Consistency
  - A database is in a consistent state before and after a transaction.
- Isolation
  - Transactions can not see uncommitted changes in the database.
- Durability
  - Changes are written to a disk before a database commits a transaction so that committed data cannot be lost through a power failure.

RDBMS Challenges

- Web-based applications caused spikes.
  - Internet-scale data size
  - High read-write rates
  - Frequent schema changes

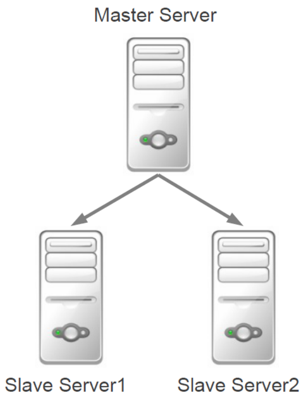
Let's Scale RDBMSs

- RDBMS were not designed to be distributed.

- Possible solutions:
  - Replication
  - Sharding

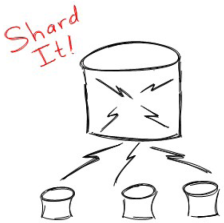
Let's Scale RDBMSs - Replication

- Master/Slave architecture
- Scales read operations

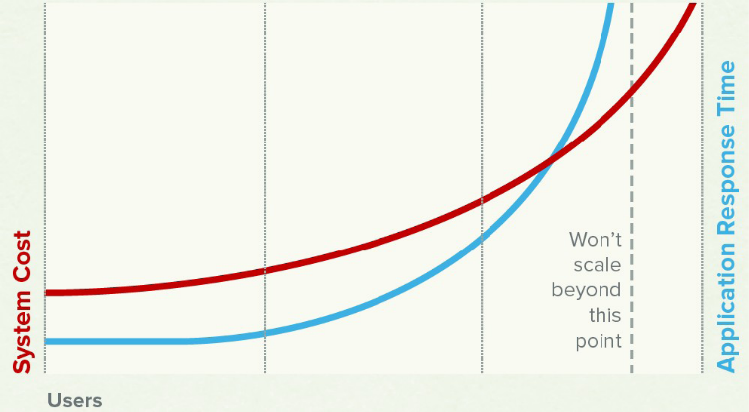


Let's Scale RDBMSs - Sharding

- Dividing the database across many machines.
- It scales read and write operations.
- Cannot execute transactions across shards (partitions).



Scaling RDBMSs is Expensive and Inefficient



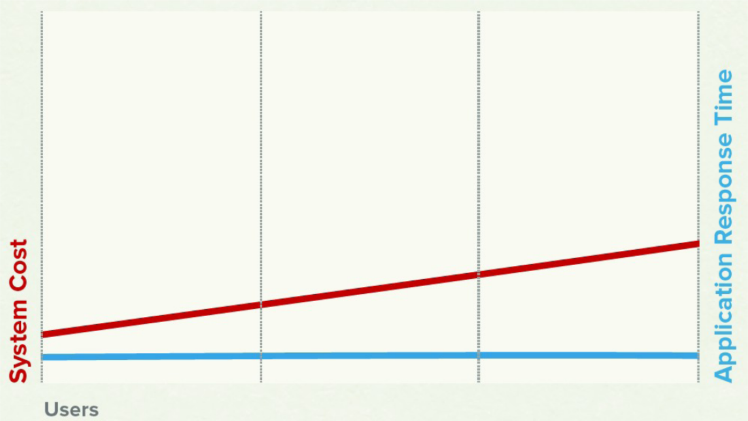
[http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQLWhitepaper.pdf]

Not Only SQL

NoSQL

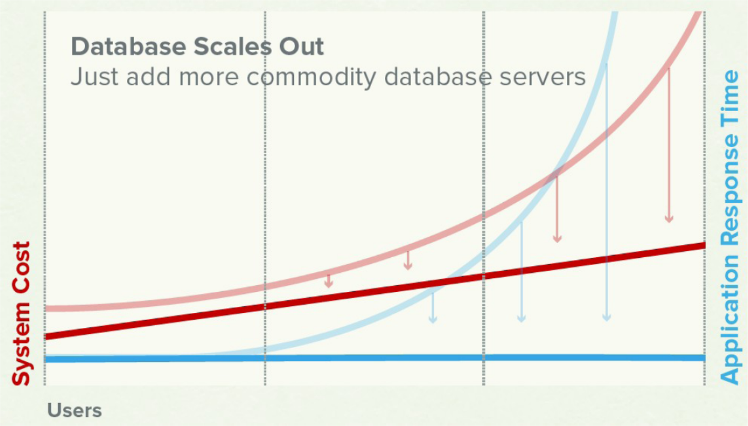
- Avoidance of unneeded complexity
- High throughput
- Horizontal scalability and running on commodity hardware
- Compromising reliability for better performance

NoSQL Cost and Performance



[http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQLWhitepaper.pdf]

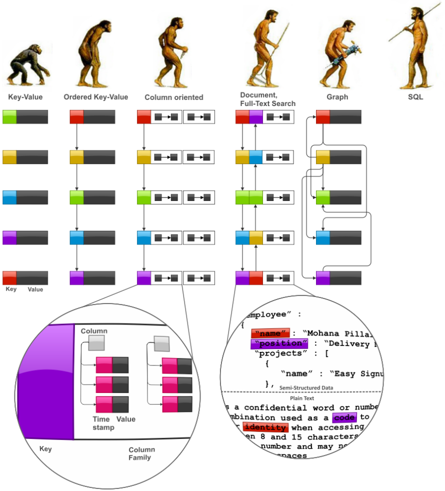
RDBMS vs. NoSQL



[http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQLWhitepaper.pdf]

NoSQL Data Models

NoSQL Data Models



[http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques]

Key-Value Data Model

- Collection of key/value pairs.
- Ordered Key-Value: processing over key ranges.
- Dynamo, Scalaris, Voldemort, Riak, ...

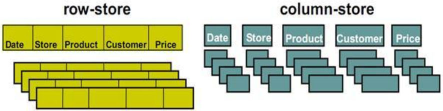
Column-Oriented Data Model

- Similar to a key/value store, but the value can have multiple attributes (Columns).

- Column: a set of data values of a particular type.

- Store and process data by column instead of row.

- BigTable, Hbase, Cassandra, ...



Document Data Model

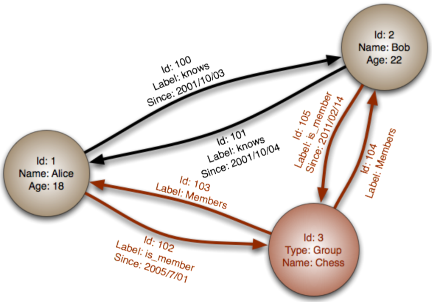
- Similar to a column-oriented store, but values can have complex documents, instead of fixed format.
- Flexible schema.
- XML, YAML, JSON, and BSON.
- CouchDB, MongoDB, ...

```
{
    FirstName: "Bob",
    Address: "5 Oak St.",
    Hobby: "sailing"
 }

 {
   FirstName: "Jonathan",
   Address: "15 Wanamassa Point Road",
   Children: [
        {Name: "Michael", Age: 10},
        {Name: "Jennifer", Age: 8},
   ]
  }
```

Graph Data Model

- Uses graph structures with nodes, edges, and properties to represent and store data.

- Neo4J, InfoGrid, ...

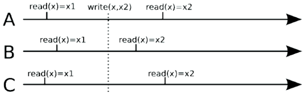


[http://en.wikipedia.org/wiki/Graph_database]

CAP Theorem

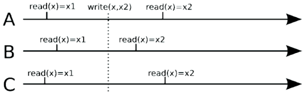
Consistency

- Strong consistency
  - After an update completes, any subsequent access will return the updated value.

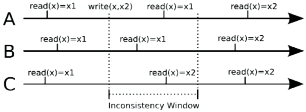


Consistency

- Strong consistency
  - After an update completes, any subsequent access will return the updated value.



- Eventual consistency
  - Does not guarantee that subsequent accesses will return the updated value.
  - Inconsistency window.
  - If no new updates are made to the object, eventually all accesses will return the last updated value.



Quorum Model

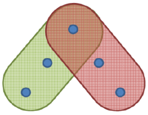
- N: the number of nodes to which a data item is replicated.
- R: the number of nodes a value has to be read from to be accepted.
- W: the number of nodes a new value has to be written to before the write operation is finished.

- To enforce strong consistency: $R + W > N$

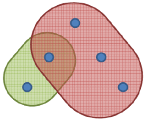
Quorum Model

- N: the number of nodes to which a data item is replicated.
- R: the number of nodes a value has to be read from to be accepted.
- W: the number of nodes a new value has to be written to before the write operation is finished.

- To enforce strong consistency: $R + W > N$



R = 3, W = 3, N = 5

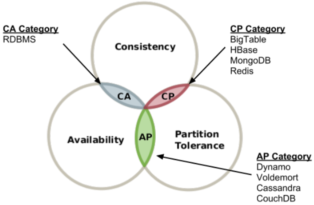


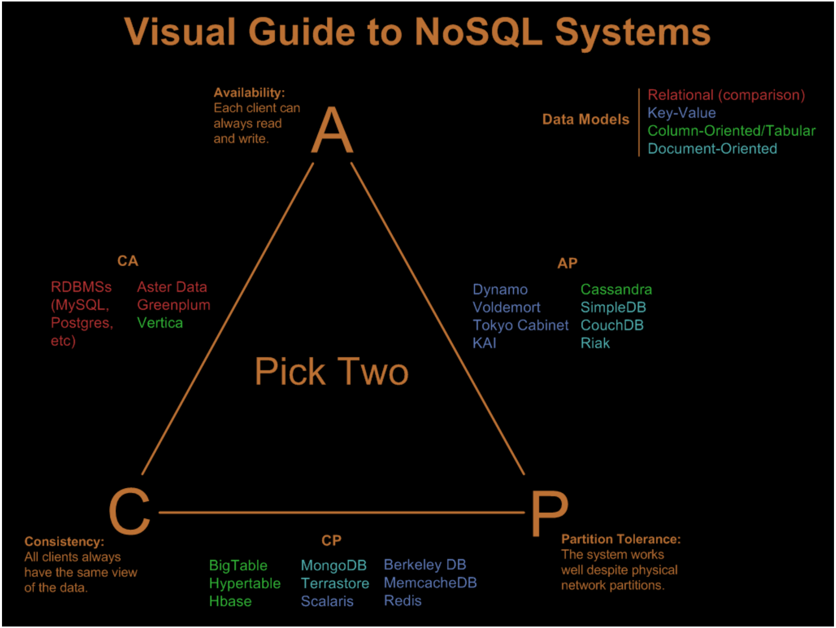
R = 4, W = 2, N = 5

CAP Theorem

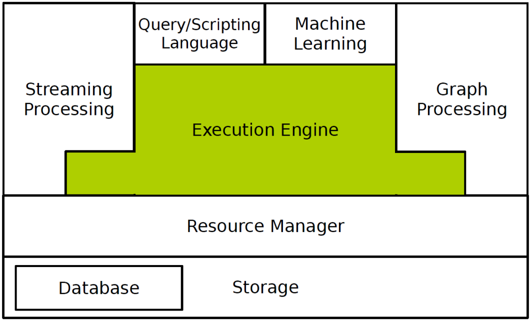
- Consistency
  - Consistent state of data after the execution of an operation.
- Availability
  - Clients can always read and write data.
- Partition Tolerance
  - Continue the operation in the presence of network partitions.

- You can choose only two!




Visual Guide to NoSQL Systems
Availability:
Each client can always read and write.
A
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
C
P
Consistency:
All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance:
The system works well despite physical network partitions.

Outline



Streaming Processing
Query/Scripting Language
Machine Learning
Graph Processing
Execution Engine
Resource Manager
Database
Storage

MapReduce

MapReduce

- A shared nothing architecture for processing large data sets with a parallel/distributed algorithm on clusters.

MapReduce Definition

- A programming model: to batch process large data sets (inspired by functional programming).

MapReduce Definition

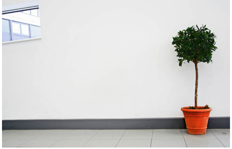
- A programming model: to batch process large data sets (inspired by functional programming).

- An execution framework: to run parallel algorithms on clusters of commodity hardware.

Simplicity

- Don't worry about parallelization, fault tolerance, data distribution, and load balancing (MapReduce takes care of these).
- Hide system-level details from programmers.

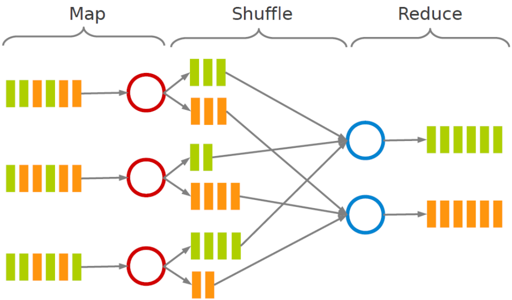
Simplicity!



Programming Model

MapReduce Dataflow

- map function: processes data and generates a set of intermediate key/value pairs.
- reduce function: merges all intermediate values associated with the same intermediate key.



Example: Word Count

- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Example: Word Count - map

- The map function reads in words one a time and outputs (word, 1) for each parsed input word.

- The map function output is:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Example: Word Count - shuffle

- The shuffle phase between map and reduce phase creates a list of values associated with each key.

- The reduce function input is:

```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1))
(Hello, (1, 1))
(World, (1, 1))
```

Example: Word Count - reduce

- The reduce function sums the numbers in the list for each key and outputs (word, count) pairs.

- The output of the reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

Example: Word Count - map

```
public static class MyMap extends Mapper<...> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
  }
}
```

Example: Word Count - reduce

```
public static class MyReduce extends Reducer<...> {
  public void reduce(Text key, Iterator<...> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;

    while (values.hasNext())
      sum += values.next().get();

    context.write(key, new IntWritable(sum));
  }
}
```

Example: Word Count - driver

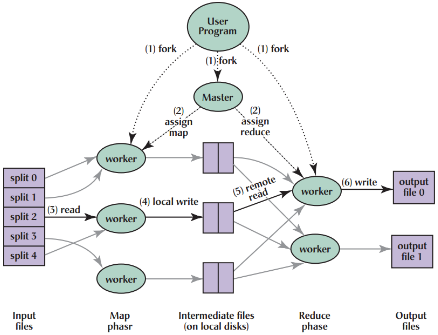
```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(conf, "wordcount");

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setMapperClass(MyMap.class);
  job.setReducerClass(MyReduce.class);

  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);

  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.waitForCompletion(true);
}
```

Execution Engine

MapReduce Execution (1/7)

- The user program divides the input files into M splits.
  - A typical size of a split is the size of a HDFS block (64 MB).
  - Converts them to key/value pairs.

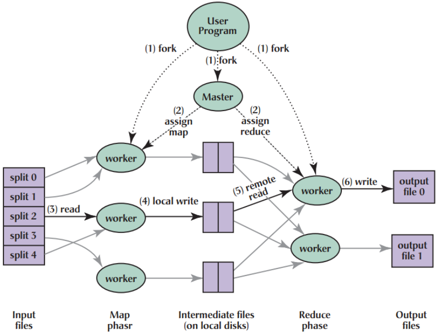
- It starts up many copies of the program on a cluster of machines.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

MapReduce Execution (2/7)

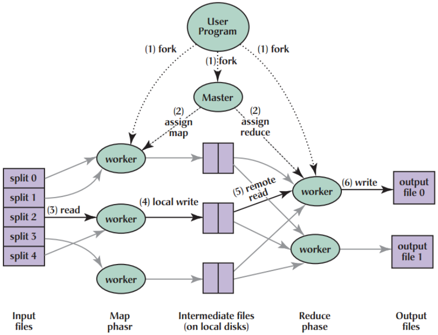
- One of the copies of the program is master, and the rest are workers.
- The master assigns works to the workers.
  - It picks idle workers and assigns each one a map task or a reduce task.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

MapReduce Execution (3/7)

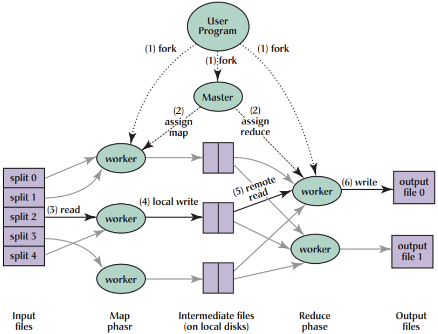
- A map worker reads the contents of the corresponding input splits.
- It parses key/value pairs out of the input data and passes each pair to the user defined map function.
- The intermediate key/value pairs produced by the map function are buffered in memory.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

MapReduce Execution (4/7)

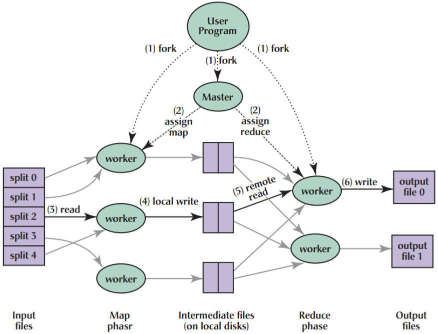
- The buffered pairs are periodically written to local disk.
  - They are partitioned into R regions (hash(key) mod R).
- The locations of the buffered pairs on the local disk are passed back to the master.
- The master forwards these locations to the reduce workers.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

MapReduce Execution (5/7)

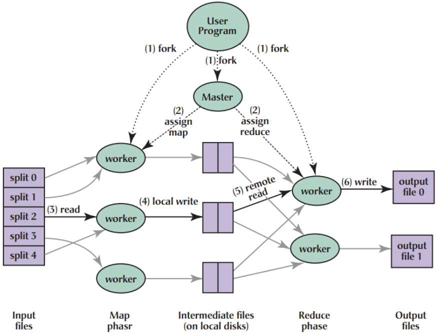
- A reduce worker reads the buffered data from the local disks of the map workers.
- When a reduce worker has read all intermediate data, it sorts it by the intermediate keys.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

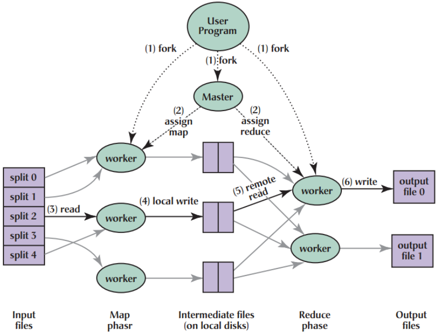
MapReduce Execution (6/7)

- The reduce worker iterates over the intermediate data.
- For each unique intermediate key, it passes the key and the corresponding set of intermediate values to the user defined reduce function.
- The output of the reduce function is appended to a final output file for this reduce partition.



MapReduce Execution (7/7)

- When all map tasks and reduce tasks have been completed, the master wakes up the user program.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

What is Spark?

- An efficient distributed general-purpose data analysis platform.
- Focusing on ease of programming and high performance.

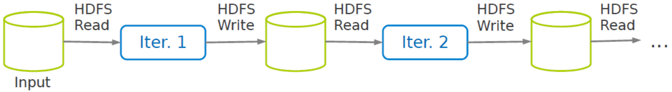
Motivation

- MapReduce programming model has not been designed for complex operations, e.g., data mining.
- Very expensive, i.e., always goes to disk and HDFS.

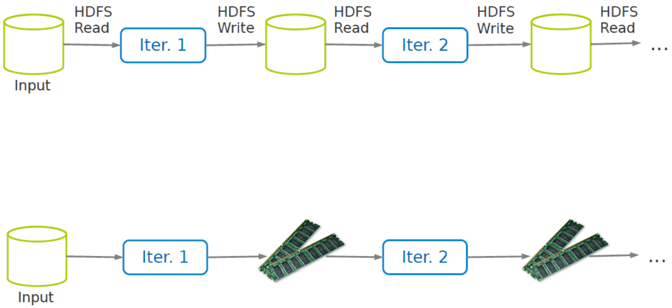
Solution

- Extends MapReduce with more operators.
- Support for advanced data flow graphs.
- In-memory and out-of-core processing.

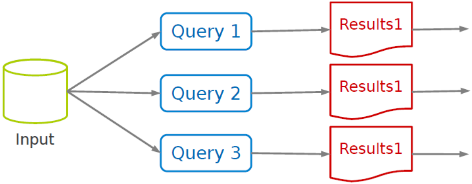
Spark vs. Hadoop



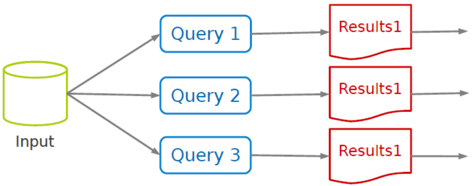
Spark vs. Hadoop

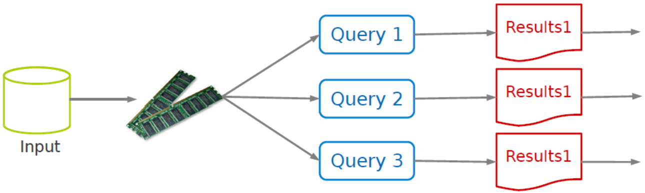


Spark vs. Hadoop



Spark vs. Hadoop





Resilient Distributed Datasets (RDD) (1/2)

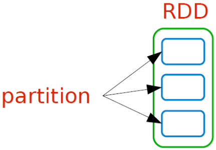
- A distributed memory abstraction.

Resilient Distributed Datasets (RDD) (1/2)

- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.



- Partitions of an RDD can be stored on different nodes of a cluster.

RDD Operators

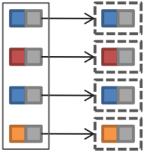
- Higher-order functions: transformations and actions.
- Transformations: lazy operators that create new RDDs.
- Actions: launch a computation and return a value to the program or write data to the external storage.

Transformations vs. Actions

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

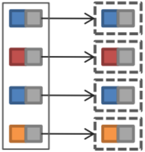
RDD Transformations - Map

- All pairs are independently processed.



RDD Transformations - Map

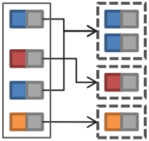
- All pairs are independently processed.



```
// passing each element through a function.
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x) // {1, 4, 9}
```

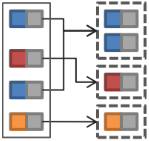
RDD Transformations - GroupBy

- Pairs with identical key are grouped.
- Groups are independently processed.



RDD Transformations - GroupBy

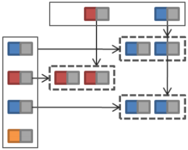
- Pairs with identical key are grouped.
- Groups are independently processed.



```scala
val schools = sc.parallelize(Seq(("sics", 1), ("kth", 1), ("sics", 2)))

schools.groupByKey()
// {("sics", (1, 2)), ("kth", (1))}

schools.reduceByKey((x, y) => x + y)
// {("sics", 3), ("kth", 1)}
```

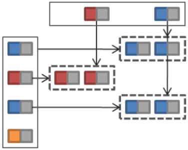
RDD Transformations - Join

- Performs an equi-join on the key.
- Join candidates are independently processed.



RDD Transformations - Join

- Performs an equi-join on the key.
- Join candidates are independently processed.



```
val list1 = sc.parallelize(Seq(("sics", "10"),
                               ("kth", "50"),
                               ("sics", "20")))

val list2 = sc.parallelize(Seq(("sics", "upsala"),
                               ("kth", "stockholm")))

list1.join(list2)
  // ("sics", ("10", "upsala"))
  // ("sics", ("20", "upsala"))
  // ("kth", ("50", "stockholm"))
```

Basic RDD Actions

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

Basic RDD Actions

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

Basic RDD Actions

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- Return the number of elements in the RDD.

```
nums.count() // 3
```

Basic RDD Actions

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- Return the number of elements in the RDD.

```
nums.count() // 3
```

- Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y) // 6
```

Creating RDDs

- Turn a collection into an RDD.

```
val a = sc.parallelize(Array(1, 2, 3))
```

- Load text file from local FS, HDFS, or S3.

```
val a = sc.textFile("file.txt")
val b = sc.textFile("directory/*.txt")
val c = sc.textFile("hdfs://namenode:9000/path/file")
```

SparkContext

- Main entry point to Spark functionality.
- Available in shell as variable sc.
- In standalone programs, you should make your own.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val sc = new SparkContext(master, appName, [sparkHome], [jars])
```

Example

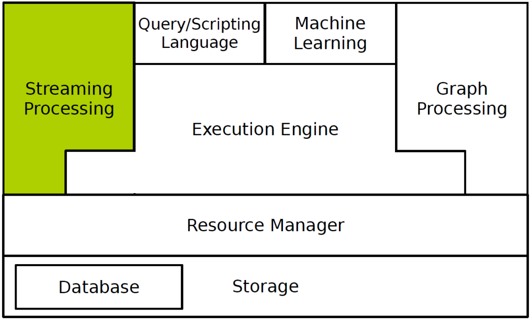
- Read data from a text file and count the total number of words..

Example

- Read data from a text file and count the total number of words..

```
val lines = sc.textFile("hamlet.txt")
val eachWordCounts = lines.flatMap(_.split(" "))
  .map(word => (word, 1))
  .reduceByKey((a, b) => a + b)
```

Outline


Streaming Processing
Query/Scripting Language
Machine Learning
Graph Processing
Execution Engine
Resource Manager
Database
Storage

Motivation

- Many applications must process large streams of live data and provide results in real-time.
- Processing information as it flows, without storing them persistently.

Motivation

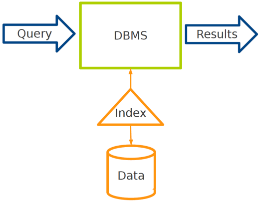
- Many applications must process large streams of live data and provide results in real-time.

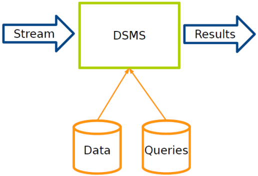
- Processing information as it flows, without storing them persistently.

- Traditional DBMSs:
    - Store and index data before processing it.
    - Process data only when explicitly asked by the users.
    - Both aspects contrast with our requirements.

DBMS vs. DSMS (1/3)

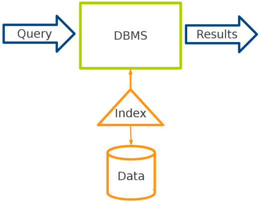
- DBMS: persistent data where updates are relatively infrequent.
- DSMS: transient data that is continuously updated.

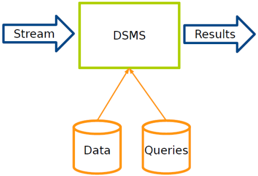




DBMS vs. DSMS (2/3)

- DBMS: runs queries just once to return a complete answer.
- DSMS: executes standing queries, which run continuously and provide updated answers as new data arrives.



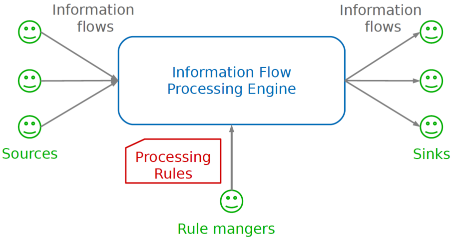


DBMS vs. DSMS (3/3)

- Despite these differences, DSMSs resemble DBMSs: both process incoming data through a sequence of transformations based on SQL operators, e.g., selections, aggregates, joins.

DSMS

- Source: produces the incoming information flows
- Sink: consumes the results of processing
- IFP engine: processes incoming flows
- Processing rules: how to process the incoming flows
- Rule manager: adds/removes processing rules



Spark Streaming

Spark Streaming

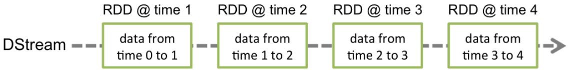
- Run a streaming computation as a series of very small, deterministic batch jobs.

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.
  - Spark treats each batch of data as RDDs and processes them using RDD operations.
  - Finally, the processed results of the RDD operations are returned in batches.

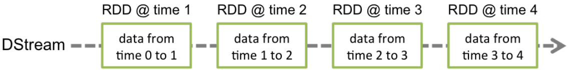
DStream

- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...



DStream

- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...

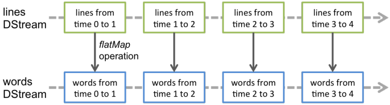


- Initializing Spark streaming

```
val scc = new StreamingContext(master, appName, batchDuration,
[sparkHome], [jars])
```

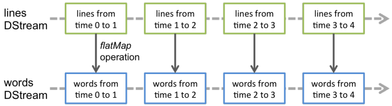
DStream Operations (1/2)

- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations (stateless/stateful operations): map, join, ...

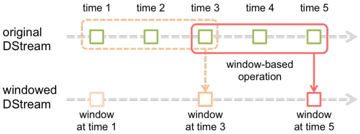


DStream Operations (1/2)

- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations (stateless/stateful operations): map, join, ...



  - Window operations: group all the records from a sliding window of the past time intervals into one RDD: window, reduceByAndWindow, ...



Window length: the duration of the window.
Slide interval: the interval at which the operation is performed.

DStream Operations (2/2)

- Output operations: send data to external entity
  - saveAsHadoopFiles, foreach, print, ...

DStream Operations (2/2)

- Output operations: send data to external entity
  - saveAsHadoopFiles, foreach, print, ...
- Attaching input sources

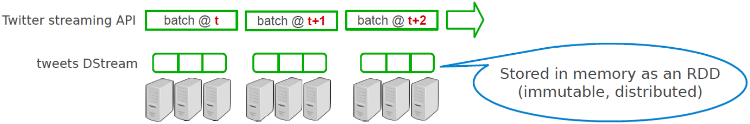
```
ssc.textFileStream(directory)
ssc.socketStream(hostname, port)
```

Example (1/3)

- Get hash-tags from Twitter.

```
val ssc = new StreamingContext("local[2]", "Tweets", Seconds(1))
val tweets = TwitterUtils.createStream(ssc, None)
```

DStream: a sequence of RDD representing a stream of data

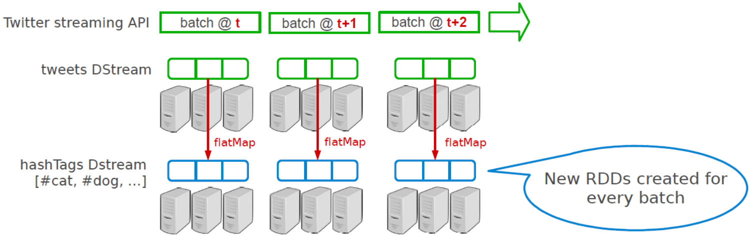


Example (2/3)

- Get hash-tags from Twitter.

```
val ssc = new StreamingContext("local[2]", "Tweets", Seconds(1))
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
```

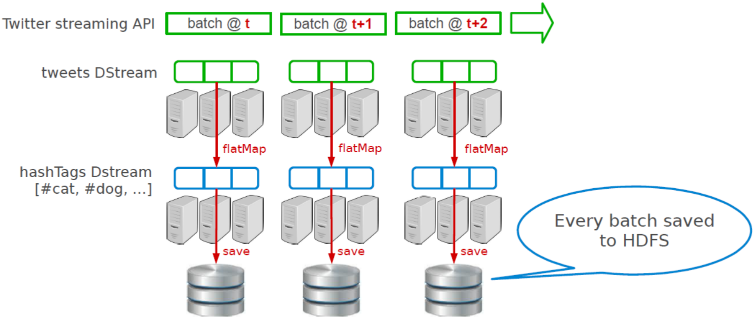
transformation: modify data in one DStream
to create another DStream



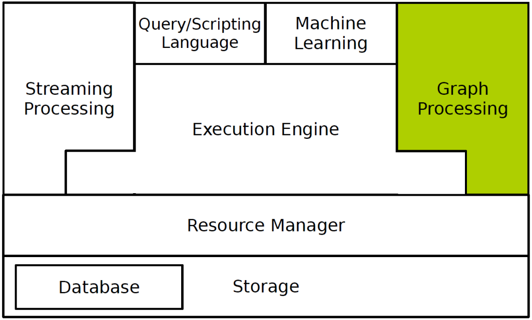
Example (3/3)

- Get hash-tags from Twitter.

```
val ssc = new StreamingContext("local[2]", "Tweets", Seconds(1))
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



Outline

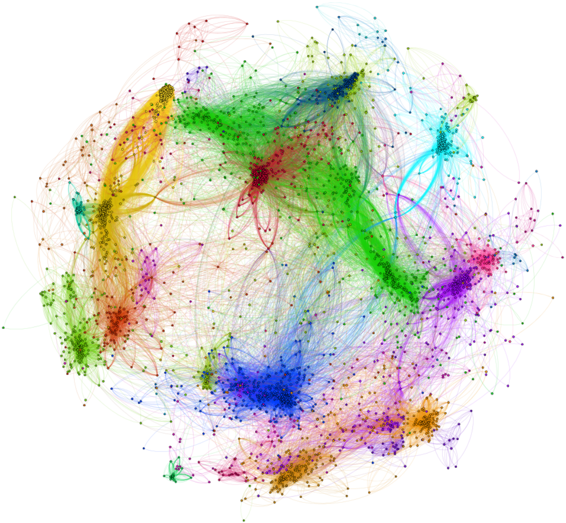

Streaming Processing
Query/Scripting Language
Machine Learning
Graph Processing
Execution Engine
Resource Manager
Database
Storage

Introduction

- Graphs provide a flexible abstraction for describing relationships between discrete objects.

- Many problems can be modeled by graphs and solved with appropriate graph algorithms.

Large Graph

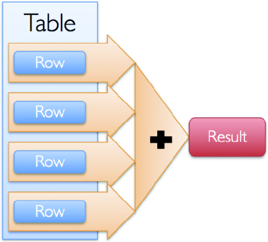


Large-Scale Graph Processing

- Large graphs need large-scale processing.
- A large graph either cannot fit into memory of single computer or it fits with huge cost.

Question

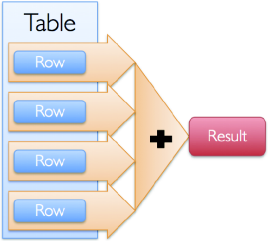
Can we use platforms like MapReduce or Spark, which are based on data-parallel model, for large-scale graph proceeding?



Data-Parallel Model for Large-Scale Graph Processing

- The platforms that have worked well for developing parallel applications are not necessarily effective for large-scale graph problems.

- Why?



Graph Algorithms Characteristics

- Unstructured problems: difficult to partition the data

Graph Algorithms Characteristics

- Unstructured problems: difficult to partition the data
- Data-driven computations: difficult to partition computation

Graph Algorithms Characteristics

- Unstructured problems: difficult to partition the data
- Data-driven computations: difficult to partition computation
- Poor data locality

Graph Algorithms Characteristics

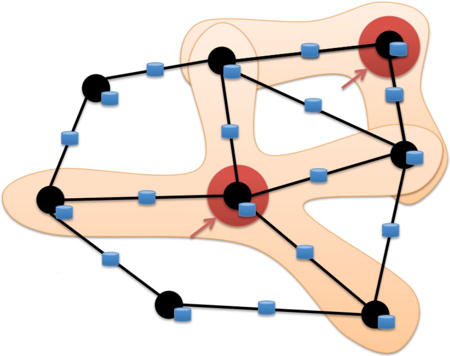
- Unstructured problems: difficult to partition the data
- Data-driven computations: difficult to partition computation
- Poor data locality
- High data access to computation ratio

Proposed Solution

Graph-Parallel Processing

Proposed Solution

Graph-Parallel Processing

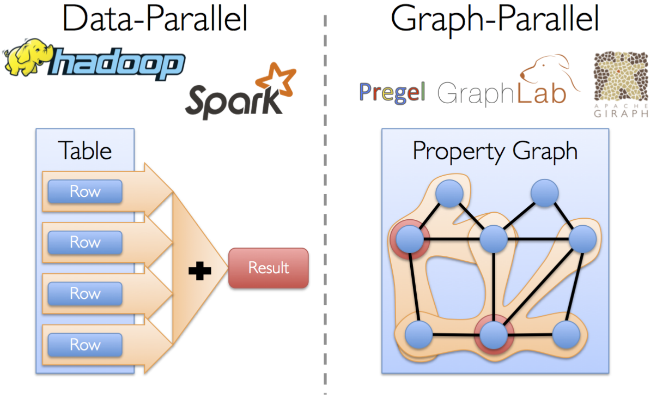


- Computation typically depends on the neighbors.

Graph-Parallel Processing

- Restricts the types of computation.
- New techniques to partition and distribute graphs.
- Exploit graph structure.
- Executes graph algorithms orders-of-magnitude faster than more general data-parallel systems.

Data-Parallel vs. Graph-Parallel Computation



Pregel

oogle

Pregel

- Large-scale graph-parallel processing platform developed at Google.
- Inspired by bulk synchronous parallel (BSP) model.

Bulk Synchronous Parallel (1/2)

- It is a parallel programming model.
- The model consists of:

Bulk Synchronous Parallel (1/2)

- It is a parallel programming model.
- The model consists of:
  - A set of processor-memory pairs.

Bulk Synchronous Parallel (1/2)

- It is a parallel programming model.
- The model consists of:
  - A set of processor-memory pairs.
  - A communications network that delivers messages in a point-to-point manner.

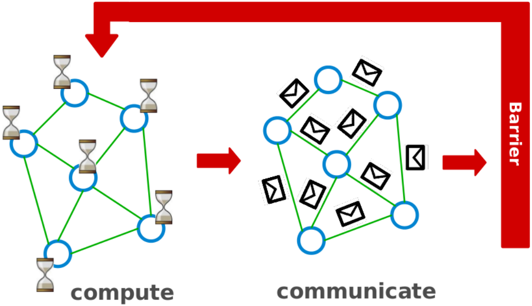
Bulk Synchronous Parallel (1/2)

- It is a parallel programming model.
- The model consists of:
  - A set of processor-memory pairs.
  - A communications network that delivers messages in a point-to-point manner.
  - A mechanism for the efficient barrier synchronization for all or a subset of the processes.

Bulk Synchronous Parallel (1/2)

- It is a parallel programming model.
- The model consists of:
  - A set of processor-memory pairs.
  - A communications network that delivers messages in a point-to-point manner.
  - A mechanism for the efficient barrier synchronization for all or a subset of the processes.
  - There are no special combining, replicating, or broadcasting facilities.

Bulk Synchronous Parallel (2/2)



All vertices update in parallel (at the same time).

Vertex-Centric Programs

- Think as a vertex.

Vertex-Centric Programs

- Think as a vertex.
- Each vertex computes individually its value: in parallel

Vertex-Centric Programs

- Think as a vertex.
- Each vertex computes individually its value: in parallel
- Each vertex can see its local context, and updates its value accordingly.

Data Model

- A directed graph that stores the program state, e.g., the current value.

Execution Model (1/3)

- Applications run in sequence of iterations: supersteps

Execution Model (1/3)

- Applications run in sequence of iterations: supersteps

- During a superstep, user-defined functions for each vertex is invoked (method `Compute()`): in parallel

Execution Model (1/3)

- Applications run in sequence of iterations: supersteps

- During a superstep, user-defined functions for each vertex is invoked (method `Compute()`): in parallel

- A vertex in superstep S can:
  - reads messages sent to it in superstep S-1.
  - sends messages to other vertices: receiving at superstep S+1.
  - modifies its state.

Execution Model (1/3)

- Applications run in sequence of iterations: supersteps
- During a superstep, user-defined functions for each vertex is invoked (method `Compute()`): in parallel
- A vertex in superstep S can:
  - reads messages sent to it in superstep S-1.
  - sends messages to other vertices: receiving at superstep S+1.
  - modifies its state.
- Vertices communicate directly with one another by sending messages.

Execution Model (2/3)

- Superstep 0: all vertices are in the active state.

Execution Model (2/3)

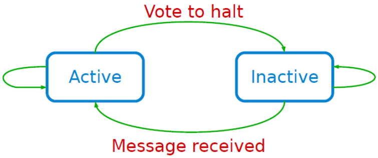
- Superstep 0: all vertices are in the active state.
- A vertex deactivates itself by voting to halt: no further work to do.

Execution Model (2/3)

- Superstep 0: all vertices are in the active state.
- A vertex deactivates itself by voting to halt: no further work to do.
- A halted vertex can be active if it receives a message.

Execution Model (2/3)

- Superstep 0: all vertices are in the active state.
- A vertex deactivates itself by voting to halt: no further work to do.
- A halted vertex can be active if it receives a message.
- The whole algorithm terminates when:
  - All vertices are simultaneously inactive.
  - There are no messages in transit.



Execution Model (3/3)

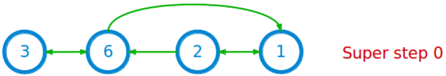
- Aggregation: a mechanism for global communication, monitoring, and data.

Execution Model (3/3)

- Aggregation: a mechanism for global communication, monitoring, and data.
- Runs after each superstep.
- Each vertex can provide a value to an aggregator in superstep S.
- The system combines those values and the resulting value is made available to all vertices in superstep $S + 1$.

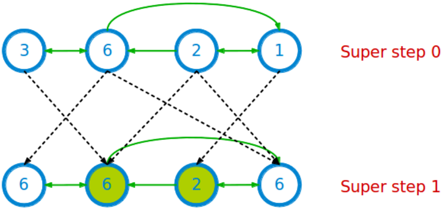
Example: Max Value (1/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



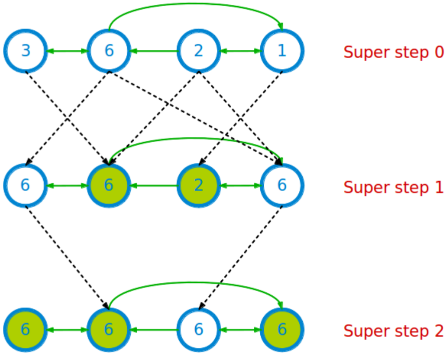
Example: Max Value (2/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



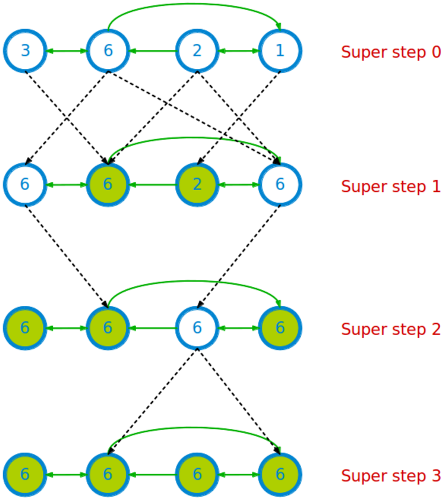
Example: Max Value (3/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



Example: Max Value (4/4)

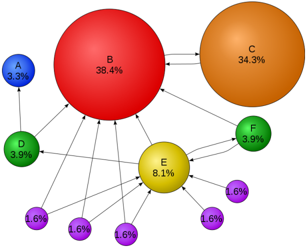
```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



Example: PageRank

- Update ranks in parallel.
- Iterate until convergence.

$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$



Example: PageRank

```
Pregel_PageRank(i, messages):
  // receive all the messages
  total = 0
  foreach(msg in messages):
    total = total + msg

  // update the rank of this vertex
  R[i] = 0.15 + total

  // send new messages to neighbors
  foreach(j in out_neighbors[i]):
    sendmsg(R[i] * wij) to vertex j
```

$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

Pregel Limitations

- Inefficient if different regions of the graph converge at different speed.
- Can suffer if one task is more expensive than the others.
- Runtime of each phase is determined by the slowest machine.

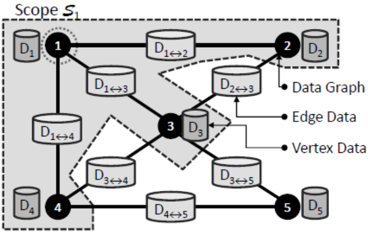
GraphLab

Data Model

- A directed graph that stores the program state, called data graph.

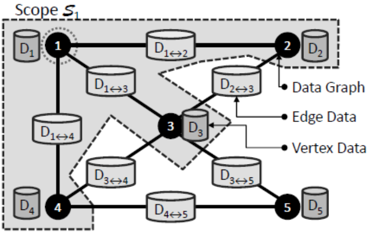
Vertex Scope

- The scope of vertex v is the data stored in vertex v, in all adjacent vertices and adjacent edges.



Programming Model (1/3)

- Rather than adopting a message passing as in Pregel, GraphLab allows the user defined function of a vertex to read and modify any of the data in its scope.



Programming Model (2/3)

- Update function: user-defined function similar to `Compute` in Pregel.
- Can read and modify the data within the scope of a vertex.
- Schedules the future execution of other update functions.

Programming Model (3/3)

- Sync function: similar to aggregate in Pregel.
- Maintains global aggregates.
- Performs periodically in the background.

Execution Model

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
    $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

Execution Model

Input: Data Graph $G = (V, E, D)$

Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$

while $\mathcal{T}$ *is not Empty* **do**

$\quad (f, v) \leftarrow \texttt{RemoveNext}(\mathcal{T})$

$\quad (\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$

$\quad \mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$

Output: Modified Data Graph $G = (V, E, D')$

- Each task in the set of tasks $\mathcal{T}$, is a tuple (f, v) consisting of an update function f and a vertex v.

Execution Model

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
  $(f, v) \leftarrow \texttt{RemoveNext}\,(\mathcal{T})$
  $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
  $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

- Each task in the set of tasks $\mathcal{T}$, is a tuple (f, v) consisting of an update function f and a vertex v.
- After executing an update function $(f, g, \cdots)$ the modified scope data in S_v is written back to the data graph.

Example: PageRank

```
GraphLab_PageRank(i)
  // compute sum over neighbors
  total = 0
  foreach(j in in_neighbors(i)):
    total = total + R[j] * wji

  // update the PageRank
  R[i] = 0.15 + total

  // trigger neighbors to run again
  foreach(j in out_neighbors(i)):
    signal vertex-program on j
```

$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

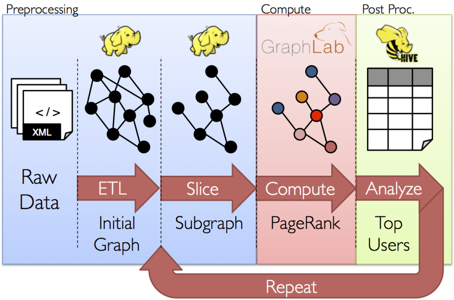
GraphX

Data-Parallel vs. Graph-Parallel Computation

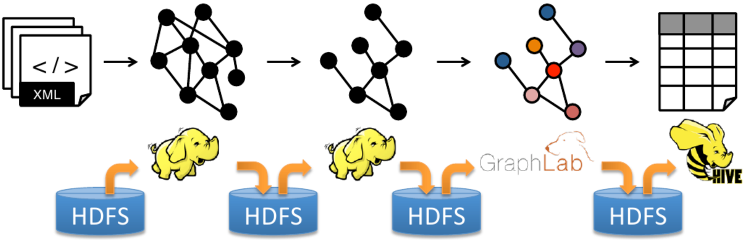
- Graph-parallel computation: restricting the types of computation to achieve performance.

Data-Parallel vs. Graph-Parallel Computation

- Graph-parallel computation: restricting the types of computation to achieve performance.
- But, the same restrictions make it difficult and inefficient to express many stages in a typical graph-analytics pipeline.

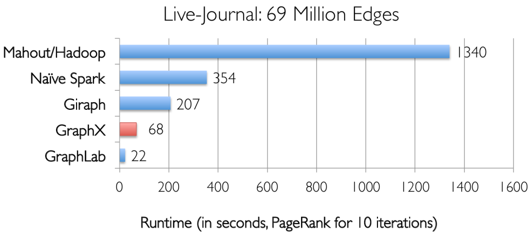


Data-Parallel and Graph-Parallel Pipeline

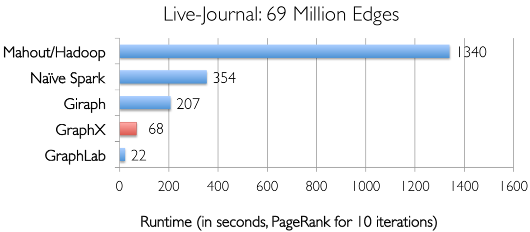


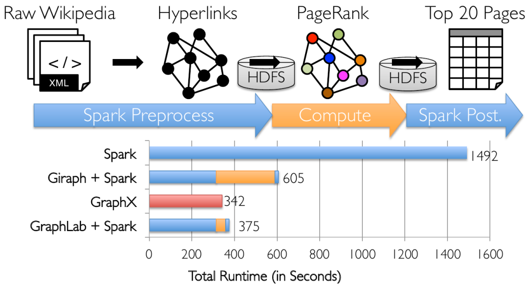
- Moving between table and graph views of the same physical data.
- Inefficient: extensive data movement and duplication across the network and file system.

GraphX vs. Data-Parallel/Graph-Parallel Systems



GraphX vs. Data-Parallel/Graph-Parallel Systems



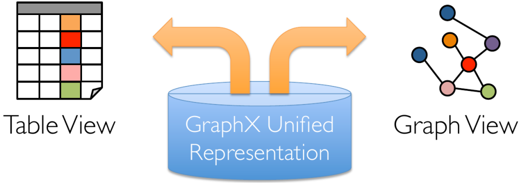


GraphX

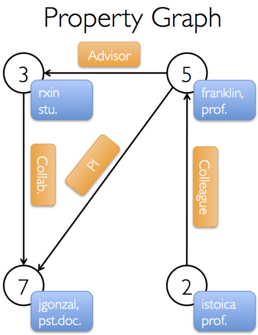
- New API that blurs the distinction between Tables and Graphs.
- New system that unifies Data-Parallel and Graph-Parallel systems.
- It is implemented on top of Spark.

Unifying Data-Parallel and Graph-Parallel Analytics

- Tables and Graphs are composable views of the same physical data.
- Each view has its own operators that exploit the semantics of the view to achieve efficient execution.



Data Model



Vertex Table

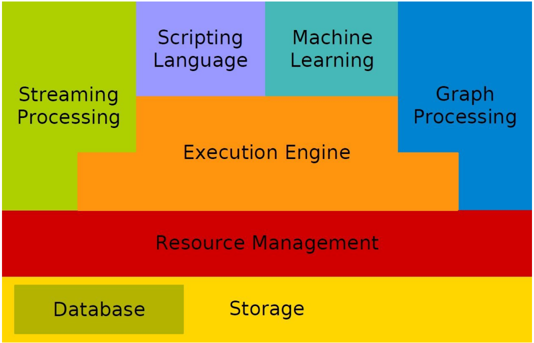
Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Summary

Summary



Streaming Processing
Scripting Language
Machine Learning
Graph Processing
Execution Engine
Resource Management
Database
Storage

Questions?