CS 644: Introduction to Big Data

Daqing Yun (daqing.yun@njit.edu)
New Jersey Institute of Technology

https://www.alcf.anl.gov/theta

Outline

- Big Data Ecosystem
  - Hadoop
  - Pig
  - Spark
  - *Tez*
  - *Storm*
  - *Pig*
  - *Oozie*
  - *HBase*
  - *Hive*

Big Data (Hadoop) Ecosystem

Big Data Applications/Domains
(Healthcare, insurance, finance, social networks, transportation, sciences, etc.)

Big Data Analytics
(Methods: AI, machine learning, visualization, etc. Modules: Pig, Hive, Mahout, etc.)

Big Data Computing
(YARN, MapReduce, Spark, Storm, Oozie, etc.)

Big Data Management
(Structures: Key-Value, Document, Graph, etc. Systems: MongoDB, Hbase, Cassandra, etc.)

Big Data Storage
(HDFS)

Big Data Networking
(HPN, SDN, etc.)

Remind -- Apache Hadoop

- The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.

- This project includes these modules:
  - **Hadoop Common**: The common utilities that support the other Hadoop modules.
  - **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
  - **Hadoop YARN**: A framework for job scheduling and cluster resource management.
  - **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

http://hadoop.apache.org

Hadoop-related Apache Projects

- **Ambari™**: A web based tool for provisioning, managing, and monitoring Hadoop clusters. It also provides a dashboard for viewing cluster health and ability to view MapReduce, Pig, and Hive applications visually.
- **Avro™**: A data serialization system.
- **Cassandra™**: A scalable multi-master database with no single points of failure.
- **Chukwa™**: A data collection system for managing large distributed systems.
- **HBase™**: A scalable, distributed database that supports structured data storage for large tables.
- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying
- **Mahout™**: A scalable machine learning and data mining library.
- **Pig™**: A high-level data-flow language and execution framework for parallel computation
- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.
- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases.
- **ZooKeeper™**: A high-performance coordination service for distributed applications.

YARN

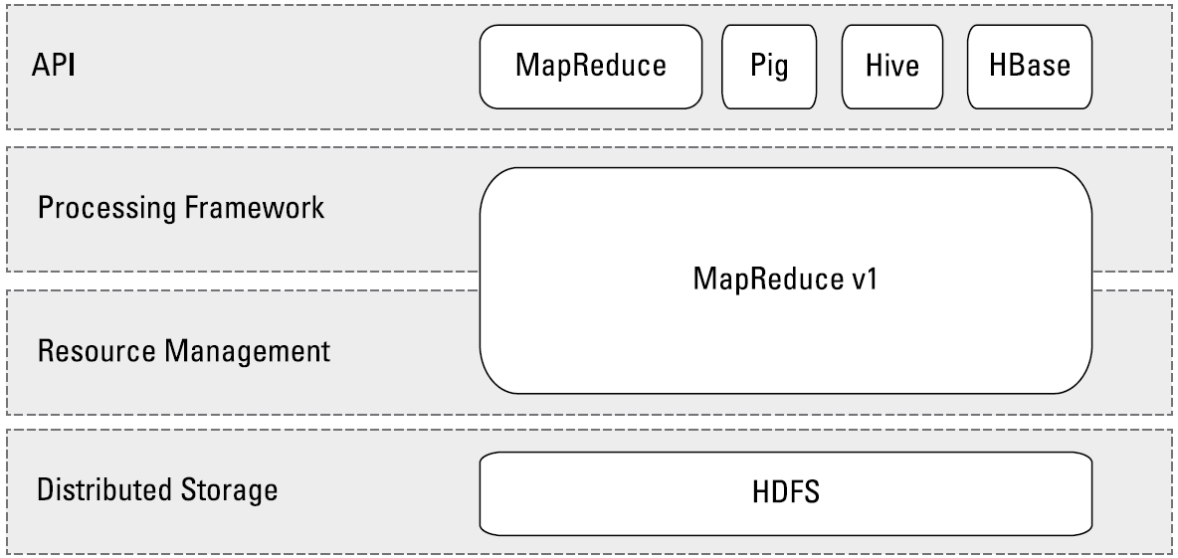
- YARN – Yet Another Resource Negotiator
  - A tool that enables the other processing frameworks to run on Hadoop
  - A general-purpose resource management facility that can schedule and assign CPU cycles and memory (and in the future, other resources, such as network bandwidth) from the Hadoop cluster to waiting applications.

YARN has converted Hadoop from simply a batch processing engine into a platform for many different modes of data processing, from traditional batch to interactive queries to streaming analysis.

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

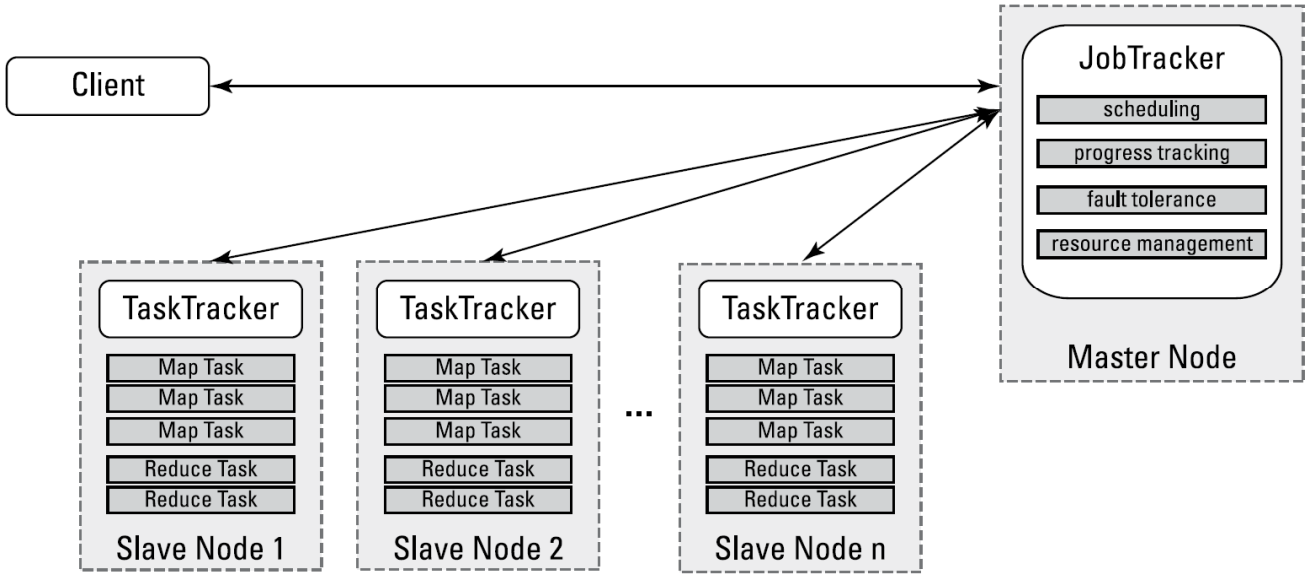
Four distinctive layers of Hadoop

- **Distributed storage:** The Hadoop Distributed File System (**HDFS**) is the storage layer where the data, interim results, and final result sets are stored.
- **Resource management:** In addition to disk space, all slave nodes in the Hadoop cluster have CPU cycles, RAM, and network bandwidth. A system such as Hadoop needs to be able to parcel out these resources so that multiple applications and users can **share the cluster in predictable and tunable ways**. This job is done by the JobTracker daemon.
- **Processing framework:** The MapReduce process flow defines the execution of all applications in Hadoop 1. This begins with the **map phase**; continues with aggregation with **shuffle, sort, or merge**; and ends with the **reduce phase**. In Hadoop 1, this is managed by the JobTracker daemon, with local execution being managed by TaskTracker daemons running on the slave nodes.
- **Application Programming Interface (API):** Applications developed for Hadoop 1 needed to be coded using the MapReduce API. In Hadoop 1, the Hive and Pig projects provide programmers with **easier interfaces** for writing Hadoop applications, and underneath the hood, their code compiles down to MapReduce.



D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

Hadoop 1 execution



1. The client application submits an application request to the JobTracker.
2. The JobTracker determines how many processing resources are needed to execute the entire application.
3. The JobTracker looks at the state of the slave nodes and queues all the map tasks and reduce tasks for execution.
4. As processing slots become available on the slave nodes, map tasks are deployed to the slave nodes. Map tasks are assigned to nodes where the same data is stored.
5. The TaskTracker monitors task progress. If failure, the task is restarted on the next available slot.
6. After the map tasks are finished, reduce tasks process the interim results sets from the map tasks.
7. The result set is returned to the client application.

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

Limitation of Hadoop 1

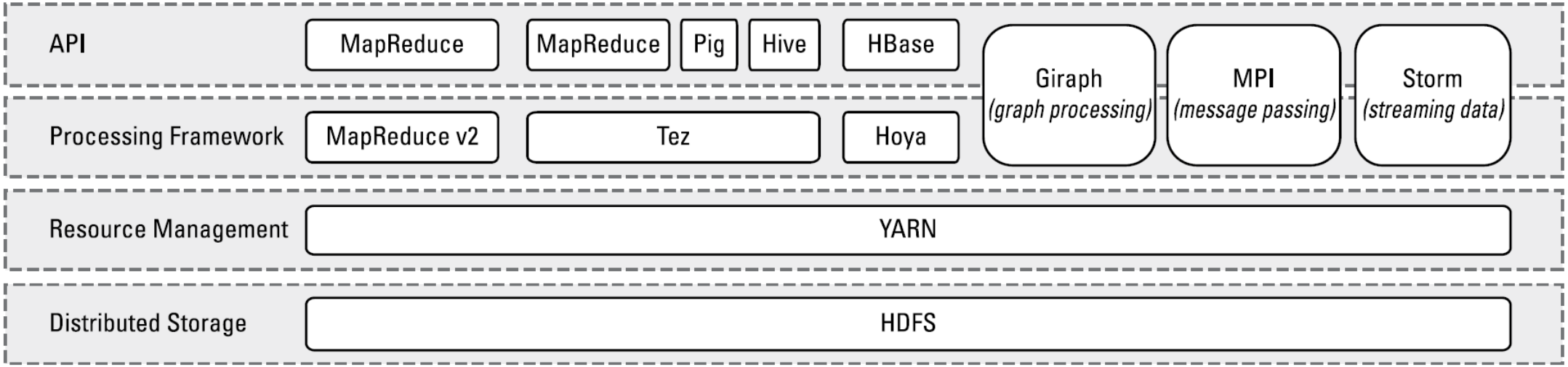
- MapReduce is a successful **batch-oriented** programming model.
- A glass ceiling in terms of wider use:
  - Exclusive tie to MapReduce, which means it could be used only for batch-style workloads and for general-purpose analysis.
- Triggered demands for additional processing modes:
  - Stream data processing
  - Message passing
  - Graph analysis

- Demand is growing for real-time and ad-hoc analysis
- Analysts ask many smaller questions against subsets of data and need a near-instant response.
- Some analysts are more used to SQL & Relational databases

YARN was created to move beyond the limitation of a Hadoop 1 / MapReduce world.

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

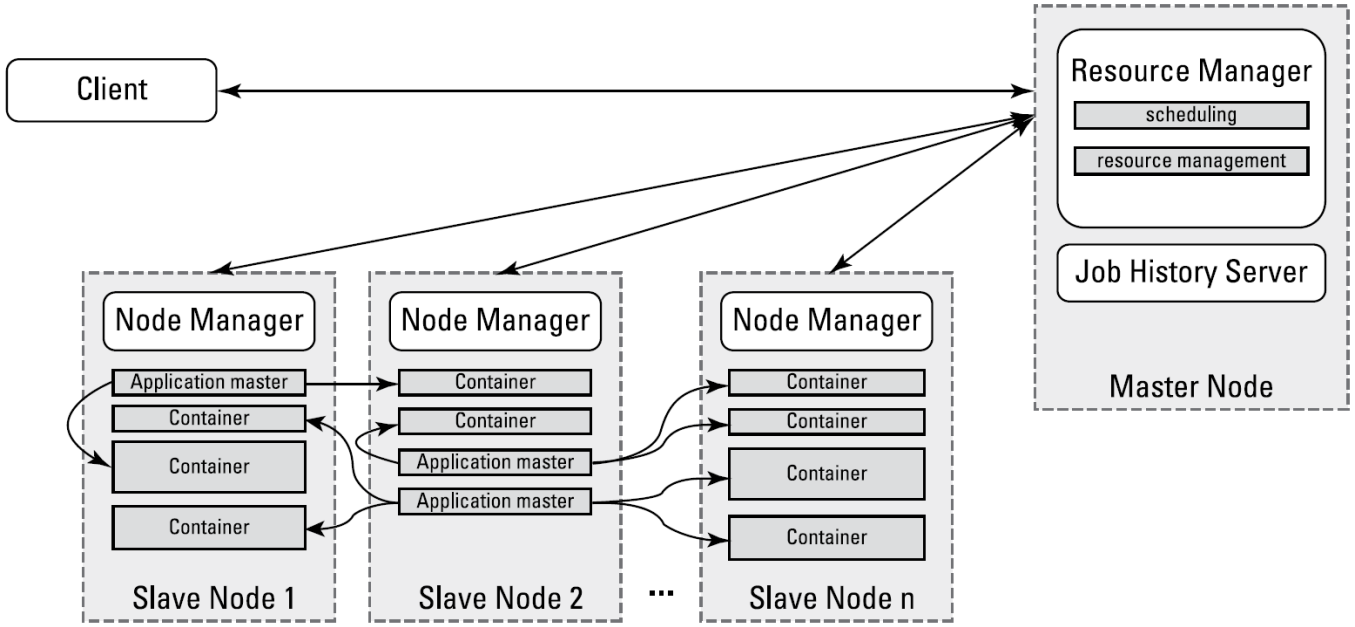
Hadoop 2 Data Processing Architecture with YARN



- **Distributed storage:** Nothing has changed here with the shift — **HDFS** is still the storage layer for Hadoop.
- **Resource management:** The key underlying concept in the shift to YARN from Hadoop 1 is **decoupling resource management from data processing**. This enables YARN to provide resources to any processing framework written for Hadoop, including MapReduce.
- **Processing framework:** Because YARN is a **general-purpose** resource management facility, it can **allocate cluster resources** to any data processing framework written for Hadoop. The processing framework then handles application runtime issues. To maintain compatibility for all the code that was developed for Hadoop 1, MapReduce serves as the first framework available for use on YARN. The Apache Tez project was an incubator project as an alternative framework for the execution of Pig and Hive applications. Tez will likely (if not currently) be viewed as a standard Hadoop configuration.
- **Application Programming Interface (API):** With the support for additional processing frameworks, **support for additional APIs** will come. Hoya (for running HBase on YARN), Apache Giraph (for graph processing), Open MPI (for message passing in parallel systems), Apache Storm (for data stream processing) are in active development.

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

YARN's application execution



1. Client submits an application to Resource Manager.
2. Resource Manager asks a Node Manager to create an Application Master instance and starts up.
3. Application Master initializes itself and register with the Resource Manager
4. Application Master figures out how many resources are needed to execute the application.
5. Application Master then requests the necessary resources from the Resource Manager. It sends heartbeat message to the Resource Manager throughout its lifetime.
6. The Resource Manager accepts the request and queue up.
7. As the requested resources become available on the slave nodes, the Resource Manager grants the Application Master leases for containers on specific slave nodes.

With YARN, you are no longer required to define how many map and reduce slots you need – only need to decide on how much memory tasks can have.

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

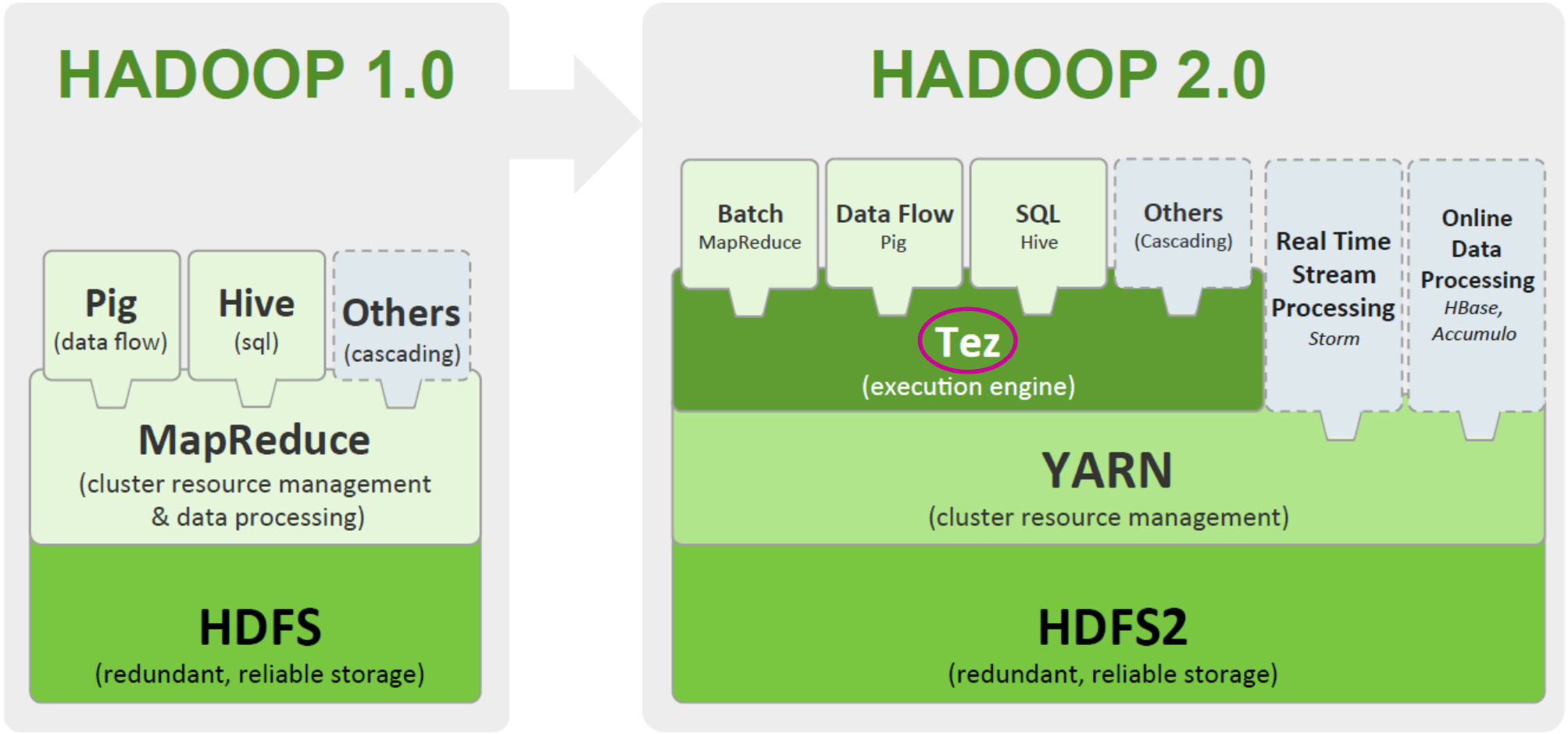
Apache Tez

Monolithic

- Resource management
- Execution engine
- User API

Layered

- Resource management -- YARN
- Execution engine -- Tez
- User API – Hive, Pig, Your App!



Source: Hortonworks

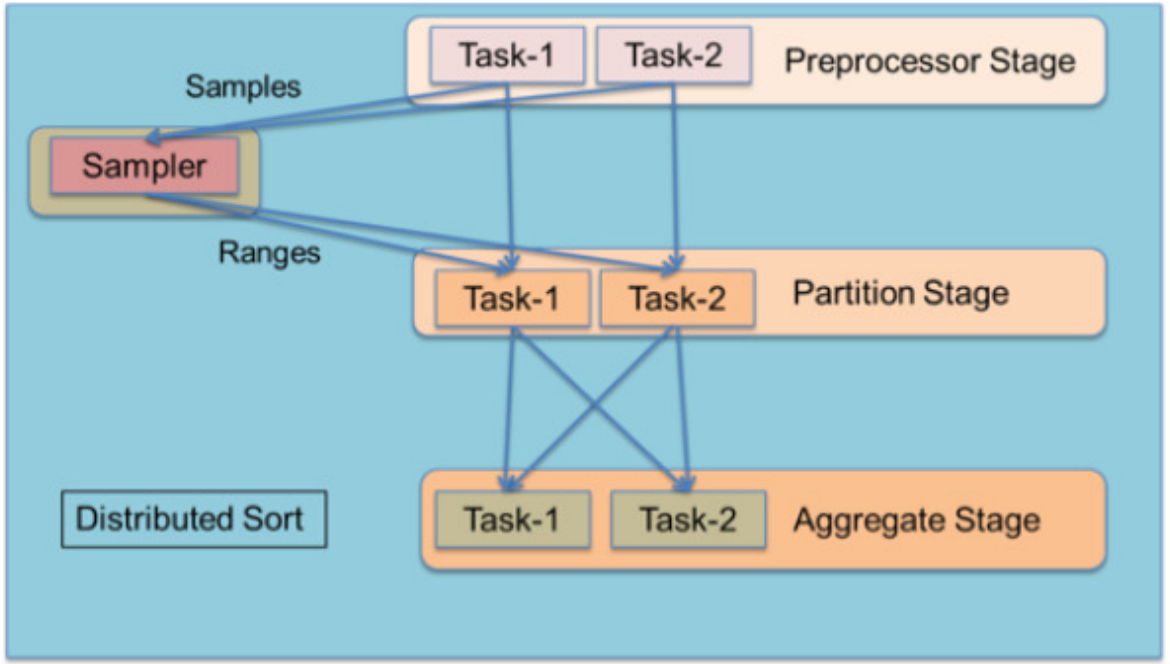
Apache Tez

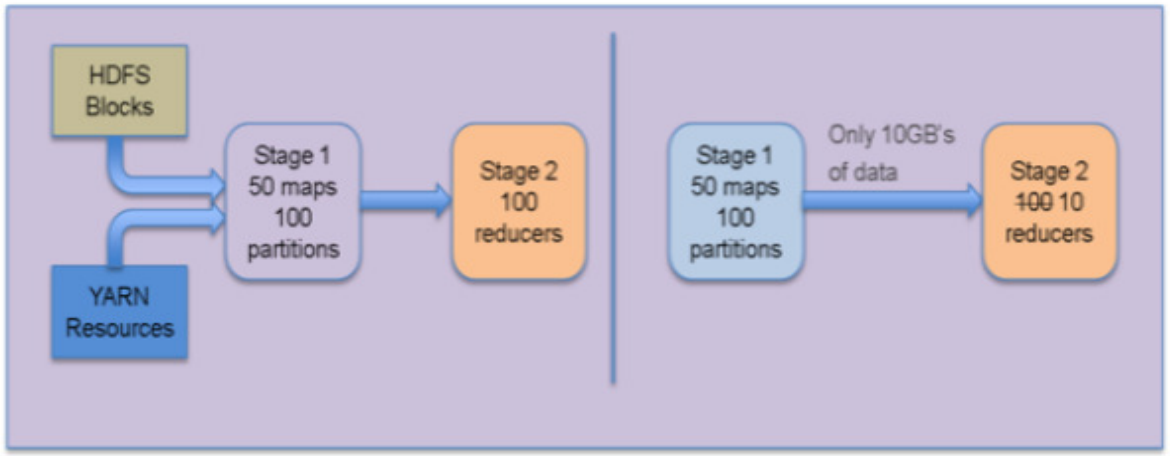
- Tez is an **orchestration** framework for **containers** running on Apache YARN.
  - It is primarily used as a **back-end** for tools such as Apache Hive and Apache Pig to run logic in parallel against potentially large datasets.
- Tez offers a **customizable** execution architecture
  - Tez allows its client application **express complex computations as dataflow graphs** where the vertices are “processing logic” and the edges are dataflows.
  - Reducing overheads and queuing effects
  - Gives system the global picture for better planning
- Tez achieves better execution **performance**
  - Leverages the resources of cluster efficiently, e.g., dynamic performance optimizations based on **real information** about the data and the resources required
  - Customizable engine to let applications tailor the job to meet their specific requirements
- Tez is an extensible framework for building high performance batch and **interactive** data processing applications, coordinated by YARN in Apache Hadoop.
  - Tez uses “**sessions**” to address the issues of latency when using interactive system to execute queries

https://hortonworks.com/apache/tez/

Tez's characteristics

- Dataflow graph with vertices to express, model, and execute data processing logic
- Performance via Dynamic Graph Reconfiguration
- Flexible Input-Processor-Output task model
- Optimal Resource Management





Resources

- Apache Tez
  - https://tez.apache.org/

- Apache Tez Overview
  - https://hortonworks.com/apache/tez/
  - http://moi.vonos.net/bigdata/tez/

- Apache Tez AWS tutorial
  - https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-tez.html

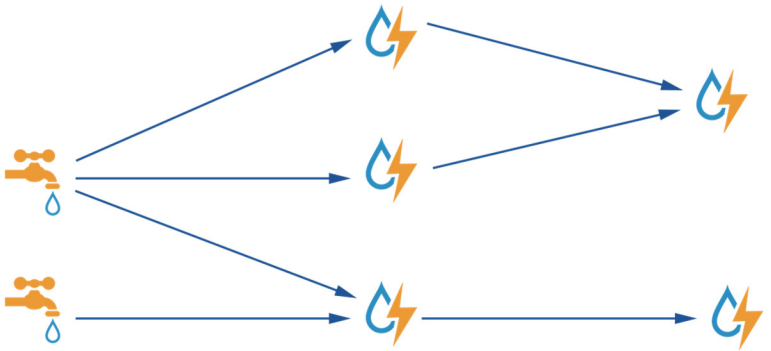
- Evaluation paper
  - Singh *et al*. Analyzing performance of Apache Tez and MapReduce with hadoop multinode cluster on Amazon cloud. *Journal of Big Data*, vol. 3, no. 19, 2016.

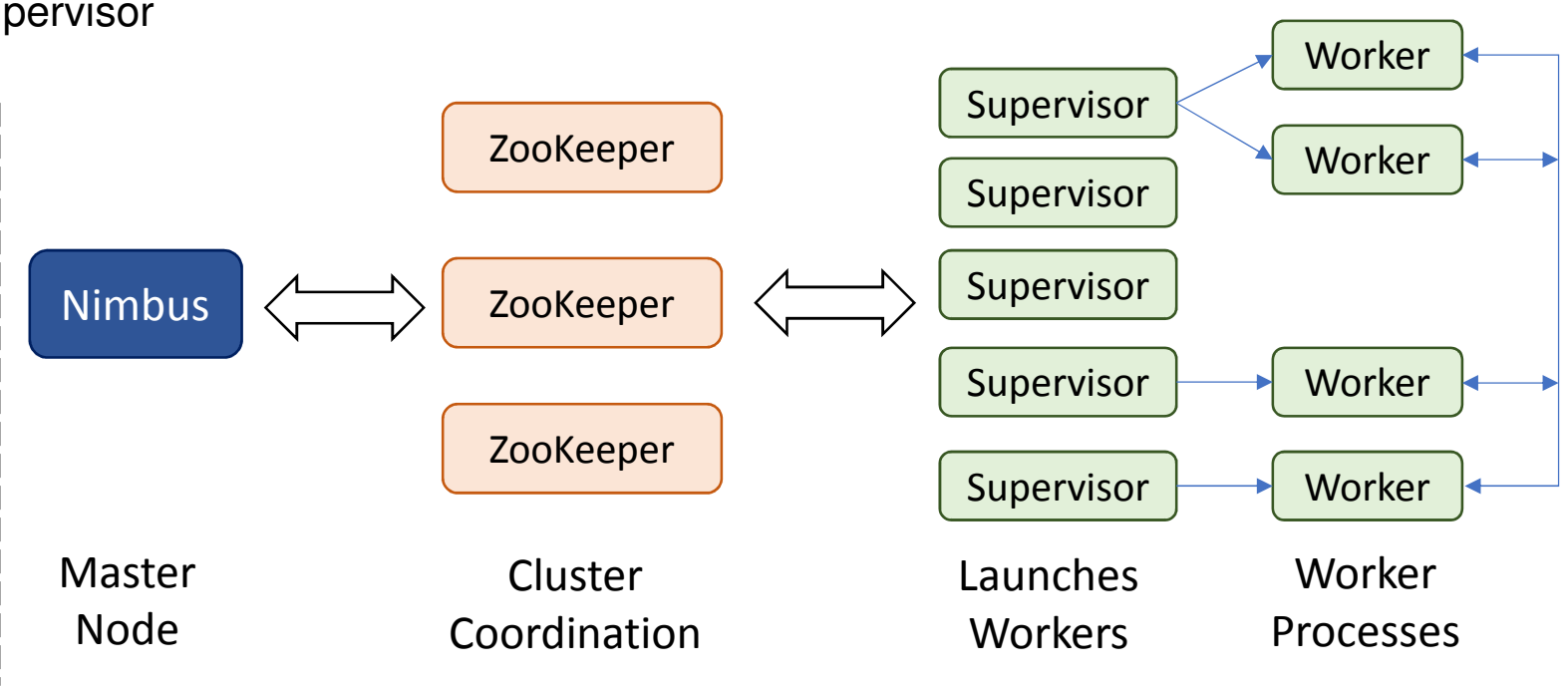
Apache Storm

Stream Processing

- On Hadoop, you run MapReduce jobs
- On Storm, you run Topologies
- Two kinds of nodes on a Storm cluster:
  - The master node runs “Nimbus”
  - The worker nodes called the Supervisor

- The master node of Storm runs a daemon called “Nimbus”. Nimbus is responsible for distributing codes, assigning tasks to machines and monitoring their performance.
- The worker node also runs a daemon called “Supervisor” which is able to run one or more worker processes on its node.
  - Each supervisor works assigned by Nimbus and starts and stops the worker processes when required.
  - Every worker process runs a specific set of topology which consists of worker processes working around machines.
  - Since Apache Storm does not have the abilities to manage its cluster state, it depends on Apache Zookeeper for this purpose.
  - Zookeeper facilitates communication between Nimbus and Supervisors with the help of message acknowledgements, processing status, etc.

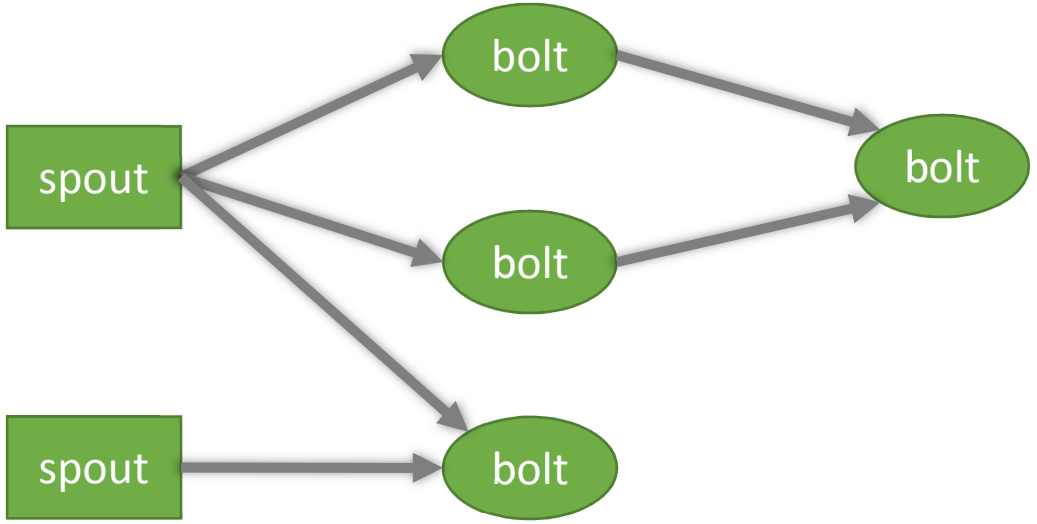




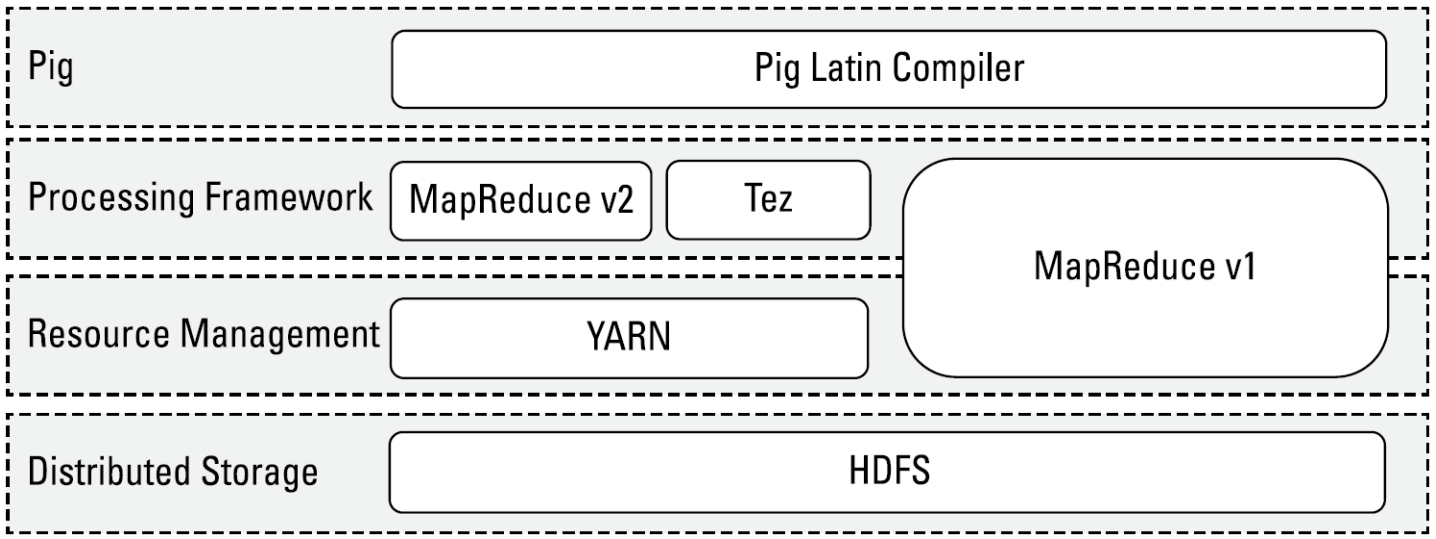
How Storm processes data?

- Tuples -- an ordered list of elements, e.g., a “4-tuple” might be (7, 1, 3, 7)
- Streams -- an unbounded sequence of tuples
- Spouts -- sources of streams in a computation (e.g., a Twitter API)
- Bolts -- process input streams and produce output streams. They can: run functions; filter, aggregate, or join data; or talk to databases.
- Topologies -- overall calculation, represented visually as a network of spouts and bolts (as in the following diagram)

- A Storm application is designed as a “topology” in the shape of a DAG with spouts and bolts acting as the graph vertices.
- Edges on the graph are named streams and direct data from one node to another. Together, the topology acts as a data transformation pipeline.



Apache Pig



- At its core, Pig Latin is a ***dataflow*** language, where you define a data stream and a series of transformations that are applied to the data as it flows through your application.
- This is in contrast to a ***control flow*** language (like C or Java), where you write a series of instructions.
- In control flow languages, we use constructs like loops and conditional logic (like an if statement). You won't find loops and if statements in Pig Latin.

The language itself:
The programming language for Pig is known as Pig Latin, a high-level language that allows you to write data processing and analysis programs.

The Pig Latin compiler:
The Pig Latin compiler converts the Pig Latin code into executable code. The executable code is either in the form of MapReduce jobs or it can spawn a process where a virtual Hadoop instance is created to run the Pig code on a single node.

The sequence of MapReduce programs enables Pig programs to do data processing and analysis in parallel, leveraging Hadoop MapReduce and HDFS. Running the Pig job in the virtual Hadoop instance is a useful strategy for testing your Pig scripts.

Pig example -- Word Count

```
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS (line:chararray);
```

Extract words from each line and put them into a pig bag datatype, then flatten the bag to get one word on each row

```
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;
```

Filter out any words that are just white spaces

```
filtered_words = FILTER words BY word MATCHES '\\w+';
```

Create a group for each word

```
word_groups = GROUP filtered_words BY word;
```

Count the entries in each group

```
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;
```

Order the records by count

```
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

Example source: https://en.wikipedia.org/wiki/Apache_Pig

Pig example -- calculating the total miles flown

```
records = LOAD '2013_subset.csv' USING PigStorage(',') AS
      (Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,
      CRSArrTime,UniqueCarrier,FlightNum,TailNum,ActualElapsedTime,
      CRSElapsedTime,AirTime,ArrDelay,DepDelay,Origin,Dest,
      Distance:int,TaxiIn,TaxiOut,Cancelled,CancellationCode,Diverted,
      CarrierDelay,WeatherDelay,NASDelay,SecurityDelay,LateAircraftDelay);

milage_recs = GROUP records ALL;

tot_miles = FOREACH milage_recs GENERATE SUM(records.Distance);

DUMP tot_miles;
```

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

Characteristics of Pig

- **Most Pig scripts start with the `LOAD` statement to read data from HDFS.** In this case, we're loading data from a `.csv` file. Pig has a data model it uses, so next we need to map the file's data model to the Pig data mode. This is accomplished with the help of the `USING` statement. We then specify that it is a comma-delimited file with the `PigStorage(',')` statement followed by the `AS` statement defining the name of each of the columns.

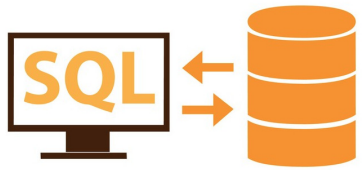
- **Aggregations are commonly used in Pig to summarize data sets.** The `GROUP` statement is used to aggregate the records into a single record `mileage_recs`. The `ALL` statement is used to aggregate all tuples into a single group. Note that some statements -- including the following `SUM` statement -- requires a preceding `GROUP ALL` statement for global sums.

- **`FOREACH . . . GENERATE` statements are used here to transform columns data.** In this case, we want to count the miles traveled in the `records_Distance` column. The `SUM` statement computes the sum of the `record_Distance` column into a single-column collection `total_miles`.

- **The `DUMP` operator is used to execute the Pig Latin statement and display the results on the screen.** `DUMP` is used in interactive mode, which means that the statements are executable immediately and the results are not saved. Typically, you will either use the DUMP or STORE operators at the end of your Pig script.

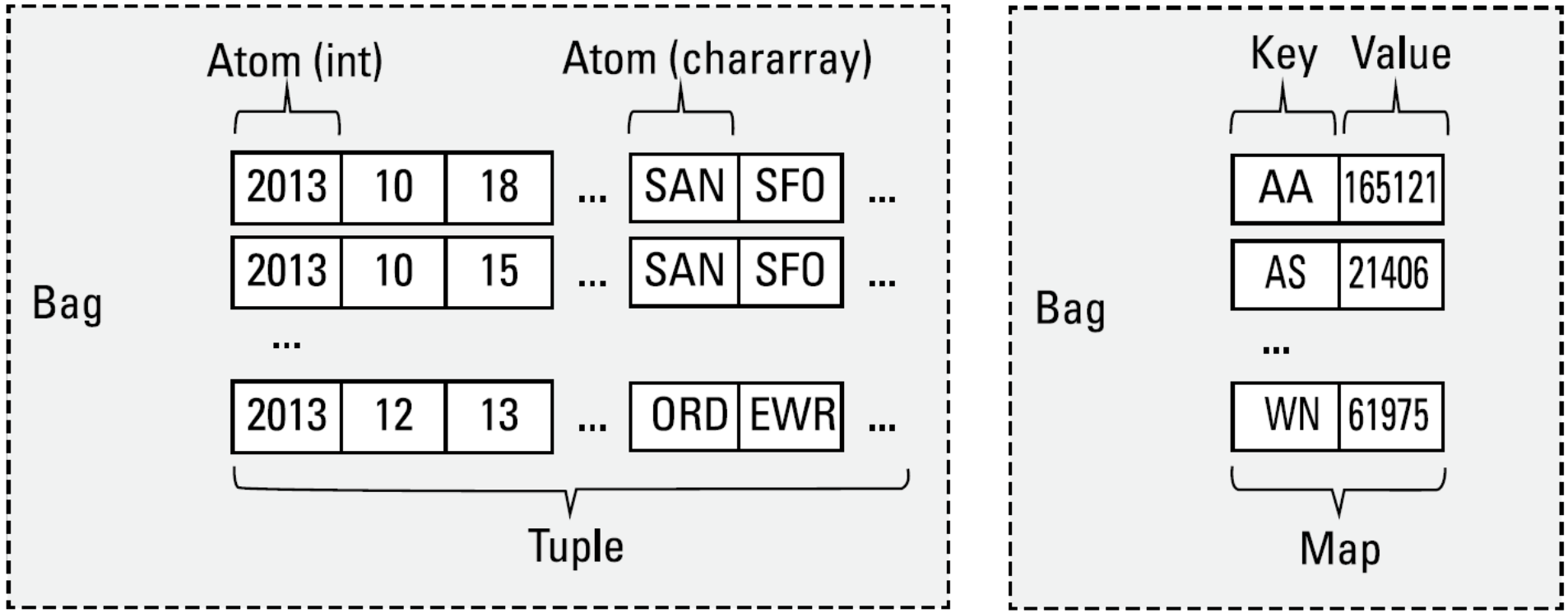
D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

Pig vs. SQL



- In comparison with SQL, Pig
  - uses lazy evaluation
  - uses ETL (extract-transform-load)
  - is able to store data at any point during a pipeline
  - declares execution plans
  - supports pipeline splits
- Pig Latin is procedural and fits very naturally in the pipeline paradigm while SQL is instead declarative
  - In SQL users can specify that data from two tables must be joined, but not what join implementation to use
  - Pig Latin allows users to specify an implementation or aspects of an implementation to be used in executing a script in several ways
- Pig Latin programming is similar to specifying a query execution plan and Pig Latin script describes a DAG rather than a pipeline
  - SQL is oriented around queries that produce a single result. SQL handles trees naturally, but has no built in mechanism for splitting a data processing stream and applying different operators to each sub-stream
- Pig Latin's ability to include user code at any point in the pipeline is useful for pipeline development
  - If SQL is used, data must first be imported into database, and then the cleansing and transformation process can begin

Pig Data Types and Syntax



- **Atom:** An atom is any single value, such as a string or number
- **Tuple:** A tuple is a record that consists of a sequence of fields. Each field can be of any type.
- **Bag:** A bag is a collection of non-unique tuples.
- **Map:** A map is a collection of key value pairs.

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

Pig Latin Operator

Operation	*Operator*	*Explanation*
Data Access	LOAD/STORE	Read and Write data to file system
	DUMP	Write output to standard output (stdout)
	STREAM	Send all records through external binary
	FOREACH	Apply expression to each record and output one or more records
	FILTER	Apply predicate and remove records that don't meet condition
	GROUP/ COGROUP	Aggregate records with the same key from one or more inputs
	JOIN	Join two or more records based on a condition
Transformations	CROSS	Cartesian product of two or more inputs
	ORDER	Sort records based on key
	DISTINCT	Remove duplicate records
	UNION	Merge two data sets
	SPLIT	Divide data into two or more bags based on predicate
	LIMIT	subset the number of records

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

Pig Latin expressions

- In Pig Latin, expressions are language constructs used with the FILTER, FOREACH, GROUP, and SPLIT operators as well as the eval functions.

- Expressions are written in conventional mathematical infix notation and are adapted to the UTF-8 character set. Depending on the context, expressions can include:
  - Any Pig data type (simple data types, complex data types)
  - Any Pig operator (arithmetic, comparison, null, boolean, dereference, sign, and cast)
  - Any Pig built in function.
  - Any user defined function (UDF) written in Java.

- In Pig Latin,
  - An arithmetic expression could look like this:

```
X = GROUP A BY f2*f3;
```

  - A string expression could look like this, where a and b are both chararrays:

```
X = FOREACH A GENERATE CONCAT(a,b);
```

  - A boolean expression could look like this:

```
X = FILTER A BY (f1==8) OR (NOT (f2+f3 > f1));
```

https://pig.apache.org/docs/latest/basic.html#expressions

Pig User-Defined Functions (UDFs)

Eval is the most common type of function. It can be used in `FOREACH` statements as shown in this script →

```
-- myscript.pig
REGISTER myudfs.jar;
A = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

Command to run the script →

```
pig -x local myscript.pig
```

The .java contains the implementation of the `UPPER` UDF →

```
1  package myudfs;
2  import java.io.IOException;
3  import org.apache.pig.EvalFunc;
4  import org.apache.pig.data.Tuple;
5
6  public class UPPER extends EvalFunc<String>
7  {
8    public String exec(Tuple input) throws IOException {
9        if (input == null || input.size() == 0 || input.get(0) == null)
10           return null;
11       try{
12           String str = (String)input.get(0);
13          return str.toUpperCase();
14       }catch(Exception e){
15           throw new IOException("Caught exception processing input row ", e);
16       }
17   }
18 }
```

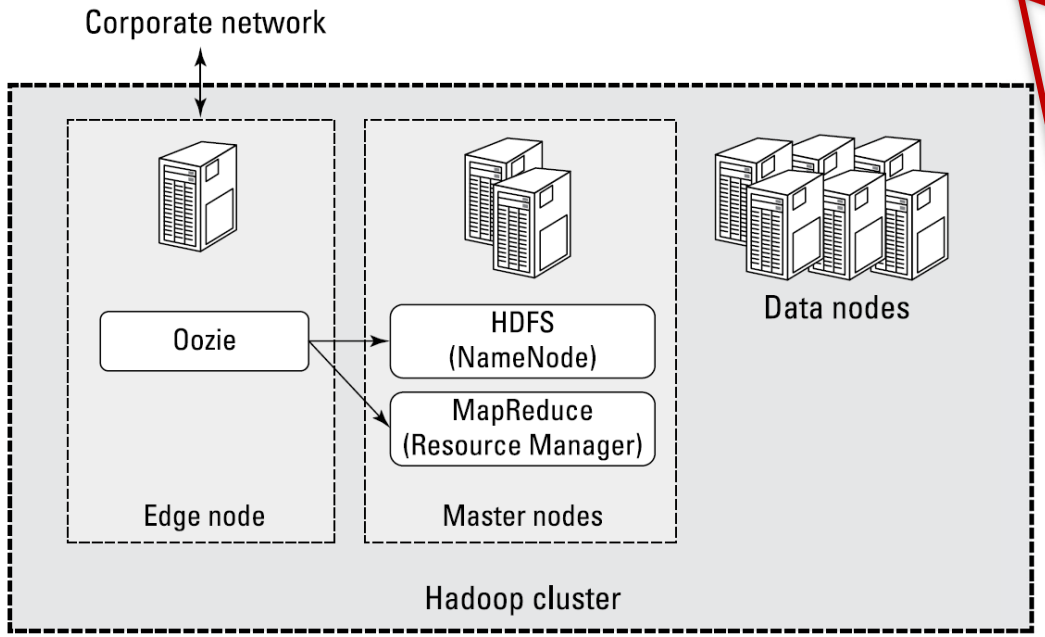
For example, if the data is stored in a file format that is not natively accessible to Pig, you can optionally add the USING function to the LOAD statement to specify a UDF that can read in (and interpret) the data.

https://pig.apache.org/docs/latest/udf.html

Apache Oozie Workflow Scheduler for Hadoop

- Oozie supports a wide range of job types, including Pig, Hive, and MapReduce, as well as jobs coming from Java programs and Shell scripts.

Each workflow XML file must have one matched pair of start and end nodes.




```
<workflow-app name="SampleWorkflow" xmlns="uri:oozie:workflow:0.1">
  <start to="firstJob"/>
  <action name="firstJob">
    <pig>...</pig>
    <ok to="secondJob"/>
    <error to="kill"/>
  </action>
  <action name="secondJob">
    <map-reduce>...</map-reduce>
    <ok to="end" />
    <error to="kill" />
  </action>
  <end name="end"/>
  <kill name="kill">
    <message>"Killed job."</message>
  </kill>
</workflow-app>
```

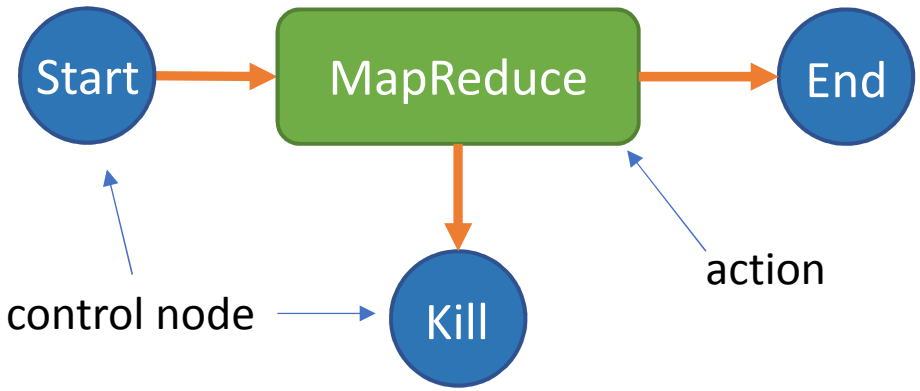
first job

second job

Sample Oozie XML file

Oozie workflows can include kill nodes, which are a special kind of node dedicated to handling error conditions.

Action and Control Nodes




```
<workflow-app name="foo-wf"..
<start to="[NODE-NAME]"/>
<map-reduce>
...
...
</map-reduce>
<kill name="[NODE-NAME]">
<message>Error occurred</message
</kill>
<end name="[NODE-NAME]"/>
</workflow-app>
```

- **Control flow**
  - start, end, kill
  - decision
  - fork, join
- **Action**
  - map-reduce
  - java
  - pig
  - hive
  - hdfs

- Workflows begin with START node
- Workflows succeed with END node
- Workflows fail with KILL node
- Several actions support JSP Expression Language (EL)

- Oozie Coordination Engine can **trigger** workflows by
  - Time (Periodically)
  - Data Availability (Data appears in a directory)

Schedule Workflow by Time

```
<coordinator-app name="sampleCoordinator"
                 frequency="${coord:days(1)}"
                 start="2014-06-01T00:01Z "
                 end="2014-06-01T01:00Z "
                 timezone="UTC"
                 xmlns="uri:oozie:coordinator:0.1">
   <controls>...</controls>
   <action>
      <workflow>
         <app-path>${workflowAppPath}</app-path>
      </workflow>
   </action>
</coordinator-app>
```

Schedule Workflow by Time and Data Availability

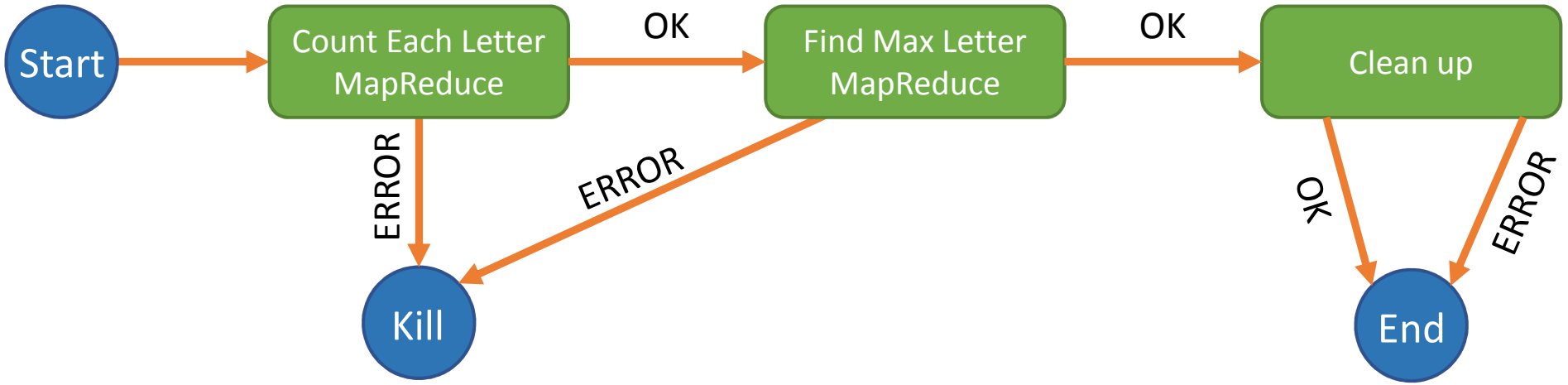
```
<coordinator-app name="sampleCoordinator"
                 frequency="${coord:days(1)}"
                 start="${startTime}"
                 end="${endTime}"
                 timezone="${timeZoneDef}"
                 xmlns="uri:oozie:coordinator:0.1">
   <controls>...</controls>
   <datasets>
      <dataset name="input" frequency="${coord:days(1)}" initial-
               instance="${startTime}" timezone="${timeZoneDef}">
         <uri-template>${triggerDatasetDir}</uri-template>
      </dataset>
   </datasets>
   <input-events>
         <data-in name="sampleInput" dataset="input">
         <instance>${startTime}</instance>
      </data-in>
   </input-events>
   <action>
      <workflow>
         <app-path>${workflowAppPath}</app-path>
      </workflow>
   </action>
</coordinator-app>
```

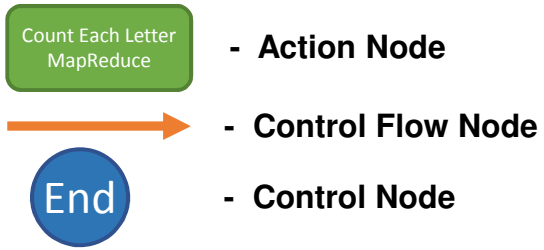
Running Oozie examples

- **Extract examples packaged with Oozie**
  - $ cd $OOZIE_HOME
  - $ tar xvf oozie-examples.tar.gz
- **Copy examples to HDFS from user's home directory**
  - $ hdfs dfs -put examples examples
- **Run an example**
  - $ oozie job -config examples/apps/map-reduce/job.properties -run
- **Check Web Console**
  - http://localhost:11000/oozie/

https://oozie.apache.org/docs/4.0.1/DG_Examples.html

An example workflow





This source is in *HadoopSamples* project under `/src/main/resources/mr/workflows`

Workflow definition

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="most-seen-letter">
   <start to="count-each-letter"/>
   <action name="count-each-letter">
      <map-reduce>
         <job-tracker>${jobTracker}</job-tracker>
         <name-node>${nameNode}</name-node>
         <prepare>
            <delete path="${nameNode}${outputDir}"/>
            <delete path="${nameNode}${intermediateDir}"/>
         </prepare>
         <configuration>
            ...
            <property>
               <name>mapreduce.job.map.class</name>
               <value>mr.wordcount.StartsWithCountMapper</value>
            </property>
            ...
         </configuration>
      </map-reduce>
      <ok to="find-max-letter"/>
      <error to="fail"/>
   </action>
   ...
```

START Action Node to count-each-letter MapReduce action

MapReduce have optional Prepare section

Pass property that will be set on MapReduce job's Configuration object

In case of success, go to the next job; in case of failure go to fail node

Package and Run your workflow

1. Create application directory structure with workflow definitions and resources
   - workflow.xml, jars, etc.
2. Copy application directory to HDFS
3. Create application configuration file
   - specify location of the application directory on HDFS
   - specify location of the namenode and resource manager
4. Submit workflow to Oozie
   - Utilize oozie command line
5. Monitor running workflow(s)
   - Two options
     - Command line ($oozie)
     - Web Interface (http://localhost:11000/oozie)

Oozie Application Directory

- Must comply to directory structure spec
- Libraries should be placed under lib directory
- workflow.xml defines workflow

```
mostSeenLetter-oozieWorkflow
|--lib/
|     |--HadoopSamples.jar
|--workflow.xml
```

Application Workflow Root

Oozie installation and configuration

- Oozie installation and configuration
  - https://oozie.apache.org/docs/4.2.0/AG_Install.html
- Oozie examples
  - https://oozie.apache.org/docs/4.0.1/DG_Examples.html

HBase

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
Google, Inc.

APACHE HBASE

- **HBase** is modeled after Google's BigTable and written in Java. It is developed on top of Hadoop Distributed File System (HDFS).

- It provides a fault-tolerant way of storing large quantities of sparse data (small amounts of information caught within a large collection of empty or unimportant data, such as finding the 50 largest items in a group of 2 billion records, or finding the non-zero items representing less than 0.1% of a huge collection).

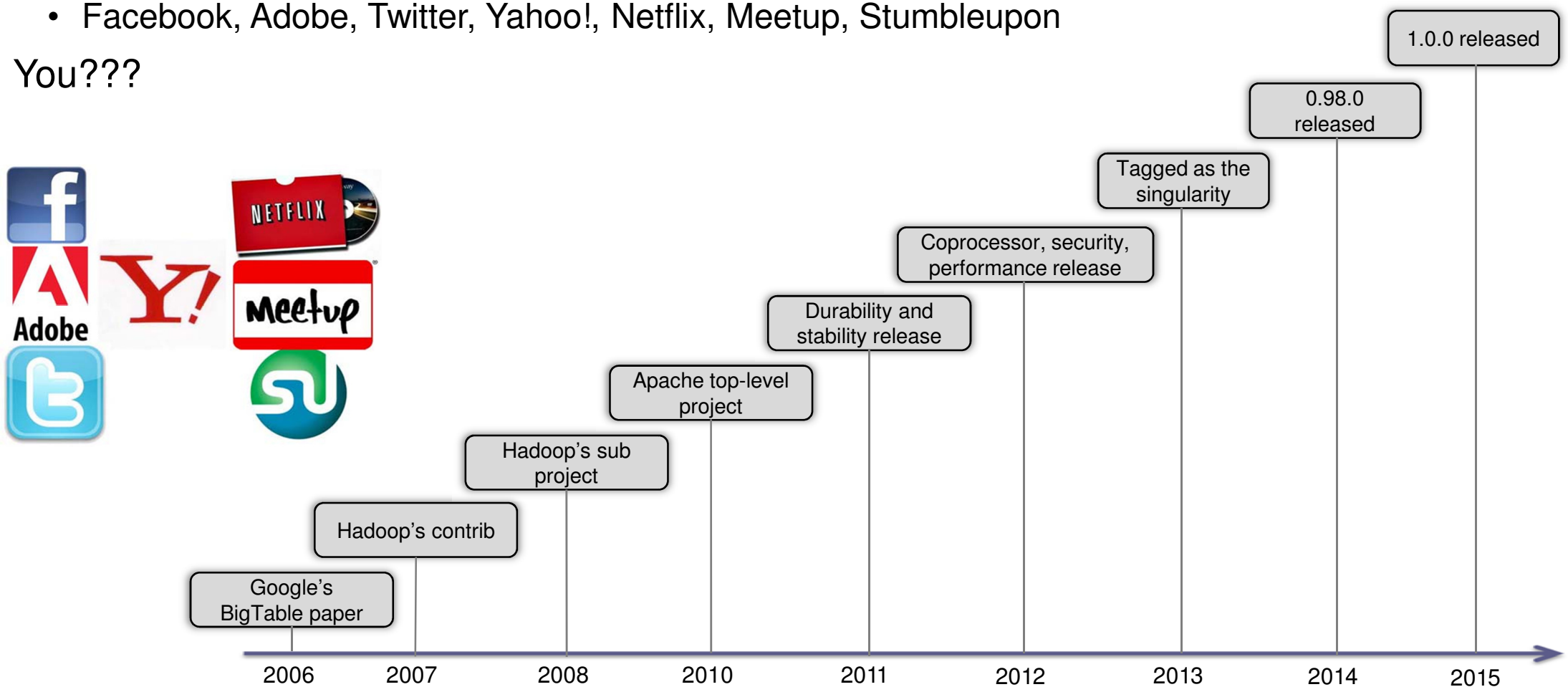
- HBase features compression, in-memory operations, and Bloom filters on a per-column basis.

- An HBase system comprises a set of tables. Each table contains rows and columns, much like a traditional database. Each table must have an element defined as a Primary Key, and all access attempts to HBase tables must use this Primary Key. An HBase column represents an attribute of an object.

HBase History

Who uses HBase?

- Here is a very limited list of well-known names
  - Facebook, Adobe, Twitter, Yahoo!, Netflix, Meetup, Stumbleupon
- You???



Source: Lars, George. HBase The Definitive Guide. O'Reilly Media. 2011

When to use HBase?

- **Not suitable for every problem**
  - Compared to RDBMs has VERY simple and limited API
- **Good for large amounts of data**
  - 100s of millions or billions of rows
  - If data is too small all the records will end up on a single node leaving the rest of the cluster idle
- **Have to have enough hardware!!**
  - At the minimum 5 nodes
    - There are multiple management daemon processes: Namenode, HBaseMaster, Zookeeper, etc.
    - HDFS won't do well on anything under 5 nodes anyway; particularly with a block replication of 3
    - HBase is memory and CPU intensive
- **Carefully evaluate HBase for mixed workloads**
  - Client Request vs. Batch processing (Map/Reduce)
    - SLAs on client requests would need evaluation
  - HBase has intermittent but large I/O access
    - May affect response latency!!!
- **Two well-known use cases**
  - Lots and lots of data (already mentioned)
  - Large amount of clients/requests (usually cause a lot of data)
- **Great for single random selects and range scans by key**
- **Great for variable schema**
  - Rows may drastically differ
  - If your schema has many columns and most of them are null

When NOT to use HBase?

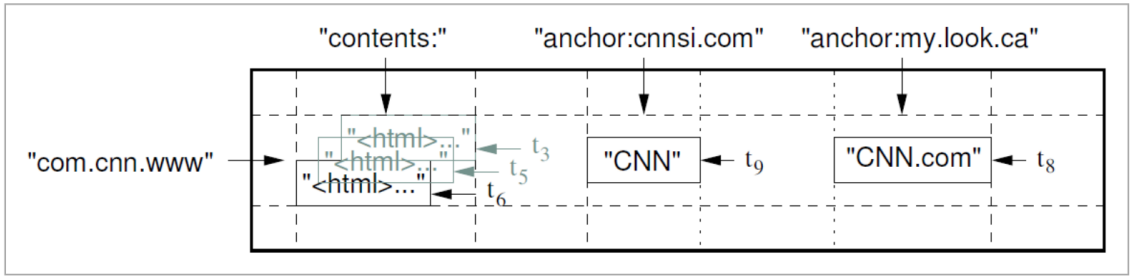
- Bad for traditional RDBMs retrieval
  - Transactional applications
  - Relational analytics
    - *group by*, *join*, and *where column like*, etc....
- Currently bad for text-based search access
  - There is work being done in this area
    - HBasene: https://github.com/akkumar/hbasene/wiki
    - HBASE-3529: 100% integration of HBase and Lucene based on HBase's coprocessors
  - Some projects provide solution that use HBase
    - Lily=HBase+Solr https://ngdata.github.io/hbase-indexer/

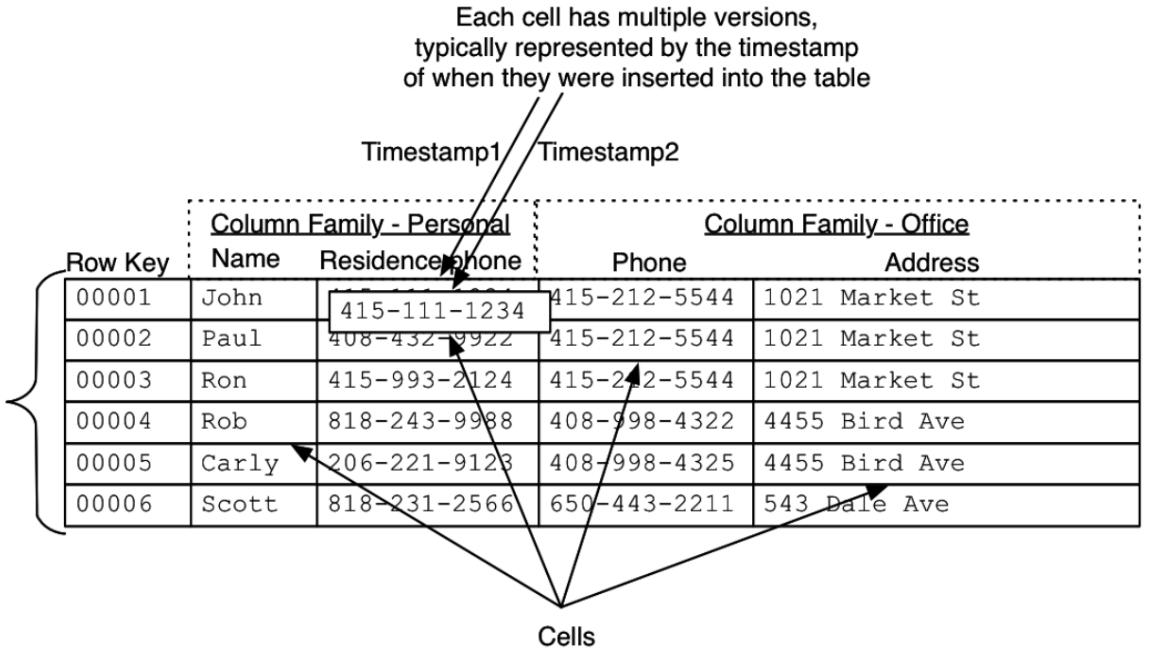
HBase Data Model

- Data is stored in Tables
- Tables contain rows
  - Rows are referenced by a unique key
    - Key is an array of bytes – good news
    - Anything can be a key: string, long and your own serialized data structures
- Rows made of columns, which are grouped in column families
- Data is stored in cells
  - Identified by row x column-family x column
  - Cell's content is also an array of bytes

HBase Families

- Rows are grouped into families
  - Labeled as "family:column"
    - Example "user:first_name"
  - A way to organize your data
  - Various features are applied to families
    - Compression
    - In-memory option
    - Stored together - in a file called HFile/StoreFile
- Family definitions are static
  - Created with table, should be rarely added and changed
  - Limited to small number of families
    - unlike columns that you can have millions of





The table is lexicographically sorted on the row keys

Characteristics of data in HBase

- HBase data stores consist of one or more tables, which are indexed by row keys
- Data is stored in rows with columns, and rows can have multiple versions

Sparse data

Traditional Customer Contact Information Table

Customer ID	*Last Name*	*First Name*	*Middle Name*	*E-mail Address*	*Street Address*
00001	Smith	John	Timothy	John.Smith@xyz.com	1 Hadoop Lane, NY 11111
00002	Doe	Jane	NULL	NULL	7 HBase Ave, CA 22222

Columns are grouped into *column families*, which must be defined up front during table creation and are stored together on disk.

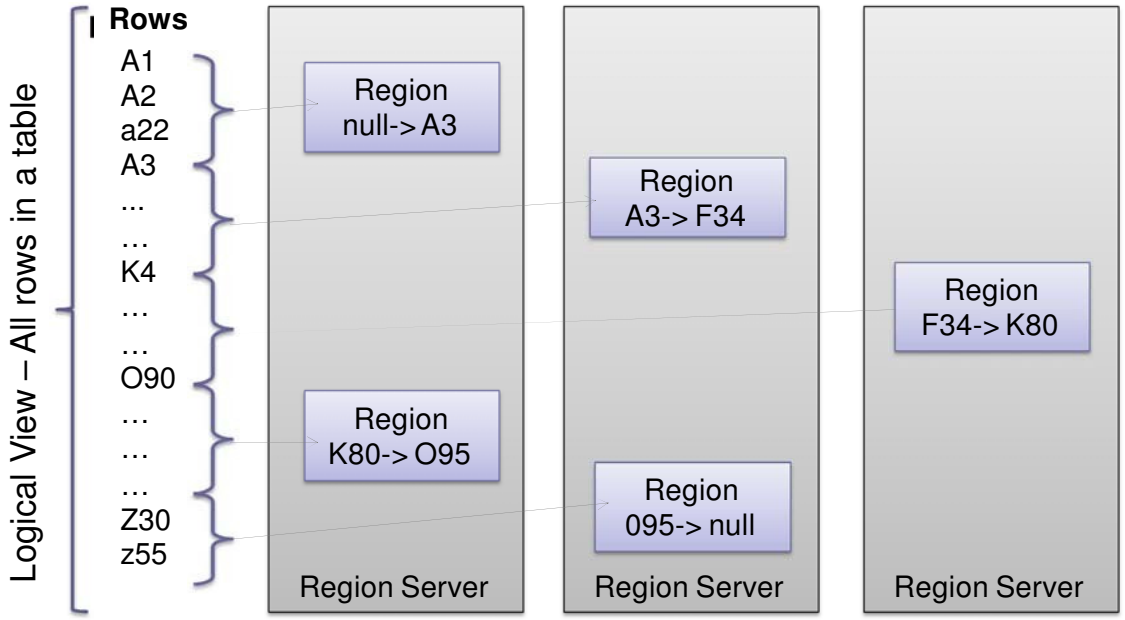
Row Key	*Column Family: {Column Qualifier:Version:Value}*
00001	CustomerName: {'FN': 1383859182496:'John', 'LN': 1383859182858:'Smith', 'MN': 1383859183001:'Timothy', 'MN': 1383859182915:'T'} ContactInfo: {'EA': 1383859183030:'John.Smith@xyz.com', 'SA': 1383859183073:'1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103:'Jane', 'LN': 1383859183163:'Doe', ContactInfo: { 'SA': 1383859185577:'7 HBase Ave, CA 22222'}

- After you specify the key, the rest is optional. The more specific you make the query, however (moving from left to right), the more granular the results.
- There are no data types in HBase, values are just one or more bytes -- you can store anything!

- HDFS lacks **random read and write access**.
- This is where HBase comes into picture.
- It's a **distributed, scalable, big data store**, modeled after Google's BigTable.
- It stores data as key/value pairs.

Rows Distribution Between Region Servers

- **Regions per server depend on hardware specs, with today's hardware it's common to have:**
  - 10 to 1000 regions per Region Server
  - Managing as much as 1GB to 2GB per region
- **Splitting data into regions allows**
  - Fast recovery when a region fails
  - Load balancing when a server is overloaded
    - May be moved between servers
  - Splitting is fast
    - Reads from an original file while asynchronous process performs a split
  - All of these happen automatically without user's involvement

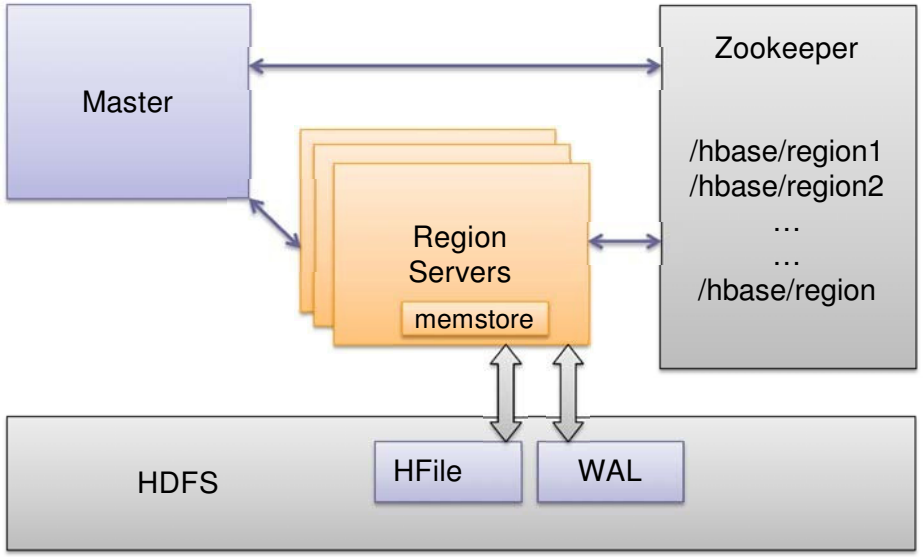


HBase Storage

- Data is stored in files called HFiles/StoreFiles
  - Usually saved in HDFS
- HFile is basically a key-value map
  - Keys are sorted lexicographically
- When data is added, it's written to a log called Write Ahead Log (WAL) and is also stored in memory (memstore)
- Flush: when in-memory data exceeds maximum value, it is flushed to an HFile
  - Data persisted to HFile can then be removed from WAL
  - Region Server continues serving read-writes during the flush operations, writing values to the WAL and memstore
- To control the number of HFiles and to keep cluster well balanced, HBase periodically performs data compactions
  - Minor Compaction: Smaller HFiles are merged into larger HFiles (n-way merge)
    - Fast - Data is already sorted within files
    - Delete markers not applied
  - Major Compaction:
    - For each region merges all the files within a column-family into a single file
    - Scan all the entries and apply all the deletes as necessary

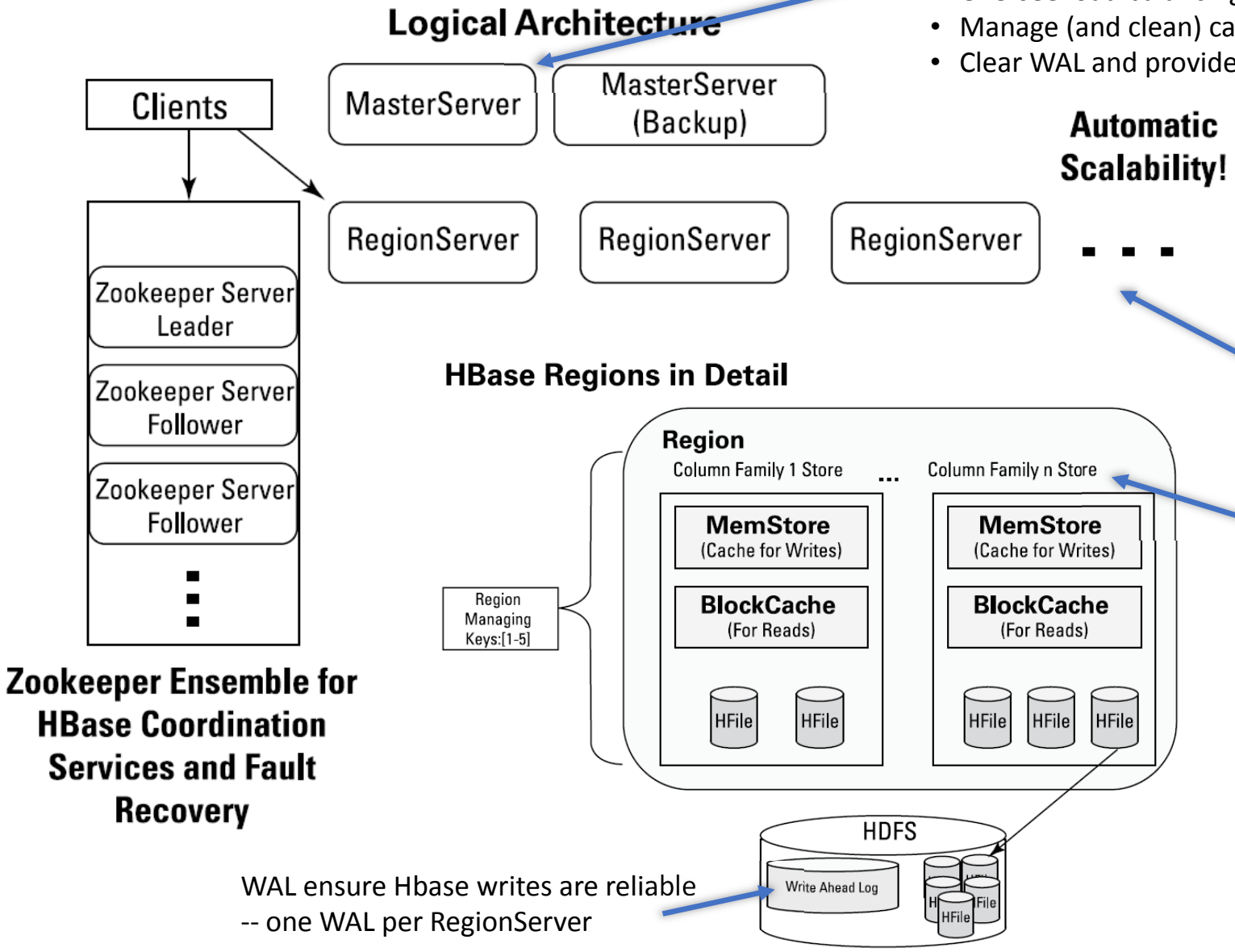
HBase Architecture

- **Table is made of regions**
- **Region – a range of rows stored together**
  - Single shard, used for scaling
  - Dynamically split as they become too big and merged if too small
- **Region Server – serves one or more regions**
  - A region is served by only 1 Region Server
- **Master Server – daemon responsible for managing HBase cluster, aka Region Servers**
- **HBase stores its data into HDFS**
  - Relies on HDFS's high availability and fault-tolerance features
- **HBase utilizes Zookeeper for distributed coordination**



HBase Architecture

- Monitor the RegionServers in the HBase cluster
- Handle metadata operations
- Assign regions and manage RegionServer failover
- Oversee load balancing of regions across all available RegionServers
- Manage (and clean) catalog tables
- Clear WAL and provide a coprocessor framework for observing master operations



Tables split into regions and served by region servers. RegionServers are the software processes (often called daemons) you activate to store and retrieve data in Hbase. Auto-sharding ensures scalability

Regions vertically divided by column families into "stores" in the HDFS using HFile objects. Stores are saved as files on HDFS

WAL ensure Hbase writes are reliable
-- one WAL per RegionServer

HBase example -- I

- Creating a table

```
hbase(main):002:0> create 'CustomerContactInfo', 'CustomerName', 'ContactInfo'
0 row(s) in 1.2080 seconds
```

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

HBase example -- II

- Entering records

```
hbase(main):008:0> put 'CustomerContactInfo', '00001', 'CustomerName:FN', 'John'
0 row(s) in 0.2870 seconds

hbase(main):009:0> put 'CustomerContactInfo', '00001', 'CustomerName:LN', 'Smith'
0 row(s) in 0.0170 seconds

hbase(main):010:0> put 'CustomerContactInfo', '00001', 'CustomerName:MN', 'T'
0 row(s) in 0.0070 seconds

hbase(main):011:0> put 'CustomerContactInfo', '00001', 'CustomerName:MN', 'Timothy'
0 row(s) in 0.0050 seconds
hbase(main):012:0> put 'CustomerContactInfo', '00001', 'ContactInfo:EA', 'John.
              Smith@xyz.com'
0 row(s) in 0.0170 seconds
hbase(main):013:0> put 'CustomerContactInfo', '00001', 'ContactInfo:SA', '1
              Hadoop Lane, NY 11111'
0 row(s) in 0.0030 seconds

hbase(main):014:0> put 'CustomerContactInfo', '00002', 'CustomerName:FN', 'Jane'
0 row(s) in 0.0290 seconds

hbase(main):015:0> put 'CustomerContactInfo', '00002', 'CustomerName:LN', 'Doe'
0 row(s) in 0.0090 seconds

hbase(main):016:0> put 'CustomerContactInfo', '00002', 'ContactInfo:SA', '7
              HBase Ave, CA 22222'
0 row(s) in 0.0240 seconds
```

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

HBase example -- III

- Scan results

```
hbase(main):020:0> scan 'CustomerContactInfo', {VERSIONS => 2}
ROW                                    COLUMN+CELL
 00001                                 column=ContactInfo:EA,
              timestamp=1383859183030, value=John.Smith@xyz.com
 00001                                 column=ContactInfo:SA,
              timestamp=1383859183073, value=1 Hadoop Lane, NY 11111
 00001                                 column=CustomerName:FN,
              timestamp=1383859182496, value=John
 00001                                 column=CustomerName:LN,
              timestamp=1383859182858, value=Smith
 00001                                 column=CustomerName:MN,
              timestamp=1383859183001, value=Timothy
 00001                                 column=CustomerName:MN,
              timestamp=1383859182915, value=T
 00002                                 column=ContactInfo:SA,
              timestamp=1383859185577, value=7 HBase Ave, CA 22222
 00002                                 column=CustomerName:FN,
              timestamp=1383859183103, value=Jane
 00002                                 column=CustomerName:LN,
              timestamp=1383859183163, value=Doe
2 row(s) in 0.0520 seconds
```

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

HBase example -- IV

- Using the ***get*** command to retrieve entire rows and individual values

```
(1) hbase(main):037:0> get 'CustomerContactInfo', '00001'
COLUMN                                  CELL
 ContactInfo:EA                         timestamp=1383859183030, value=John.
                Smith@xyz.com
 ContactInfo:SA                         timestamp=1383859183073, value=1 Hadoop
                Lane, NY 11111
 CustomerName:FN                        timestamp=1383859182496, value=John
 CustomerName:LN                        timestamp=1383859182858, value=Smith
 CustomerName:MN                        timestamp=1383859183001, value=Timothy
5 row(s) in 0.0150 seconds

(2) hbase(main):038:0> get 'CustomerContactInfo', '00001',
                       {COLUMN => 'CustomerName:MN'}
COLUMN                                  CELL
 CustomerName:MN                        timestamp=1383859183001, value=Timothy
1 row(s) in 0.0090 seconds

(3) hbase(main):039:0> get 'CustomerContactInfo', '00001',
                       {COLUMN => 'CustomerName:MN',
                       TIMESTAMP => 1383859182915}
COLUMN                                  CELL
 CustomerName:MN                        timestamp=1383859182915, value=T
1 row(s) in 0.0290 seconds
```

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

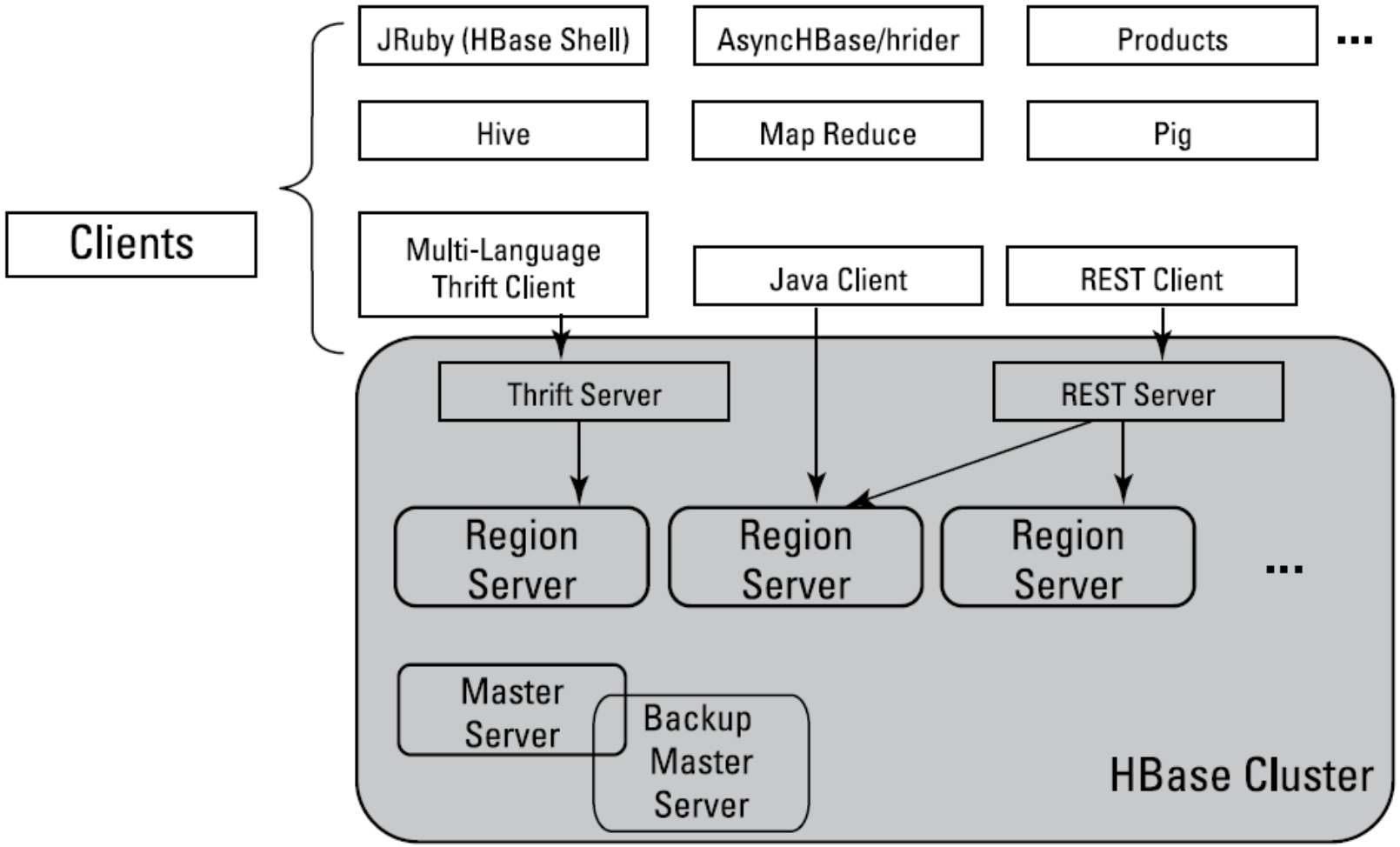
HBase vs. RDBMS

HBase	RDBMS
Column-oriented	Row-oriented (mostly)
Flexible schema, add columns on the fly	Fixed schema
Good with sparse tables	Not optimized for sparse tables
No (declarative) query language*	SQL
Wide tables	Narrow tables
Joins using MR – not optimized	Optimized for joins (small, fast ones too!)
Tight integration with MR	Not really
De-normalize data	Normalize as one can
Horizontal scalability – just add hardware	Hard to shard and scale
Consistent	Consistent
No transactions	Transactional
Good for semi-structured data as well as structured data	Good for structured data

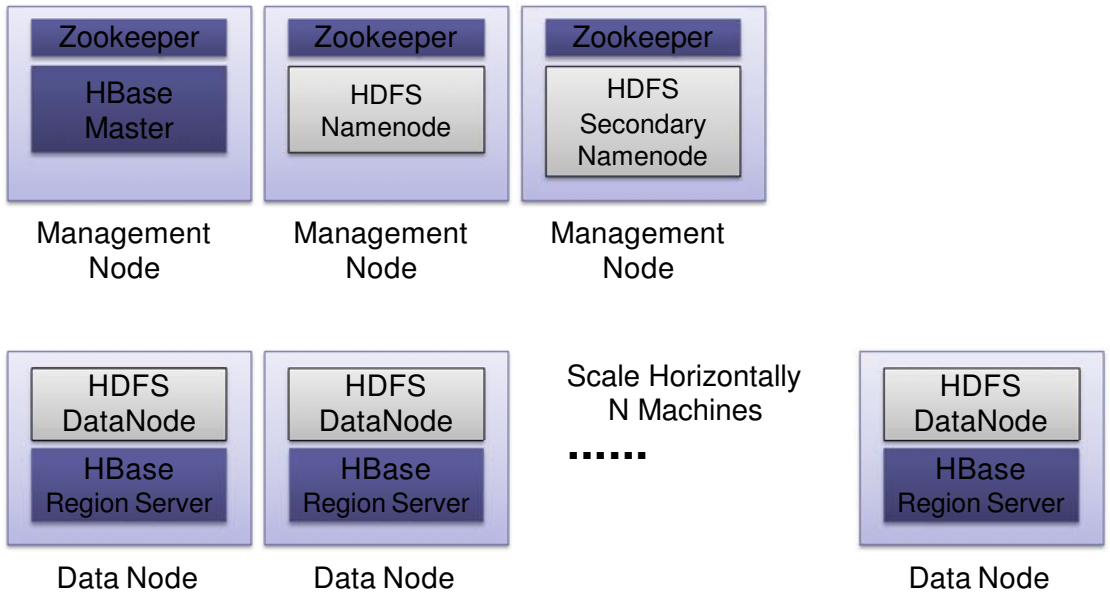
*HBase supports queries built based on proprietary APIs, REST queries, and languages (Java, Python, C/C++, PHP. Ruby). Apache Hive aids portability of SQL-based applications to HBase-based applications.

Getting things done with HBase

HBase Client Ecosystem



HBase Deployment

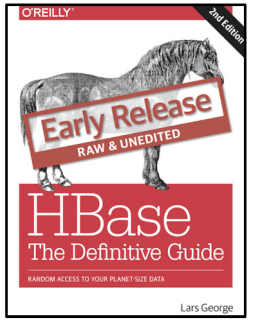


Resources

- **Home Page**
  - http://hbase.apache.org
- **Mailing Lists**
  - http://hbase.apache.org/mail-lists.html
  - Subscribe to User List
- **Wiki**
  - http://wiki.apache.org/hadoop/Hbase
- **Videos and Presentations**
  - http://hbase.apache.org/book.html#other.info

Resources: Books

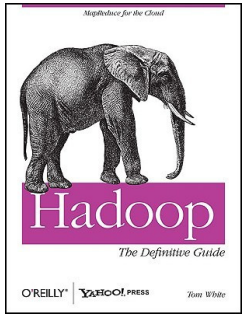
- **HBase: The Definitive Guide by Lars George**
  - Publication Date: September 20, 2011



- **Apache HBase Reference Guide**
  - Comes packaged with HBase
  - http://hbase.apache.org/book/book.html

APACHE HBASE

- **Hadoop: The Definitive Guide by Tom White**
  - Publication Date: May 22, 2012
  - Chapter about Hbase



Apache Hive

What is Hive?

- A system for managing and querying data stored in various databases and file systems that integrate with Hadoop
  - Uses MapReduce for execution
  - HDFS for storage
  - Extensible to other Data Repositories
- Key Building Principles:
  - SQL abstraction to integrate SQL-like queries (HiveQL) into the underlying Java without the need to implement queries in the low level
  - Extensibility
    - Pluggable map/reduce scripts in the language of your choice
    - Rich and User Defined data types and User Defined Functions
  - Interoperability (Extensible framework to support different file and data formats)
- Hive aids portability of SQL-based applications to Hadoop
- Amazon maintains a software fork of Apache Hive (https://bit.ly/2QShkWu)

Apache Hive

Hive applications

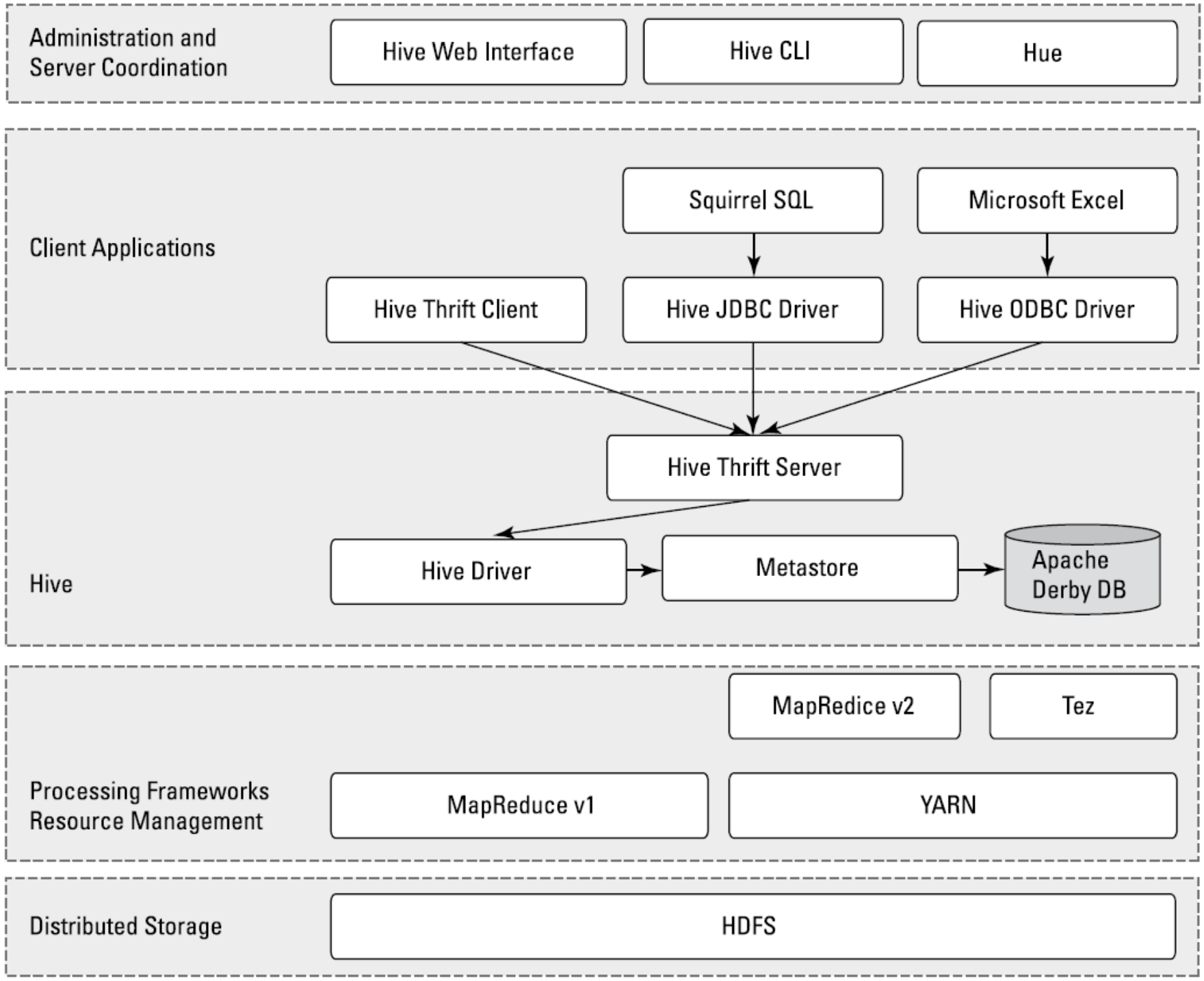
- Log processing
- Text mining
- Document indexing
- Customer-facing business intelligence (e.g., Google Analytics)
- Predictive modeling, hypothesis testing

Hive components

- CLI/UI/Shell: allows interactive queries like MySQL shell connected to database -- also supports web and JDBC clients
- Driver: session handles, fetch, execute
- Compiler: parse, plan, optimize
- Executor: DAG of stages (M/R, HDFS, or metadata)
- Metastore: schema, location in HDFS, SerDe

Metastore

- Database: namespace containing a set of tables
- Holds table definitions (column types, physical layout)
- Partition data
- Uses JPOX ORM for implementation; can be stored in Derby, MySQL, many other relational databases



Using the Hive CLI to Create a Table

```
(A) $ $HIVE_HOME/bin hive --service cli
(B) hive> set hive.cli.print.current.db=true;
(C) hive (default)> CREATE DATABASE ourfirstdatabase;
OK
Time taken: 3.756 seconds
(D) hive (default)> USE ourfirstdatabase;
OK
Time taken: 0.039 seconds
(E) hive (ourfirstdatabase)> CREATE TABLE our_first_table
           (
                        > FirstName        STRING,
                        > LastName         STRING,
                        > EmployeeId       INT);
OK
Time taken: 0.043 seconds
hive (ourfirstdatabase)> quit;
(F) $ ls /home/biadmin/Hive/warehouse/ourfirstdatabase.db
our_first_table
```

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

Another Hive Example

```
(A) CREATE TABLE IF NOT EXISTS myFlightInfo (
  Year SMALLINT, DontQueryMonth TINYINT, DayofMonth
          TINYINT, DayOfWeek TINYINT,
  DepTime SMALLINT, ArrTime SMALLINT,
  UniqueCarrier STRING, FlightNum STRING,
  AirTime SMALLINT, ArrDelay SMALLINT, DepDelay SMALLINT,
  Origin STRING, Dest STRING, Cancelled SMALLINT,
  CancellationCode STRING)
COMMENT 'Flight InfoTable'
PARTITIONED BY(Month TINYINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
(B) STORED AS RCFILE
TBLPROPERTIES ('creator'='Bruce Brown', 'created_at'='Mon
          Sep  2 14:24:19 EDT 2013');
```

D. deRoos *et al*, Hadoop for Dummies, John Wiley & Sons, 2014

HiveQL

- Internally, a compiler translates HiveQL statements into a directed acyclic graph of MapReduce, Tez, or Spark jobs, which are submitted to Hadoop for execution.
- Word Count program counts the number of times each word occurs in the input. The word count can be written in HiveQL as

```
CREATE TABLE docs (line STRING);

LOAD DATA INPATH 'docs' OVERWRITE INTO TABLE docs;

CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
  (SELECT explode(split(line, '\s')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
```

Create a new table called docs with a single column of type STRING called line

Load the specified file or directory ("docs") into the table. OVERWRITE specifies that the target table is to be re-written; otherwise the data would be appended.

Create a table called word_counts with two columns: word and count, which draws its input from the inner query

Split the input words into different rows of a temporary table aliased as w

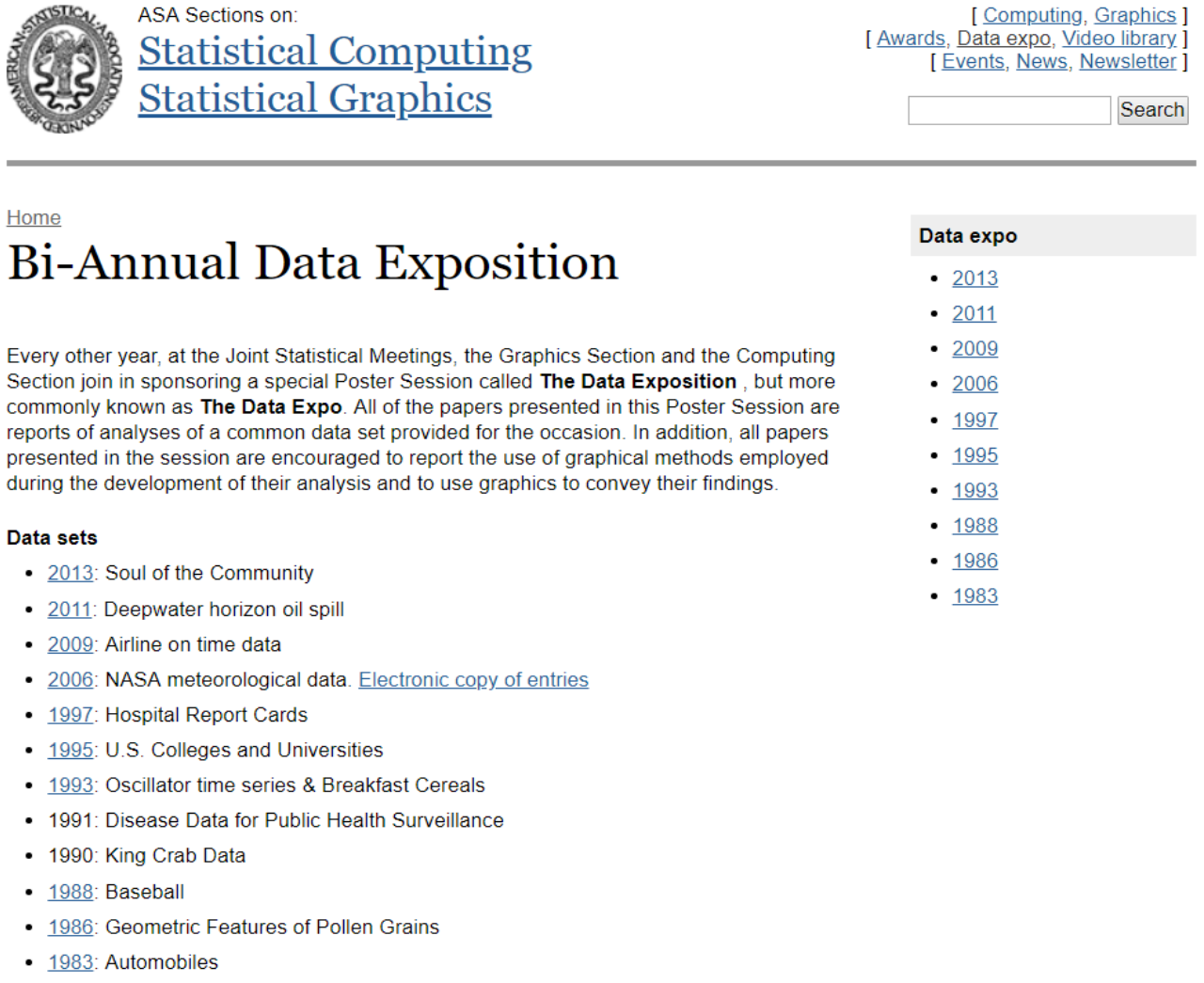
Groups the results based on their keys. This results in the count column holding the number of occurrences for each word of the word column

Sorts the words alphabetically

Wampler *et al.* Programming Hive Data Warehouse and Query Language for Hadoop, O'Reilly, 2012

Exercises

1. Download Airline Data and one of your own selected datasets from http://stat-computing.org/
2. Learn to use PIG. You can try the example in the reference
3. Use Oozie to schedule a few jobs
4. Try HBase. Use your own example
5. Try Hive. Use your own example



ASA Sections on:
Statistical Computing
Statistical Graphics

[Computing, Graphics]
[Awards, Data expo, Video library]
[Events, News, Newsletter]

Search

Home

Bi-Annual Data Exposition

Every other year, at the Joint Statistical Meetings, the Graphics Section and the Computing Section join in sponsoring a special Poster Session called **The Data Exposition** , but more commonly known as **The Data Expo**. All of the papers presented in this Poster Session are reports of analyses of a common data set provided for the occasion. In addition, all papers presented in the session are encouraged to report the use of graphical methods employed during the development of their analysis and to use graphics to convey their findings.

Data sets

- 2013: Soul of the Community
- 2011: Deepwater horizon oil spill
- 2009: Airline on time data
- 2006: NASA meteorological data. Electronic copy of entries
- 1997: Hospital Report Cards
- 1995: U.S. Colleges and Universities
- 1993: Oscillator time series & Breakfast Cereals
- 1991: Disease Data for Public Health Surveillance
- 1990: King Crab Data
- 1988: Baseball
- 1986: Geometric Features of Pollen Grains
- 1983: Automobiles

Data expo

- 2013
- 2011
- 2009
- 2006
- 1997
- 1995
- 1993
- 1988
- 1986
- 1983

NoSQL Database

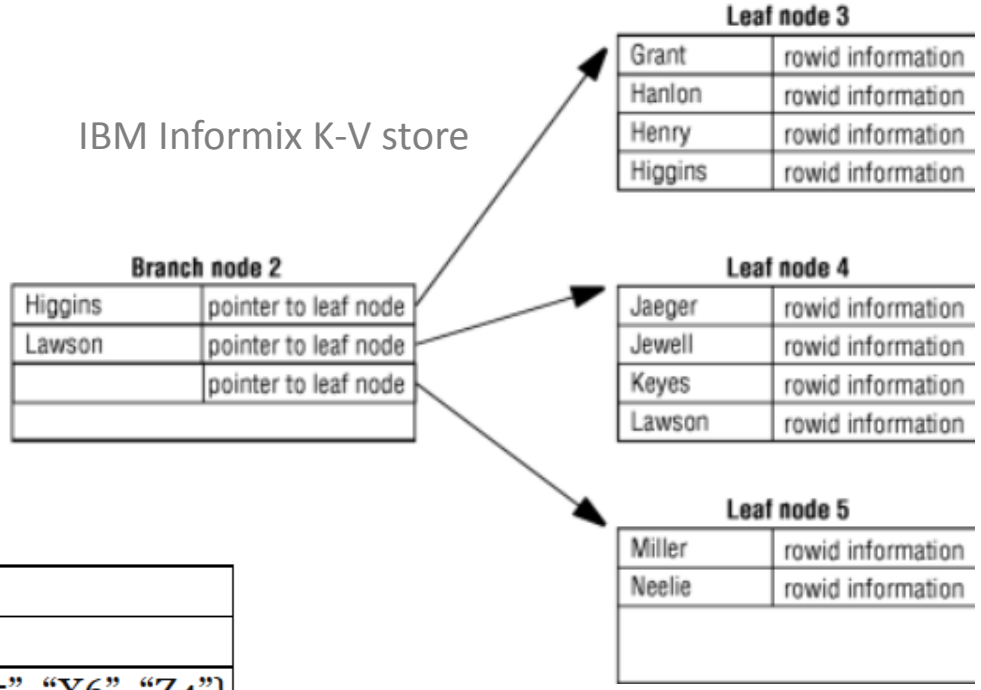
- Key-Value Store
- Document Store
- Tabular Store
- Object Database
- Graph Database
  - Property graphs
  - Resource Description Framework (RDF) graphs

NoSQL Database

- Key-Value Store
- Document Store
- Tabular Store
- Object Database
- Graph Database
  - Property graphs
  - Resource Description Framework (RDF) graphs

Key-Value Store

IBM Informix K-V store



Example Data Represented in a Key–Value Store

Key	Value
…	
"BMW"	{"1-Series", "3-Series", "5-Series", "5-Series GT", "7-Series", "X3", "X5", "X6", "Z4"}
"Buick"	{"Enclave", "LaCrosse", "Lucerne", "Regal"}
"Cadillac"	{"CTS", "DTS", "Escalade", "Escalade ESV", "Escalade EXT", "SRX", "STS"}
…	

- Get(*key*), which returns the value associated with the provided *key*.
- Put(*key*, *value*), which associates the *value* with the *key*.
- Multi-get(*key1*, *key2*,.., *keyN*), which returns the list of values associated with the list of *keys*.
- Delete(*key*), which removes the entry for the *key* from the data store.

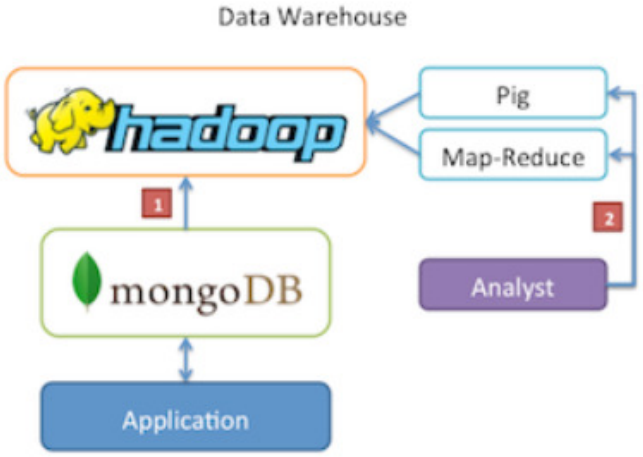
Document Store

Relational data model

Highly-structured table organization with rigidly-defined data formats and record structure.

Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.



The following diagram highlights the components of a MongoDB insert operation:

```
db.users.insert (          <-- collection
   {
     name: "sue",          <-- field: value
     age: 26,              <-- field: value
     status: "A"           <-- field: value
   }                                          } document
)
```

The components of a MongoDB insert operations.

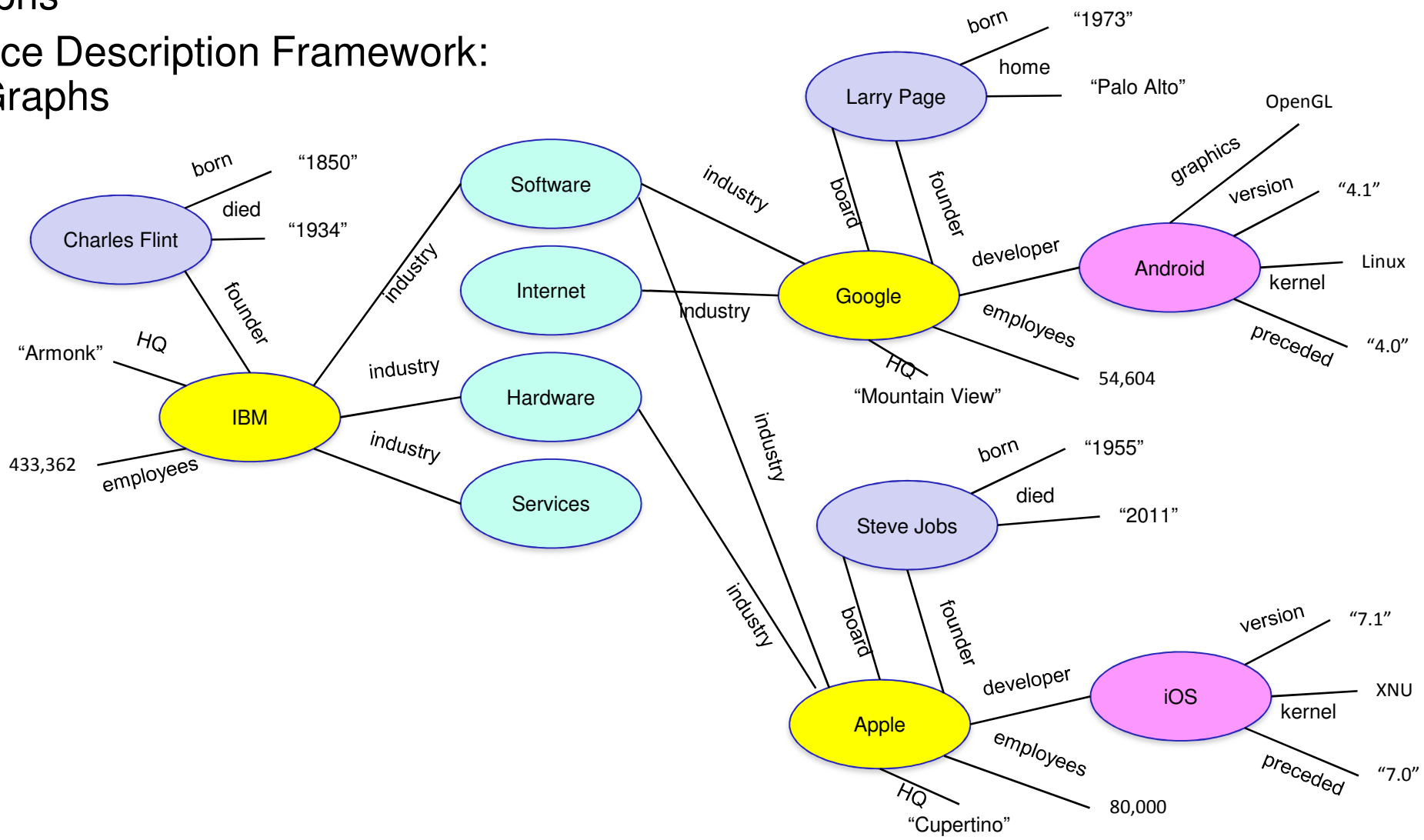
The following diagram shows the same query in SQL:

```
INSERT INTO users                    <-- table
            ( name, age, status )    <-- columns
VALUES      ( "sue", 26, "A" )       <-- values/row
```

The components of a SQL INSERT statement.

Graph Data

- Property Graphs
- RDF (Resource Description Framework: Triplestore) Graphs



What is the fundamental challenge for RDB on Linked Data?

In Relational DB, relationships are *distributed.* It takes a long time to *JOIN* to retrieve a graph from data

User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

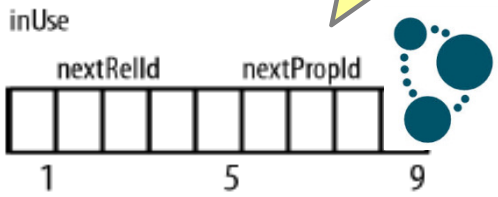
LineItem

OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

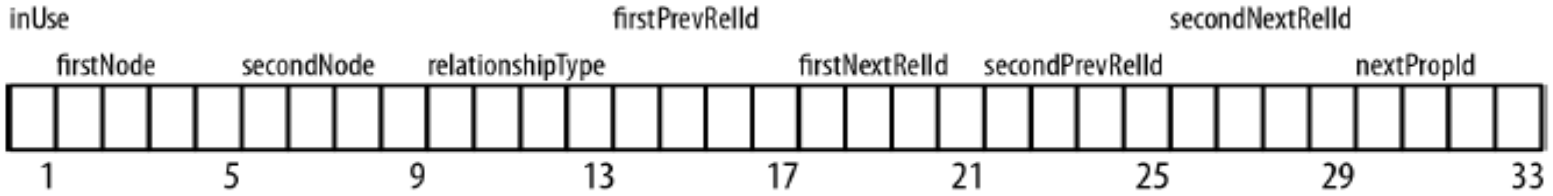
Product

ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

Native Graph DB stores nodes and relationships directly. It makes retrieval efficient.

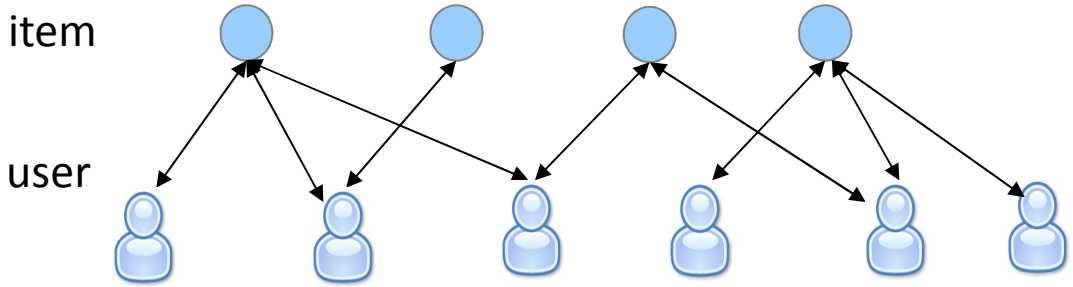


Relationship (33 bytes)



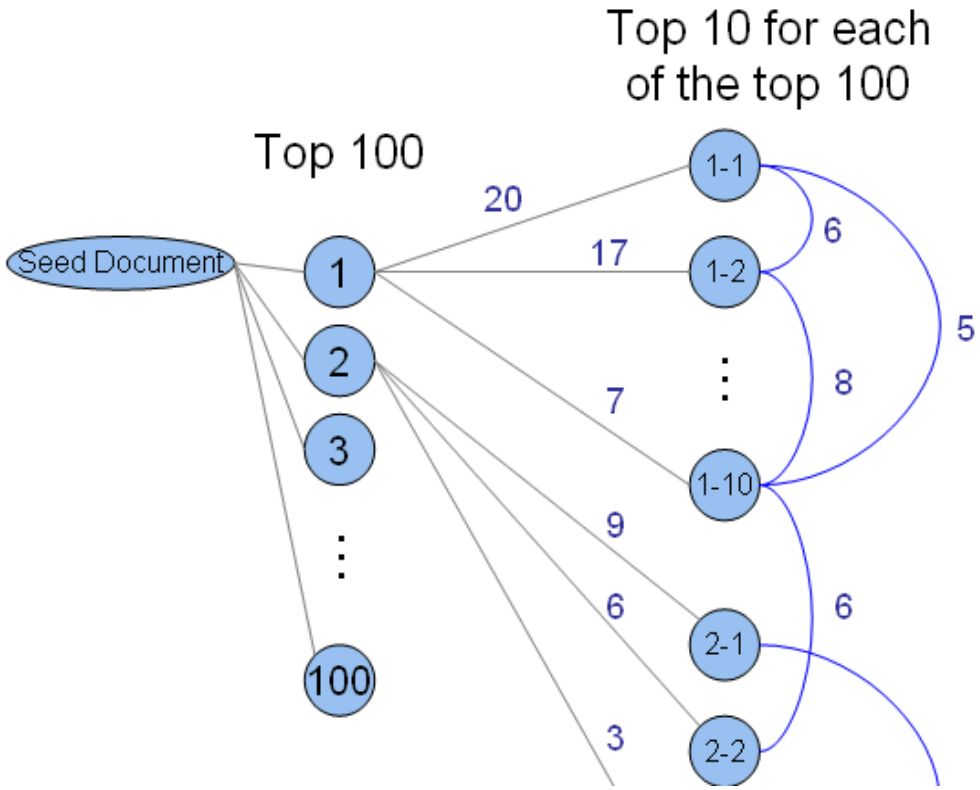
Retrieving multi-step relationships is a "**graph traversal**" problem

Preliminary datastore comparison for Recommendation & Visualization



People who bought this also bought that..

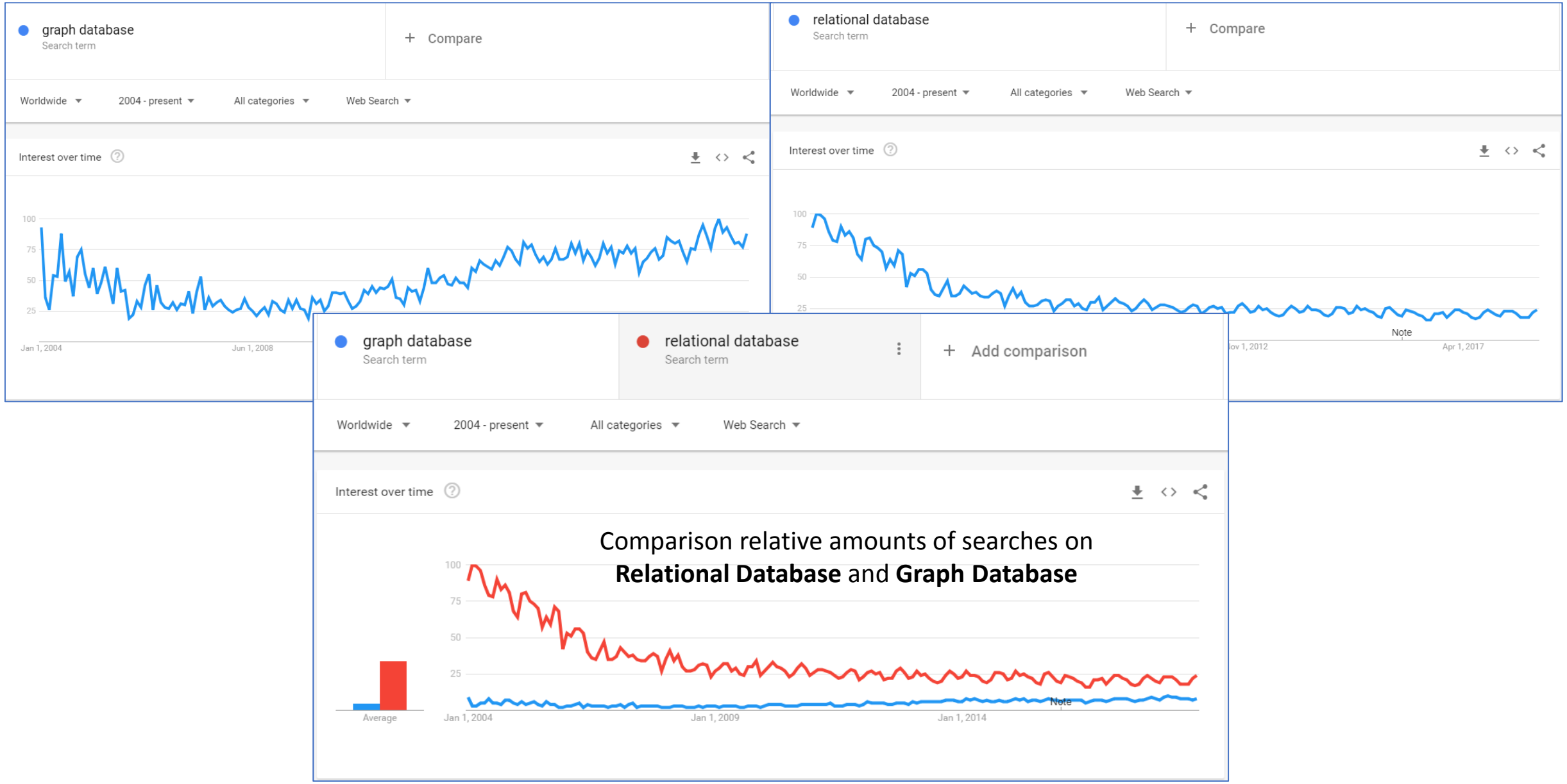
Recommendation ==> 2-hop traversal & ranking



Visualization ==> 4-hop traversal & rankings

Google Trends on Relational vs Graph Databases (10/16/2018)

Trends of search interest on **Graph Database** and **Relational Database**



How to Visualize Huge Static Graph?

76425 species

Tree of Life by Dr. Yifan Hu

14.8 million tweets

The information diffusion graph of the death of Osama bin Laden by Gilad Lotan

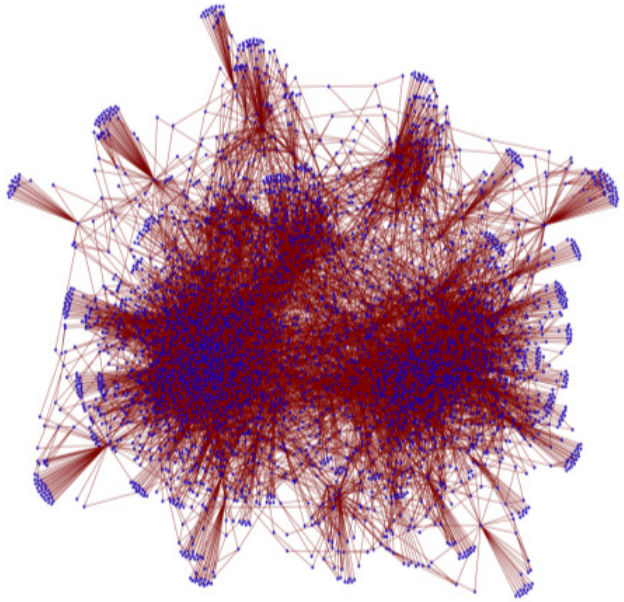
500 million users

Facebook friendship graph by Paul Butler

Challenging Task :

Squeezing millions and even billions of records into million pixels (1600 X 1200 ≈ 2 million pixels)

Visualization Key Challenges



Visual clutter

How can we encode the information intuitively?

Performance issues

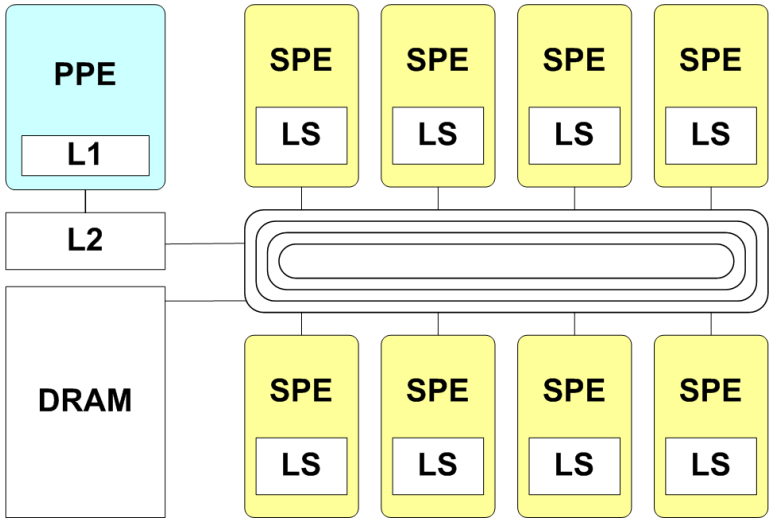
How can we render the huge datasets in real time with rich interactions?

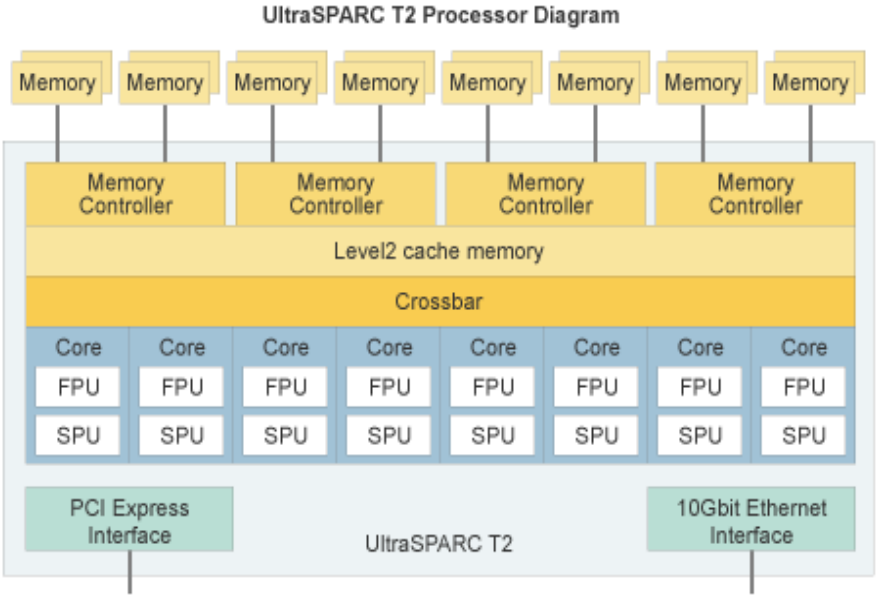
Cognition

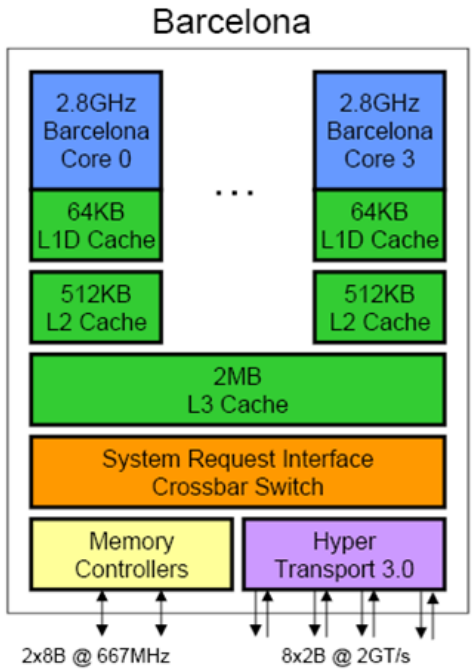
How can users understand the visual representation when the information is overwhelming?

Platform Dependent Graphical Models

- Homogeneous multicore processors
  - Intel Xeon E5335 (Clovertown)
  - AMD Opteron 2347 (Barcelona)
  - *Netezza (FPGA, multicore)*
- Homogeneous manycore processors
  - Sun UltraSPARC T2 (Niagara 2), GPGPU
- Heterogeneous multicore processors
  - Cell Broadband Engine
- Clusters
  - HPCC, DataStar, BlueGene, etc.

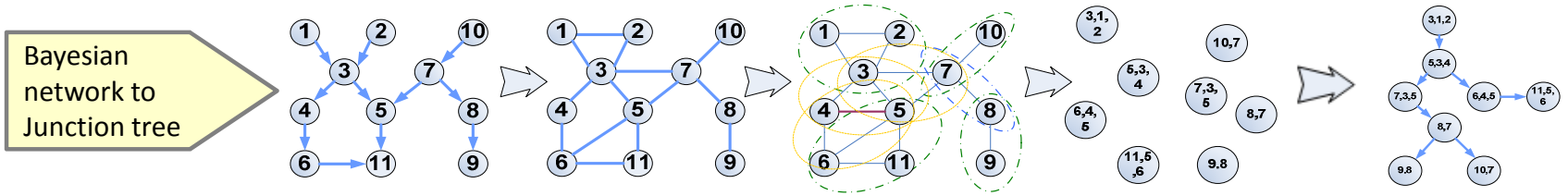






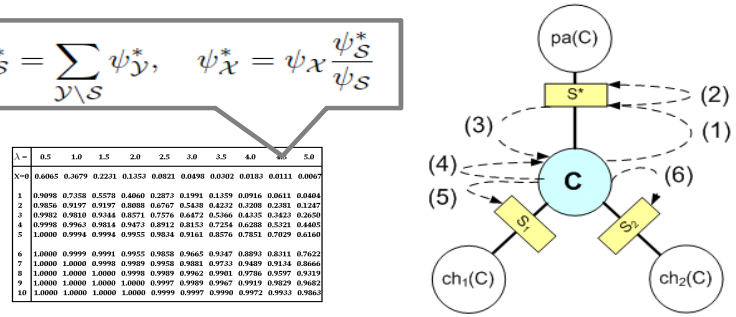
Graph Workload Types

- Type 1: Computations on graph structures / topologies
  - e.g., converting Bayesian network into junction tree, graph traversal (BFS/DFS), etc.
  - Characteristics: poor locality, irregular memory access, limited numeric operations



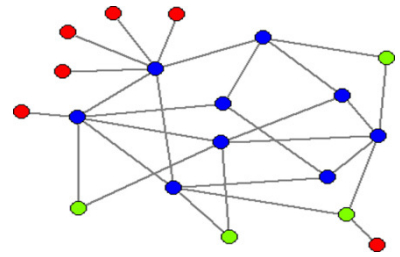
- Type 2: Computations on graph with rich properties
  - e.g., Belief propagation: diffuse information through a graph using statistical models
  - Characteristics
    - Locality and memory access pattern depend on vertex models
    - Typically a lot of numeric operations
    - Hybrid workload

$$\psi^*_{\mathcal{S}} = \sum_{\mathcal{Y}\setminus\mathcal{S}} \psi^*_{\mathcal{Y}}, \quad \psi^*_{\mathcal{X}} = \psi_{\mathcal{X}} \frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}}$$



- Type 3: Computations on dynamic graphs
  - e.g., streaming graph clustering, incremental k-core, etc.
  - Characteristics
    - Poor locality, irregular memory access
    - Operations to update a model (e.g., cluster, sub-graph)
    - Hybrid workload

3-core subgraph



Large-scale graph benchmark – Graph 500

Top Ten from June 2018 BFS

RANK	MACHINE	VENDOR	INSTALLATION SITE	LOCATION	COUNTRY	YEAR	NUMBER OF NODES	NUMBER OF CORES	SCALE	GTEPS
1	K computer	Fujitsu	RIKEN Advanced Institute for Computational Science (AICS)	Kobe Hyogo	Japan	2011	82944	663552	40	38621.4
2	Sunway TaihuLight	NRCPC	National Supercomputing Center in Wuxi	Wuxi	China	2015	40768	10599680	40	23755.7
3	DOE/NNSA/LLNL Sequoia	IBM	Lawrence Livermore National Laboratory	Livermore CA	USA	2012	98304	1572864	41	23751
4	DOE/SC/Argonne National Laboratory Mira	IBM	Argonne National Laboratory	Chicago IL	USA	2012	49152	786432	40	14982
5	JUQUEEN	IBM	Forschungszentrum Juelich (FZJ)	Juelich	Germany	2012	16384	262144	38	5848
6	ALCF Mira - 8192 partition	IBM	Argonne National Laboratory	Chicago IL	United States	2012	8192	131072	36	4212
7	ALCF Mira - 8192 partition	IBM	DOE/ALCF	Argonne National Laboratory	USA	2012	8192	131072	36	3556.7
8	Fermi	IBM	CINECA	Casalecchio Di Reno	Italy	2012	8192	131072	37	2567
9	ALCF Mira - 4096 partition	IBM	Argonne National Laboratory	Chicago IL	United States	2012	4096	65536	35	2348
10	Tianhe-2 (MilkyWay-2)	National University of Defense	Changsha China	Changsha China	China	2013	8192	196608	36	2061.48

https://graph500.org/

Thanks ! ☺

Questions ?