Module 5

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 5 Study Guide and Deliverables

Readings:	There are no readings from our textbooks for this module.
Assignments:	Assignment 5 due Wednesday, August 12 at 5:00 PM ET You may choose to submit Assignment 5.0 or Assignment 5.1 for your grade. You may also submit the other assignment for extra credit.
Assessments:	Quiz 5 due Wednesday, August 12 at 5:00 PM ET
Term Project Note:	During this module you should be finishing the technical implementation of your term project and completing the term paper and presentation. You are encouraged to present your partially completed project products to your facilitator for feedback before the final delivery.
Live Classrooms:	Supplementary Live Session, Wednesday, August 5, 8:00 PM- 10:00 PM ET Current week's assignment review and examples, Thursday, August 6, 8:00 PM - 9:00 PM ET Live office help, Saturday, August 8, 11:00 AM - 12:00 PM ET Live office help, Monday, August 10, 8:00 PM - 8:45 PM ET

Learning Objectives

By reading this lesson, participating in discussions, and completing the assignments, you will be able to:

- Understand the different levels of consistency
- Understand the concepts of consistency, availability and partition tolerance in distributed databases
- Understand when it could be practical to move to NoSQL solutions
- Describe the design models of NoSQL databases
- Describe the main features of some NoSQL database products like MongoDB, Neo4j, BigTable, and DynamoDB

Big Data Databases

"What is Big Data?"

> The term “big data” has come to refer to the many kinds of unstructured and variably structured data that are difficult to impossible to store and process in traditional SQL relational, object-oriented, or object-relational databases. Sometimes this data is truly big, such as all the world’s tweets that are analyzed for and by companies such as Apple, Sometimes big data is no bigger than IT professionals have been analyzing for years; a common example is the comment and other text fields in data warehouses. The common ground for all data termed “big data” is that it isn’t amenable to traditional structured analysis such as in a dimensional data warehouse. Within the context of this course the term “big data databases” refers to databases that can efficiently store and process hundreds of petabytes of data.

This lecture introduces the broad topic of parallel scalable non-relational databases that can store and process internet-scale and unstructured and multi-structured data. These massively parallel databases are based on key-value, key-document-value and other more scalable data models, rather than the relational model. Because all contemporary relational database management systems also support SQL, these non-relational databases have also been termed “NoSQL” databases. This can be confusing, because many of the non-relational database systems support variants of SQL, so “noSQL” has come to be interpreted as “Not only SQL.” Within the last few years numerous scalable databases have been developed for processing the torrents of “big data” available from web sites and other systems. Accordingly we refer to these massively scalable databases as “big data” databases. Scalable big data database systems share common characteristics:

- They are based on key-value or other scalable data models rather than the relational model.
- To protect against data loss, data is stored redundantly, often triplicated, without transaction logs.
- Because it is normal to have parts of these very large systems fail, these systems are designed with multiple fault tolerance and continuous non-interruptive recovery.
- Because big data is too big to move, processing is based on sending requests to the distributed data, processing it where it is stored, and aggregating the results.
- Big data databases support more scalable transaction models such as “eventual consistency” rather than ACIDS transactions.
- Most big data databases support the storage and manipulation of semi-structured and unstructured data, rather than only structured data that fits a pre-specified schema.
- Big data databases that obtain high scalability do so through distributed fault-tolerant architectures that include transparent fault recovery. These architectures can scale well horizontally to hundreds of petabytes and hundreds of billions of operations per day.
- Big data databases are used in applications where high performance with large amounts of data is more important than consistency. Examples include search engines and serving web pages, documents, and streaming media on high-traffic web sites. Many big data databases were developed to support web sites such as Google, Amazon, Facebook, Twitter, and YouTube, which have size and throughput scalability requirements that exceed the capability of ordinary SQL relational databases.
- Big data databases generally require more read transactions than interactive updates or deletes. This is one reason why it is easier to maximize performance using non-relational models.

Big data databases such as MongoDB implement the schema when the data is read, rather than before the data is written, as in an RDBMS. This makes loading fast and flexible, and increases the processing required during data read. This does not support integrity enforcement as data is loaded, because there is no preexisting schema.

> ***Transparent recovery*** means that the database automatically detects faults (such as failed storage, communications or processing) and restores fault-tolerance without interrupting ongoing operations. This usually involves creating replacement redundant copies of data on functional resources. Because failures in large systems are a normal part of operations it is common to store data three times, so that if one copy fails there is still redundancy during recovery.

Scalability Limitations of SQL Relational Databases

Before discussing big data databases it is helpful to review the limitations of relational databases, to understand the applications for which relational databases are necessary or preferable and applications for which relational databases are infeasible or unnecessarily expensive. Recall that the relational model is based on data stored in tables, where the rows in the tables all have the same predefined structure, and

that the only way that data can be related between tables is by repeating the data values in multiple tables. The relational model has been extended in the ANSI/ISO standards by adding object extensions to the relational model, and this improves scalability, but the limitations of the relational model remain. The main limitations are:

Limitations in locating rows in large tables. In the relational model rows are identified and located using data values in key columns, and indexes are used to improve scalability. When physical tables reach multi-terabyte size operations such as storage allocation, backup, and recovery become difficult. Consequently relational databases have long supported row partitioning of large tables to permit large logical tables to be partitioned into manageable sized physical tables. Together with tablespaces to represent composite physical storage, this allows relational databases to support many-terabyte tables on appropriate platforms. Efficiently locating rows in a row partitioned table requires the functionality of a global index that is shared by all of the processes that need to find rows in the partitioned table. The size of these global indexes scales as the size of the table, and global indexes are expensive to maintain in distributed databases, so creating, maintaining, and storing many copies of global indexes does not scale well. Key-value and other massively parallel databases circumvent this limitation by encoding the allocation in the key, which is machine generated in the same way that an OID is generated in object-oriented databases.

> Advanced topic
>
> Using Partition Keys as Global Index Columns: It is often feasible to row partition a large table in such a way that the partitioning key is part of a composite key for common global retrievals, thereby eliminating the need for a global index. For example, suppose a large logical Sales_Order_Item table is row partitioned by the sale_date column, so that there is one physical table for each date. Then the application code could examine any request against Sales_Order_Item that includes a restriction on sale_date, and direct that request to the appropriate physical table. The physical tables can have local indexes to support efficient identification of the rows that correspond to the other columns of the request key, such as sales_order_number or product_id. This eliminates the need to globally index Sales_Order_Item for requests that are restricted by sale_date.

Join limitations. The only way that relational databases relate data in different tables is by repeating values in two or more tables, and then joining those tables. Efficient execution of joins of large tables requires indexes on the joined columns of both tables. Unfortunately, there are no scalable implementations of global indexes. Key-value, object-oriented, and object-relational databases have more scalable means to relate records, using references that encode the location of the referenced records.

Transaction limitations. SQL relational databases aspire to the ACIDS (Atomicity, Consistency, Isolation, Durability, and Serializability) transaction model. Each of these aspects of a transaction has scalability limitations:

- ***Atomicity*** is the assurance that all parts of a transaction are executed, if the transaction is executed, or none, if the transaction is aborted. Recall that implementing atomicity in distributed transactions in ACIDS-compliant databases requires two-phase and three-phase commit protocols, and that those protocols can be slow. When a database includes multiple copies of each data item on different loosely coupled geographically distributed computers implementing atomicity is even more difficult and slower.
- ***Consistency*** is the assurance that all database constraints are satisfied when a transaction has committed. Big data databases have limited database constraints and at best eventual consistency models, where changes will eventually propagate to all redundant copies of data, assuming that there are no intervening changes.
- ***Isolation*** is the assurance that uncommitted changes are not visible to other sessions. Big data databases provide no or limited global isolation, because global isolation requires global locks, which do not scale well.
- ***Durability*** is the assurance that committed changes will survive failure of the platforms, including storage. Because of their fault tolerance and continuous non-interruptive recovery big data databases prevent data loss, but because of the lack of transactions it can be difficult to assure that changes are not overwritten.
- ***Serializability*** is the assurance that to each user of the database it appears that their requests are executed in the order they were presented, with no intervening operations. Big data databases may support the assurance that to each user the database appears to be processing requests in order that they are presented, but the lack of isolation and global consistency means that there is no assurance that requests of other users will not be interspersed. This makes big data databases unsuitable for most transaction processing.

Strong, Weak, and Eventual Consistency

Consistency is the property of a transaction that ensures that all database constraints are satisfied when a transaction has committed. Consistency is straightforward with single-instance databases but becomes more complex in distributed big data databases.

Consistency has several levels:

Strong: Updates occur as part of one atomic transaction and all copies of the data are updated simultaneously. After the update each access of the data will show a consistent view of the data. In Figure 1, the single-node transaction will result in strong consistency. Distributed databases implement strong consistency using distributed locks and two-phase commit.

Weak: When updates occur on at least one copy of the data the system does not guarantee that reads on the other copies will reflect the latest updates, although they may. It is also possible that updates will not be seen in the correct order. The “lazy replication” scheme described above results in weak consistency. Weak replication is needed in large distributed big data databases because not all copies of data may be accessible when one copy is updated.

Eventual: This is a type of weak consistency. With eventual consistency if one copy of the data is updated and there are no changes to the data all the copies of the data will eventually be updated, but in the meantime, only some reads of the data will reflect the updated version.

Read-your-writes: This is a variation of eventual consistency. *Read-your-writes consistency* ensures that users see the updates they made with every subsequent read, even though the updates might not have propagated to all copies. With *Read-your-writes* consistency, not all users have the same view of the data.

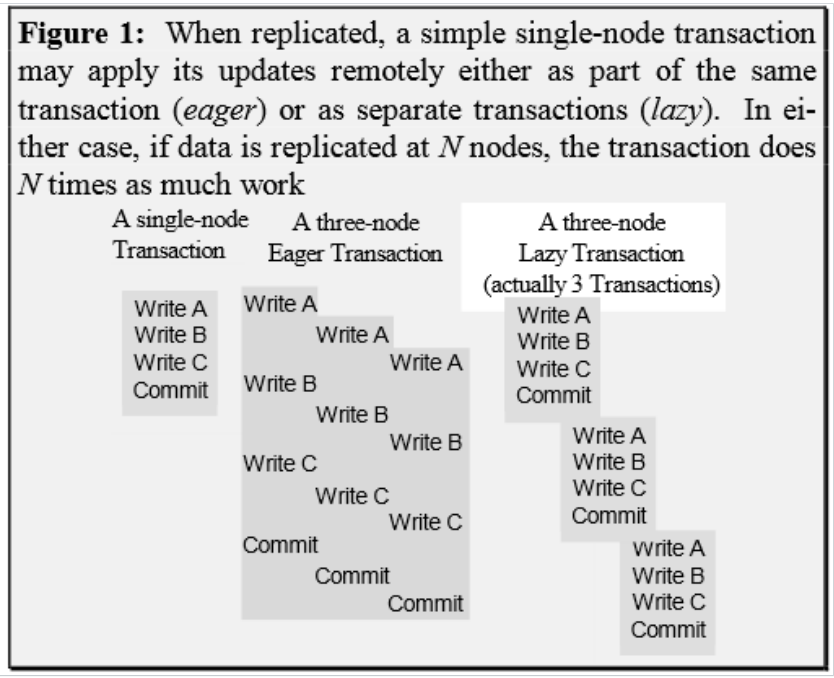
Eager and Lazy Replication

To ensure strict concurrency, every copy of the data needs to be updated across all nodes as a single transaction. With distributed data, this is referred to as *eager replication*. Eager replication ensures that all replicas are updated as one atomic transaction.(See “A three-node Eager Transaction” in Figure 1.)

Although strong consistency may be possible with eager replication, update performance and availability will suffer, since many copies of the data have to be updated before the next transaction is executed. If a network partition is unavailable, or unable to send messages to the other nodes that house the data, the update transaction could take prohibitive amounts of time to successfully complete. With more complicated write transactions, or a heavy user load, more conflicts and deadlocks can occur and performance can suffer even more, so eager replication is difficult to scale.

Many NoSQL implementations use *lazy replication*. which updates one copy immediately, and then asynchronously updates the remaining copies. (See “A three-node Lazy Transaction” in Figure 1.) Depending on which copy of the data is being read, the next query may see the updated version of the data or it may see an older copy of the data.

This diagram illustrates the steps involved in both eager and lazy replication techniques:



Gray, Helland, O'Neil, & Shasha

CAP Theorem

Eric Brewer, in a keynote speech at the Proceedings of the Annual ACM Symposium on Principles of Distributed Computing in 2000, introduced his CAP theorem, which proposed the idea that it was not possible to have all three properties of consistency, availability and partition tolerance simultaneously, and that at best only two of these were possible at any one time.

Partition tolerance refers to overcoming network performance issues. In other words, if partition tolerance has been achieved, and if one of the servers crashes, or if the remote machines are unable to communicate with each other or with the server, the system should still work correctly.

If consistency is strictly maintained, the distributed nodes will be unavailable for a certain amount of time so that the update can be applied to all copies of the database. So it is difficult if not impossible to both maintain strict consistency and high availability.

Many applications are willing to accept less stringent consistency rules in exchange for scalability and high availability. But it is important to be aware of the tradeoffs and the implications involved. At one extreme, it is possible that the application will always be able to read some version of the data, but may not see the latest updates. At the other extreme, the application will always see the current state of the data, but will, at times, be unable to process new transactions until all copies have been updated.

BASE (Basically Available, Soft-State, Eventually Consistent)

If a database system uses strict consistency, all users will have exactly the same view of the data all the time. This can be critical for certain applications. But to achieve strict consistency, all of the replicated nodes need to wait until all the changes are propagated to all replicas before executing any more transactions.

However, with many NoSQL models, strong consistency is not essential, but scalability and availability are. BASE is an alternative to the ACID properties of a relational database. BASE stands for "Basically Available", "Soft-State", and "Eventually Consistent".

> **Basically Available:** The system will be running as much as possible. This means that *when* components fail, only part of the system will fail, and the system itself will keep operating as if nothing happened. Failures have to be handled as they happen, and most importantly, must be transparent to the user processes.
>
> **Soft-State:** Information is in a *soft-state* when it is not yet written to disk. As an example, suppose a user places an item in his or her shopping cart. But the change to the shopping cart hasn't yet been made permanent, so the change is in a soft-state. A problem could happen if the item is no longer available when the user goes to "checkout". Many times, especially if this happens infrequently, web users are patient with this type of "glitch".

In contrast, *hard-state* means thatthe system has been updated, and the changes have been made permanent. To continue the previous example, every time a user views his or her shopping cart, it looks exactly the way he or she expects it to be.

Eventually consistent: This is a weak consistency model where all copies of the data are eventually updated.

MapReduce

MapReduce is a data model developed by Google for processing large volumes of data in parallel. It consists of a two-phase approach, a set of mapper tasks that reads through data sequentially and creates key/value pairs, and a set of reducer tasks, that reads the sets of key/value pairs and outputs a list of values for a given key. A MapReduce job is usually run on data that is not contained in a database.

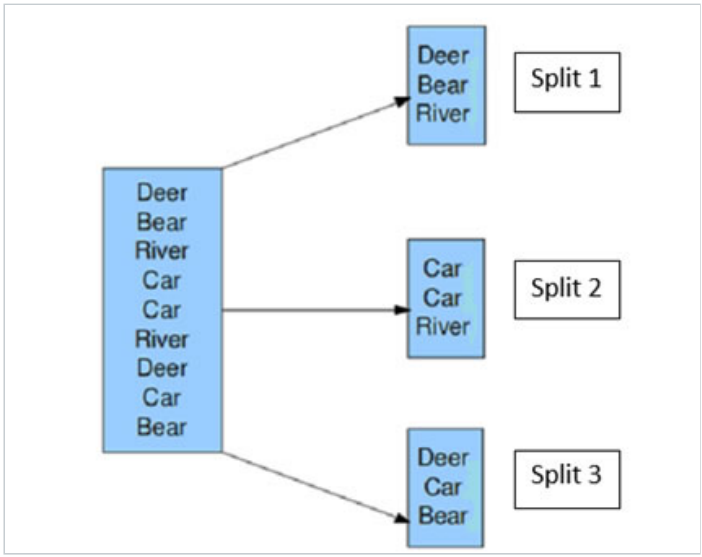
A framework like the Hadoop framework or Google File System parallelizes the mapper tasks and the reducer tasks, sorts the mapper outputs, and stores the data in a distributed file system. Each mapper runs on a portion of the data concurrently with other mappers running on other, different portions of the data. Similarly, the reducers run in parallel on different mapper outputs.

This parallelization makes MapReduce highly scalable. Terabytes of data can be processed in parallel across multiple machines, making it possible to analyze web logs, patterns in videos, or in Twitter feeds, things that were very difficult or that were not possible with traditional DBMS processing. Frameworks like Hadoop abstract the more difficult aspects of parallel programming, enabling the programmer to focus on algorithms like MapReduce that manage the data.

Example

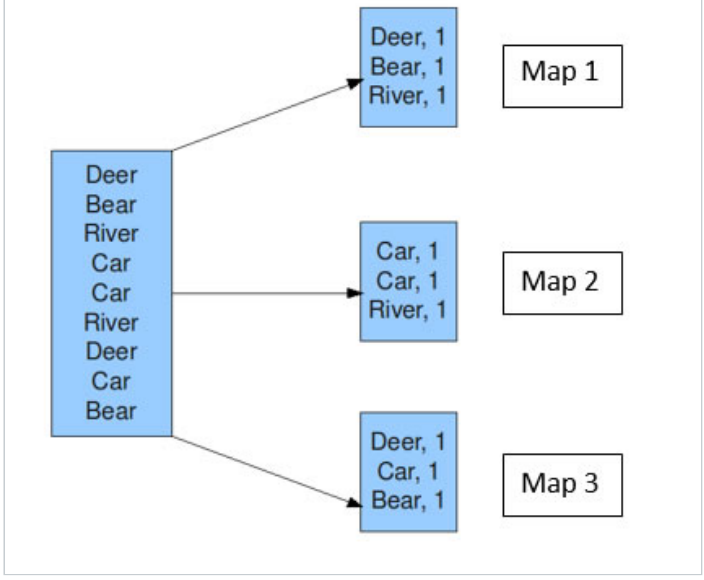
One easy way to see how MapReduce works is through a WordCount application. Suppose the task was to count the occurrences of different words in a word list. For example, we want to see how many times "deer" appears in the list.

The framework first splits the data into different pieces or "splits". In this example, Split 1 would contain the first three words, "Deer", "Bear" and "River". Split 2, would contain the next three, "Car", "Car" and "River", and Split 3 would contain "Deer", "Car" and "Bear".



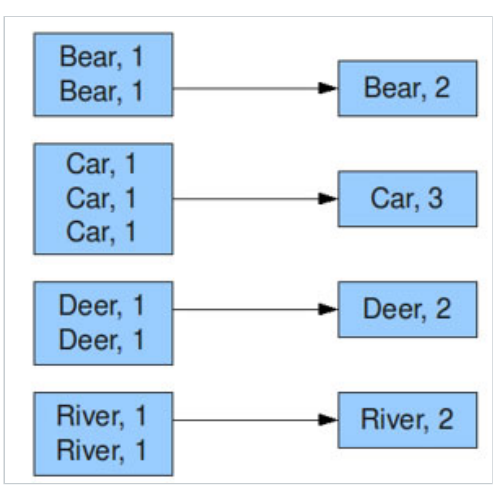
Source: Introduction to Hadoop

Mapper tasks run on each of the splits, outputting key/value pairs of the word and the count of each individual occurrence of the word, which is 1. The output of a mapper task is called a map.



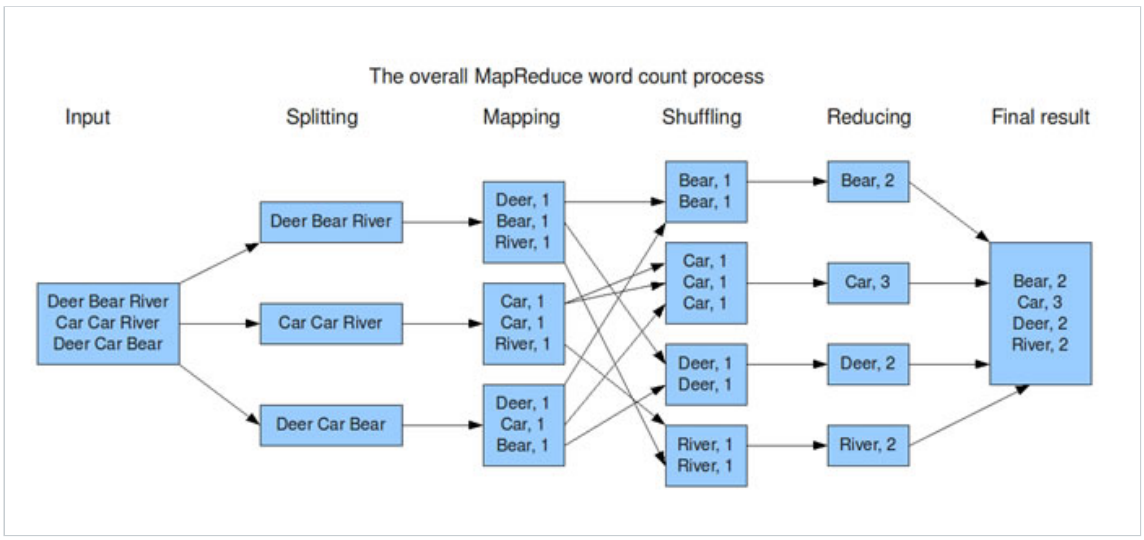
Source: Introduction to Hadoop

Then the framework groups together identical keys and sorts them in a step that is sometimes referred to as "shuffling". The input to the reducer tasks is sorted by key, which in this case is each word. The reducer for WordCount aggregates the keys, and the final output file is a file containing the key and the list of values, which in our example is just the sum of all the occurrences of the word. For our example, the reducer output would be:



Source: Introduction to Hadoop

This is an overview of the entire process.



Source: Introduction to Hadoop

Although this seems very straightforward, MapReduce can handle many complex data management tasks. Reducer outputs can be reused as mapper inputs to produce more significant results. The use of regular expression checking and the ability to specify which data elements or combinations can become key/value pairs gives MapReduce almost unlimited possibilities for implementing high-level algorithms, such as Google's PageRank algorithm and many others.

Inside MapReduce

There are different implementations for MapReduce, but one of the most popular is as part of the Hadoop framework (created from the Google File System design). The framework handles many issues behind the scenes to make MapReduce work on data sets as large as petabytes of data, or on data sets that use volumes of semi-structured or unstructured data, such as videos and web logs. Hadoop has, as its basis, a distributed file system that makes it ideal for running MapReduce jobs. Like other MapReduce implementations, Hadoop partitions and stores the data. It implements a publisher/subscriber structure, with a publisher system called the "NameNode" which handles the metadata, and individual machines called "DataNodes" that house and monitor the data.

The framework schedules the mapper and reducer tasks through another publisher/subscriber structure. With the earlier versions of Hadoop, the publisher is called the "JobTracker". The JobTracker schedules and monitors several mapper and reducer programs in parallel on the different machines that house the data for a particular job. These "subscriber" tasks are called "TaskTrackers".

Instead of the traditional method of paging in the data for the code to process, termed "data migration", with Hadoop/MapReduce the code is moved to the machines that house the pieces of data. This is called "computation migration". This highly scalable design makes it easy to run multiple jobs in parallel, which makes it a more practical way to process very large sequential data sets in a reasonable amount of time.

Strengths of MapReduce

MapReduce can combine data from many different sources, such as data from a file system, output from database queries, structured input files, web logs, etc. It can do some things that are difficult for SQL, particularly with processing large volumes of semi-structured or unstructured data, or with detecting patterns in video feeds or in sequences like DNA. It is well-designed for column-oriented databases that can easily fit into a key/value structure. It has high-fault tolerance, it can process large volumes of data with optimal performance, and it can execute more complicated analysis of the data than SQL alone can do.

Limits of MapReduce

MapReduce is not good at resolving all data computation problems, and is not a replacement for a relational database. It doesn't easily handle small reads. By itself, it can't do direct queries and can't process data directly from the database (although Apache has a database product called HBASE that works with Hadoop's HDFS and MapReduce framework to enable some queries.) It's not designed for multiuser interactive updates. Security is still an issue, as is network bandwidth.

Hadoop

Hadoop is a Java-based open-source framework designed for processing large datasets using Google's MapReduce algorithms. Hadoop clusters can scale to thousands of servers and can store and process petabytes to exabytes of data. It is optimized for scalability and throughput rather than latency. Many commercial products, including IBM's Big Insights, are based on Hadoop.

Structure

Hadoop has two main components—the HDFS file system and MapReduce processing. The Hadoop Distributed File System (HDFS) has a directory structure and user interface similar to the Linux file system. HDFS is distributed, so files can reside on any of the machines in the cluster. The Hadoop framework keeps track of where the files are located.

HDFS has a publisher/subscriber structure. The publisher system is called the *NameNode* and the subscribers are called the *DataNodes*. The NameNode stores the metadata, handles client requests, and monitors the DataNodes. The NameNode keeps track of how the data has been replicated and on which machines; the DataNodes house the actual data.

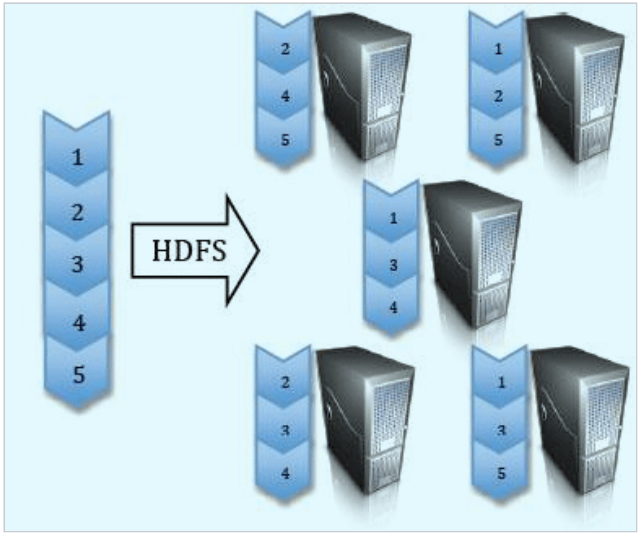
Rather than scale *up* to a more powerful and more expensive single machine, the designers of Hadoop scaled *out* to create clusters of multiple low-cost networked machines. If more power is needed or the files are too large, more machines can be added to the cluster.

Computing systems consisting of networked clusters of inexpensive machines have significant cost and scalability advantages, but they also pose significant reliability challenges, including processors with little inherent fault-tolerance and the risk of network failures. Hadoop obtains high reliability on unreliable hardware by using software fault tolerance technologies including data replication and monitoring processes.

Hadoop replicates the data into chunks of data called "splits," and puts the splits on different machines. The data is cached on the client machine until there is enough data to "fill" a data split, which is usually 64 or 128MB. Using large block sizes reduces the number of disk reads and writes and is one of the methods that Hadoop uses to maximize read and write performance.

When a client application needs to write a data block, it sends a message to the NameNode. The NameNode replies with the locations of three DataNodes to host the replicas. The first location that it recommends is on the machine closest to the client requesting the job. The second split goes onto a second machine on the same server rack, and the third goes elsewhere in the cluster. The client sends its data in packets to the first machine, which pipes it to the second, which pipes it to the third.

The diagram below shows how HDFS replicates a file. Each number represents a block of data that is part of one data file. Each block is replicated to three different machines, although this can be configured differently if more copies are needed.



"Awadallah, A. (n.d.). *Introducing Hadoop: The Modern Data Operating System.* Retrieved 09 20, 2012, from Stanford University: http://www.stanford.edu/class/ee380/Abstracts/111116-slides.pdf

When a client wants to read from a data file, it sends a message to the NameNode requesting the locations of its data. After the NameNode replies, the client will try to read the data on the closest DataNode first. But a read attempt can fail if the data block is corrupted, or if the DataNode is down, or if for some reason, the DataNode no longer has that particular data block. If the read fails, the client will try the next machine in the sequence. Since the data has been replicated, if there is a problem with the data on one DataNode, another DataNode can automatically take over processing the same data.

The NameNode processes all client requests. It periodically pings the DataNodes with a "heartbeat" report to find out if that DataNode is still functioning, and if its data copies are still readable. The reports are sent approximately every three seconds. If the NameNode fails to get a response in a reasonable amount of time, it will conclude that the DataNode is down. If this happens, the NameNode will send the jobs that the DataNode was running to a different DataNode, and will begin the process of replicating the data blocks from the failed DataNode onto another machine. Hadoop is "self-healing" because it can recover some problems like these by itself, without user intervention.

Consistency

Hadoop is built on a write-once, read-many model. Any number of readers can access the data simultaneously. A writer is granted a "lease" on a file and only one writer at a time can "own" the lease at any given time. When the writer completes, the lease is revoked and is available for a different writer. The lease does not prevent other readers from accessing the file even as it is being written.

This can be a problem, though, because of the existence of the replicas. If one replica is updated, there may be a lapse of time before the data can be replicated to the other copies. Although this happens quickly with HDFS, it is possible that a reader can get a "stale" copy of the data that hasn't yet been updated. This is termed "eventual consistency," because, although the data isn't immediately consistent, eventually all the copies will be identical.

MapReduce Job Management

HDFS is the storage component of Hadoop and MapReduce is the processing component. MapReduce works by generating key/value pairs, and performing operations on those pairs, taking advantage of HDFS to run its jobs in parallel. Hadoop has incorporated a system for running map and reduce jobs on its file system.

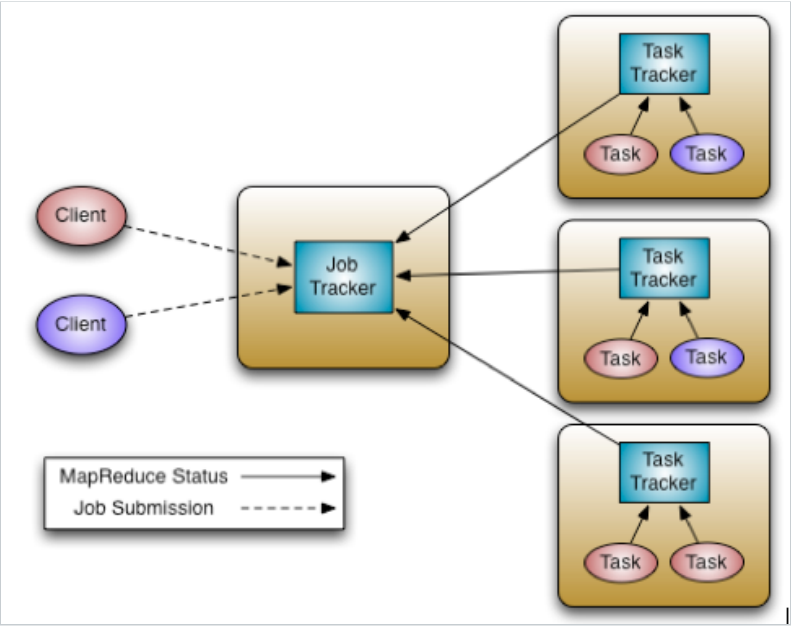
In its initial design, Hadoop set up a JobTracker to monitor the MapReduce jobs, and TaskTrackers to perform them. Since all the "administration work" was handled by the one JobTracker, this proved to be a performance bottleneck. This changed in version 2.0 and later of Hadoop. Because many stable versions are still at version 1.2 or under, and because the new design builds on the old one, it's important to understand both techniques.

Original MapReduce Job Management

Before version 2.0, Hadoop managed its jobs this way:

- The JobTracker responding to a client request for the job locates the data, determines how many mappers and reducers it will need, and assigns the mapper and reducer tasks to possibly many TaskTrackers.
- The TaskTrackers execute a loop that pings the JobTracker through its own series of heartbeat reports, to say that it's ready to start executing a job. The JobTracker schedules the task and selects a TaskTracker to run it. For a map task, the JobTracker will try to select a TaskTracker that is closest to the data to maximize data locality. For a reduce task, the JobTracker assigns the next available TaskTracker. Since the reducer is using mapper output as its input, there is no need to consider data locality.
- A TaskTracker can execute either map or reduce tasks, and often several simultaneously. The TaskTracker runs the actual code.

Pre-version 2.0 version of MapReduce



Murthy, A. (n.d.). *Apache Hadoop Yarn Background and an Overview*. Retrieved 12 3, 2012, from http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview

"Modern" MapReduce Job Handling-YARN

The design with just one Job Tracker limited scalability and often produced performance bottlenecks, so a new design was created termed YARN, which stands for "Yet Another Resource Negotiator." YARN splits the functions of the JobTracker into two separate processes—a *Resource Manager* that resides on the NameNode, and manages the resources for all applications running in the cluster and a per-application "*Application Master*" daemon that oversees its own particular application or job. Each DataNode also has its own "*Node Manager*" which controls the resources on a single machine. Resources are in "containers" of allotted memory, disk space, etc.

(White 2012, Apache 2013)

The Resource Manager

Within the Resource Manager are two components, the *Scheduler* and the *Applications Manager*. The Scheduler does nothing but scheduling. It will allocate containers to the application as needed and schedule its jobs accordingly. But it doesn't track jobs to see if they're still running, and it doesn't restart the application if it fails. These jobs are now the responsibility of the *Applications Manager*. The Applications Manager accepts job submissions and will restart the job if it fails.

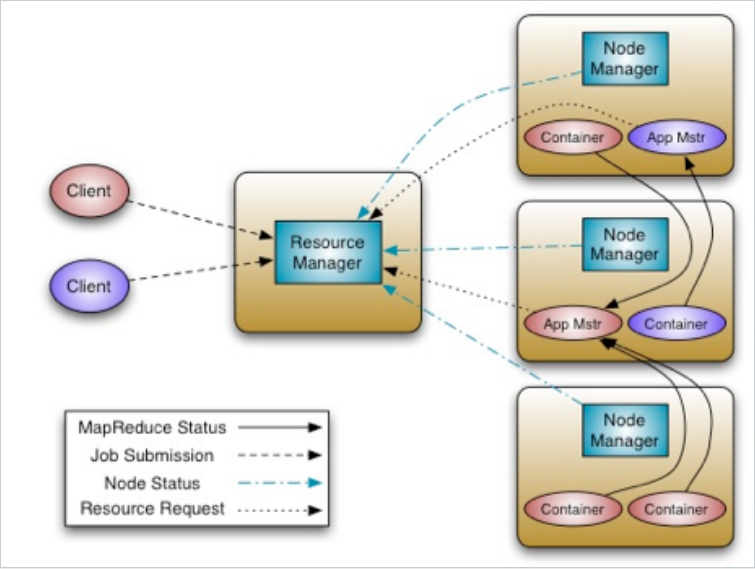
The *per-application Application Master* decides what resources it will need for the application. It may decide to run in the same JVM with the application, depending on the size of the job. If not, it will negotiate with the *Scheduler* for containers that will handle all of its map and reduce tasks. It also tracks the status of the jobs and reports their progress to the Applications Manager.
Each DataNode has its own "Node Manager" which controls the containers on their machine to ensure that they don't exceed their allotted resources. The Node Manager reports on the status of the resources to the Scheduler.

Lifecycle of a job with the YARN architecture

In the following illustration, the client submits a job to the Resource Manager. The Applications Manager responds to a client with the job ID number, notifies the Scheduler, and starts up the per-application Application Master for the application. The per-application Application Master starts running and negotiates with the Applications Manager. The Applications Manager notifies the client of the job status. All communication with the client happens through the Applications Manager in most cases.

The per-application Application Master then creates "resource requests" for the Scheduler. The per-application Application Master submits the requests, gets containers from the Resource Manager, and then negotiates with the Node Manager for the node on which the container resides to start the container. The per-application Application Master monitors the tasks of its application and requests alternate resources if one of its containers fails. The Applications Manager monitors all of the Application Masters and verifies that they're still running through heartbeat reports. If the Application Master does not respond, the Applications Manager will restart the application with a different Application Master.

Modern version of MapReduce



Murthy, A. (n.d.). *Apache Hadoop Yarn Background and an Overview*. Retrieved 12 3, 2012, from http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview

Advantages of YARN

Dividing the JobTracker into two components, a scheduler and application monitor, reduces one of the bottlenecks in the Hadoop framework, making it possible to run more jobs more efficiently. In previous versions of Hadoop, some tasks were designated as mapper tasks and some as reducer tasks. This could become a performance bottleneck if not enough mappers were designated. Reducer tasks would remain idle, waiting for the mappers to complete. With YARN, instead of allocating different tasks, an application can request different amounts of memory in its containers. So the number of mappers that can run is contingent on the amount of memory requested rather than on the designated number of mapper tasks.

There are some other major advantages to this approach. It is now possible to run algorithms other than MapReduce on the Hadoop framework. Even with MapReduce, it now becomes possible to run different versions of MapReduce even on the same cluster.

Joins with Hadoop and MapReduce

Hadoop is able to support both map-side and reduce-side joins in MapReduce. Both inner and outer joins are supported. It is possible to join data from different data sources, and to join data from more than two data sources.

A map-side join, also called a replicated join, performs the join as part of the map step, before the data is processed, so it is more efficient than a reduce-side join, but also harder to set up. A reduce-side join, also called a repartition join, is easier, but performance can be a factor because it reviews the entire dataset.

Map-side joins are only efficient if one dataset is small, and one is large. They require one entire table to be included in one mapper, so precise configuration requirements are necessary. Map-side joins are generally only worthwhile when there is a smaller dataset to join with a larger one. With a map-side join, the key from the large dataset is matched up to the smaller dataset and the join is done before the data is run through the mapper.

With a reduce-side join, the mapper creates a "join key" that is used to join the data sets. The reducer uses the join keys to join the mapper outputs, and then partitions the data. Finally it does a Cartesian join and then outputs the results.

Joins use data sources, which are similar to a table in a database. A data source can be one file, or multiple files, but the files must all share the same basic structure.

Suppose we want to do a reduce-side join to see if the budget for a movie has an effect on the kinds of ratings and reviews it will get. For the join to work, the entire file has to be formatted the same way.

Title	Year	Budget (from imdb)

"Les Miserables"	2012	$61,000,000
"Star Wars"	1977	$11,000,000
"Lincoln"	2012	$65,000,000
"The Curious Case of Benjamin Button"	2008	$150,000,000
"Spy Game"	2001	$92,000,000

Movie reviews might look something like this:

Title	Rating	Review
"Les Miserables"	8	"Good Anne Hathaway vehicle"
"Star Wars"	9	"Classic movie"
"Lincoln"	9	"Great period piece"
"The Curious Case of Benjamin Button"	7	"Not my favorite"
"Spy Game"	8	"The acting was superb!"

It is important to include the name or title of the data source in the mapper output. Along with the name of the data source, the mapper output has to contain the join key. The join key is the same for all the files to be joined, in this example, movie name, which is the "key" output of the mapper. The "value" output will be the name of the data source, along with the value to be output, for our Movies table, its budget and movie year.

Key	Value
"Les Miserables"	#Movies61,000,0002012
"Star Wars"	#Movies11,000,0001977
"Lincoln"	#Movies65,000,0002012
"The Curious Case of Benjamin Button"	#Movies150,000,0002008
"Spy Game"	#Movies92,0002001

A separate mapper program would run for each data source. The mapper output for the movie reviews data would have the movie name as the "key" and the data source, rating and review as its "value".

Key	Value
"Les Miserables"	#Reviews8"Great music"
"Star Wars"	#Reviews9"Classic movie"
"Lincoln"	#Reviews9"Great period piece"
"The Curious Case of Benjamin Button"	#Reviews7"Not my favorite"

"Spy Game"	#Reviews8"Superb acting"

The input to the reducer will contain duplicate keys with different data sources and values. The next step for the reducer is to create a Cartesian join. The reducer then sends the output to a third step called combine() which determines the type of join.

For our project, in the reducer step, we can sort the output by rating in descending order. And the output from this join will look something like this:

Title	Year	Budget (from imdb)	Rating	Review
"Star Wars"	1977	$11,000,000	9	"Classic movie"
"Lincoln"	2012	$65,000,000	9	"Great period piece"
"Les Miserables"	2012	$61,000,000	8	"Great music"
"Spy Game"	2001	$92,000,000	8	"Superb acting!"
"The Curious Case of Benjamin Button"	2008	$150,000,000	7	"Not my favorite"

(Holmes, 2012)

Related Apache Projects

Many Apache projects have been built for or are using the Hadoop framework, including:

- ***HBase*** is a column-oriented NoSQL distributed database that sits as a layer on top of HDFS. HBase is based on Google's BigTable and is optimized for random reads and writes on very large data tables. You read only the columns you need, which are grouped together in "column families". HBase does not , support secondary indexes, stored procedures or query languages such as SQL, or ACID transactions.
- ***Cassandra*** is also a fault-tolerant column-oriented distributed. Like HBase, Cassandra uses Google's Big Table column-oriented data model. Cassandra can be layered on Hadoop, but it is most frequently run on Amazon's DynamoDB. Cassandra also supports Pig and Hive, but may not support everything that Hadoop does.
- ***Hive*** is a data warehouse system that allows ad-hoc queries and analysis of large datasets. Hive uses a SQL-like language called *HiveQL* and stores its metadata in a relational database called "Derby." Because HBase and Cassandra lack their own versions of SQL, they are sometimes integrated with Hive to take advantage of Hive's version of SQL. Tables in Hive can be either internal or external. Internal tables are managed by Hive in its own warehouse directory. They can be used if the data is entirely created, maintained and deleted within Hive, but Hive also runs on HDFS files, which would be external to Hive. Hive will use MapReduce to process its queries if necessary.

Hive does not provide indexing capability, or the ability to modify records.

- ***Pig*** is a high-level imperative data query language that uses a number of steps instead of the descriptions of declarative languages such as SQL. Pig can work with any kind of data. It doesn't require a fixed schema. Instead it creates its own schema, based on the modules used to create it, and casts any field to null that doesn't fit into its derived schema, so the data doesn't have to all have the same structure. Pig is often used in *ETL* (Extraction, Transform and Load) processes.
- ***Mahout*** is a library with machine-learning algorithms that can be used with Hadoop's MapReduce, which gives it the ability to run on very large datasets. The library includes

algorithms for clustering, classification and recommenders.

- ***Sqoop*** is a set of tools to import or export relational database data into Hadoop through a JDBC connection string. It supports any database that is supported by JDBC. Tools within Sqoop will import a database table into HDFS or export an HDFS directory to a relational database table, etc. Sqoop also works with Hive.
- ***Zookeeper*** is a centralized service that coordinates the configuration of distributed systems. With distributed transactions there are many possible sources of error. To ensure that a transaction was successful takes some work for the programmer. Zookeeper works at that level to ensure that the jobs complete, or notifies the user when they don't.

(Holmes, 2012)

Apache Spark

The material in this section identified by "Optional Technical Detail" does not appear on the quiz or final exam.

Spark Overview

Apache Spark is a fast, versatile, distributed computation framework for executing code in parallel across many machines (Databricks). Developers at UC Berkeley and UC Berkeley's AMPLab created Spark to increase processing efficiency for large-scale iterative applications. Spark stores intermediate datasets in distributed memory for faster access. This is in contrast to Hadoop MapReduce, which writes to disk or other durable storage between steps.

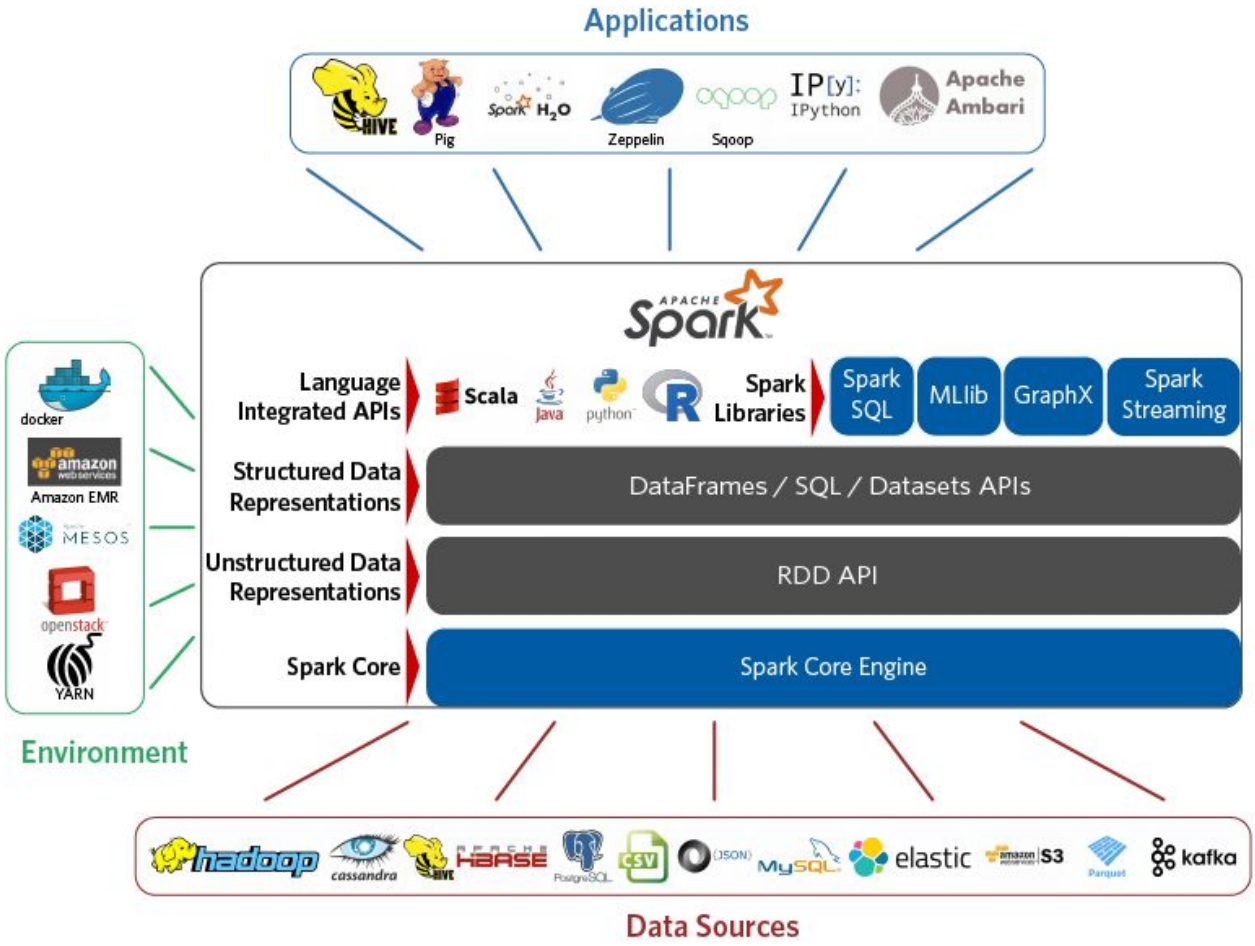
Resilient Distributed Datasets (RDDs) are the original Spark data structure and are one of the key concepts behind Spark. RDDs are immutable, partitioned, distributed, fault-tolerant collections of data. RDDs provide fault-tolerance by tracking their *lineage* instead of relying on replication. If a partition of an RDD is lost, the RDD has enough information to rebuild the lost partition (Zaharia et al., 2012). As Spark applications perform operations on RDDs, Spark creates intermediate RDDs. Programmers can specify which intermediate RDDs should persist in memory. Programmers can also specify whether the Spark application returns the results to the calling application or writes the results to a storage system. (Zaharia, et al., 2012)

Spark currently supports three data representation APIs: Resilient Distributed Datasets (RDDs), *DataFrames*, and *Datasets*. DataFrames and Datasets are built on top of RDDs and inherit RDD functionality. DataFrames and Datasets are higher-level APIs used with structured data. RDDs are a lower-level API, used for unstructured data. Spark *applications* in Spark 2.0 and later versions more frequently rely on DataFrame and Dataset APIs than the original, unstructured RDD API. Spark implements DataFrames, Datasets, and RDDs in Scala, Python, Java, and R language-integrated programming interfaces (Zaharia et al., 2012).

Developers can deploy Spark on a single computer or on a *cluster*. A cluster in the big data sense refers to up to thousands of interconnected computers in one datacenter. Big data clusters store and process up to many petabytes of data. Cluster users submit applications to the cluster for processing. Machines in the cluster distribute, track, and process submitted applications. Spark can use its own built-in *Standalone* resource scheduler or it can rely on *Apache Mesos* or *Hadoop YARN* cluster management.

The Spark ecosystem consists of the Spark core, Spark libraries and APIs, and a host of data sources, environments, and applications. The Spark ecosystem is illustrated in Figure 1, which is an interactive graphic.

Figure 1: The Spark Ecosystem



Roll over the graphic to learn more.

Adapted from: (Databricks. (2017). Step 1: Why Apache Spark?. *7 Steps for a Developer to Learn Apache® Spark™: Highlights from Databricks' Technical Content*)

At the center of the Spark ecosystem are the tightly integrated Spark Core Engine and Spark Libraries. Programmers can access Spark Core Engine functions and Spark Library functions through a single API.

The *Spark Core API* interacts with storage systems, manages memory, recovers from faults, and provides scheduling, distribution, and monitoring of applications across the cluster (Karau, 2015). The Spark Core can scale out from a single node to thousands of nodes. Spark can also be installed locally on Linux, Windows, MacOS, or Docker.

Spark does not manage its own storage, so it relies on other storage environments. Spark can create distributed datasets (Datasets, DataFrames, and RDDs) from any storage source supported by Hadoop, including local file systems, HDFS, Cassandra, HBase, and Amazon S3 (Apache Spark, n.d.-c). Spark supports text files and Hadoop key-value formats, SequenceFile and InputFormat (Apache Spark).

Optional Technical Detail: Spark Libraries

Spark SQL

The ***Spark SQL*** library adds support for structured data and query optimization. Spark SQL interacts with DataFrames and Datasets. Spark SQL supports SQL queries and Hive SQL queries. It also supports many data sources including Hive tables, Parquet, Avro, JSON, and the standard database connectors, JDBC and ODBC. With Spark SQL, programmers can submit SQL queries from within Spark commands. Spark 2.0 and later versions support ANSI 2003 Compliant SQL.

Spark Streaming

Spark Streaming processes live data streams while providing the same fault tolerance, scalability, and throughput as the Spark Core. Examples of live data streams are Twitter public streams and log file updates.

With Spark Streaming, Spark can ingest streaming data from sources like Kafka, Flume, or TCP Sockets and process the data with functions like map, reduce, or join. Spark's machine learning or graph processing libraries can then further analyze the processed data. (Apache Spark Overview)

Spark developers are currently working on an upgrade to Spark Streaming called Structured Streaming. It is ALPHA in the latest installation of Spark (2.1). Spark users can create *continuous applications* using Structured Streaming. Continuous applications combine input streams and static data. Structured Streaming builds an unbounded table with existing and incoming data. Spark users can perform ETL operations, ad-hoc queries, or batch jobs on the unbounded table. (Databricks, 2017)

Spark MLlib

Spark MLlib is Spark's advanced analytics library. MLlib allows data scientists and data engineers to run advanced analytics algorithms against large distributed collections of data. MLlib includes classification, regression, clustering, and collaborative filtering algorithms as well as a gradient descent optimization algorithm. MLlib algorithms can perform the following tasks: preprocessing (cleaning data), feature engineering, supervised learning, unsupervised learning, recommendation engines, and graph analysis. (Karau, 2015)

Machine Learning generalized workflow:

1. Preprocessing raw data
2. Manipulate the cleaned data based on the task to perform
3. Perform additional feature analysis, or introduce data from other sources
4. Build and test a model
5. Use the model for discovery, predictions, or for recommendations

(Karau, 2015)

Spark GraphX and GraphFrames

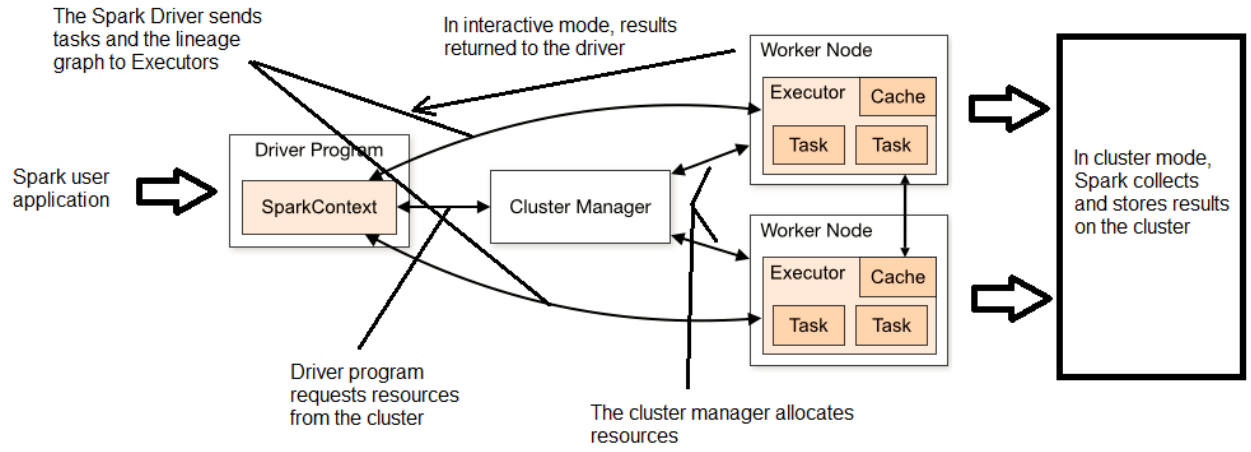
GraphX is Spark's distributed, parallel graph processing library. GraphX includes a set of fundamental graph processing operators. It also includes a collection of graph algorithms and builders to simplify graph analytics. One GraphX example is the Google PageRank algorithm. GraphX uses RDDs and is only available in Scala. (Apache Spark Overview)

Spark developers created GraphFrames as an upgrade to GraphX. It is not currently part of the core Spark installation but can be added as a Spark package. GraphFrames uses DataFrames instead of RDDs to take advantage of Spark's query optimization. GraphFrames also adds support for Python and Java along with Scala. (Databricks, 2017)

SparkR

R is an open source statistical and graphical language. RStudio is an integrated development environment (IDE) for running R. SparkR provides access to distributed DataFrames using R commands. SparkR can be used to connect to a Spark cluster through RStudio.

Figure 2. Spark Runtime Architecture



Adapted from: https://spark.apache.org/docs/latest/cluster-overview.html

Spark Cluster

A Spark Cluster is a collection of machines or nodes on which Spark is installed. Machines can exist on premises or in a shared datacenter. The Spark machines in a cluster have the roles of a Spark Master (also a cluster manager in Standalone mode), Spark Workers, and at least one Spark Driver. These roles can exist all on one machine or on different machines depending on the deployment mode. (Databricks, 2017)

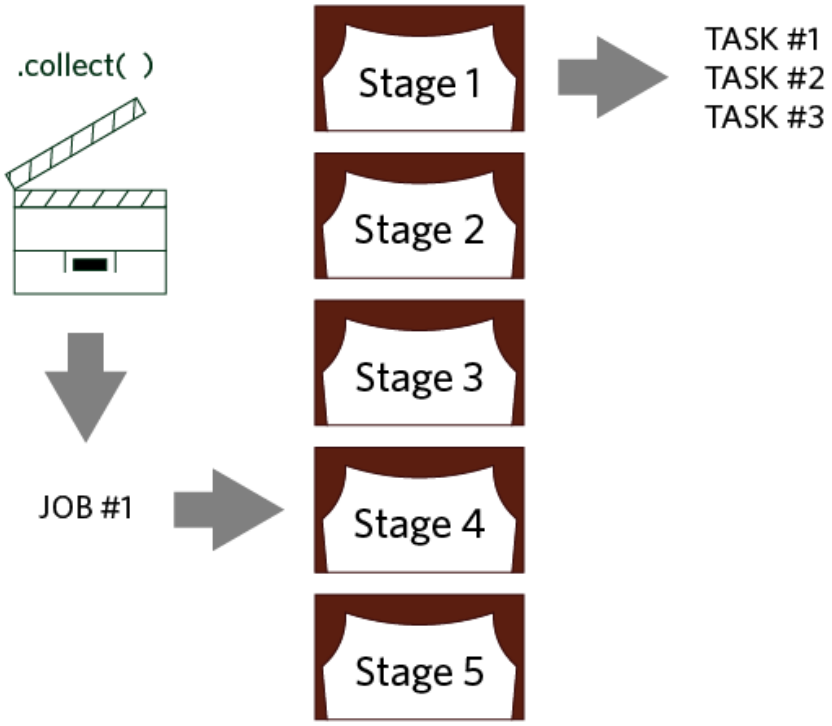
Spark Master

The Spark Master role differs depending on where Spark is installed. In Local mode the Spark Master runs on the same machine as the Spark Driver, Worker, and Executor. The Spark Master acts as a cluster manager in a Standalone deployment mode. In YARN and Mesos clusters, their cluster management handles the Spark Master role. (Databricks, 2017)

Spark Driver

The Spark Driver node runs the Spark Driver process. The Spark Driver process maintains relevant information about the Spark Application, responds to the Spark user's program, and distributes and schedules work on the Executors allocated by the cluster manager. The Spark Driver program converts the job into stages and stages into tasks. The Driver then distributes tasks for execution. (Databricks, 2017; Zaharia & Chambers, 2017 04)

Figure 3



Adapted from: Databricks. (2017)

In Local mode or interactive cluster mode, the machine where the application was submitted remains as the Driver for the duration of the application. It receives the computed results once the application completes. In cluster mode, the application files are submitted to the cluster. Spark then hands off the Driver role to one of the Worker nodes. In cluster mode, Spark collects and stores results on the cluster. (Karau, Konwinski, Wendell & Zaharia 2015 02)

Spark Application

A Spark Application is the code submitted by a Spark user to run in Spark. Once the Spark Application is running, it is represented by the associated Driver process and Executor processes. The Driver node manages the Spark Application. The Executors process the application and return results. (Zaharia & Chambers, 2017 04)

Spark Worker

The Spark Worker has a relatively simple but important role in the Spark ecosystem. It is responsible for launching an Executor JVM as directed by the Spark Master. (Databricks, 2017)

Spark Executor

A Spark Executor is a JVM container on a Spark Worker. The Spark Executor has an allocated amount of cores and memory. The Spark Executor executes tasks it receives from the Driver and caches data partitions. (Databricks, 2017)

Optional Technical Detail: Spark Unstructured APIs

The original, lower level, data abstraction for Spark is the Resilient Distributed Dataset (RDD). Spark can store RDDs in distributed memory and operate on them in parallel. The RDD concept is based on the 2010 paper, Spark: Cluster Computing with Working Sets and the 2012 paper, Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, written at UC Berkeley and UC Berkeley AMPLab by Matei Zaharia, Mosharaf Chowdhury, and several others. RDDs are unstructured and allow more granular control over the data, but do not see the optimization benefits that the higher-level DataFrame and Dataset APIs do.

Resilient Distributed Dataset (RDD)

“RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators” (Zaharia et al., 2012, p. 1). With RDDs, every row is a Java object (Zaharia & Chambers, 2017 04). Spark applications perform operations on each row of the RDD and return a new RDD. RDD operations fall into two categories, transformations (lazy) and actions (eager). Transformations update the query plan, in Spark called a *Directed Acyclic Graph (DAG)*. This allows Spark to evaluate the entire application to determine the most efficient way to run it. Actions cause the application to run, based on the DAG, and return results to the calling application or write results to durable storage.

Spark creates different types of RDDs as it applies different transformations. The table below displays some of the more common RDD types. Some environments like Cassandra have very specialized RDDs with the following functionality…

Table 1. Types of RDDs

Type of RDD	Description
ShuffledRDD	Created from operations like reduceByKey
MapPartitionsRDD	Created from operations like map and flatmap
HadoopRDD	Default format when reading a text file
CassandraRDD	Specialized RDD for interacting with a Cassandra database.

Spark offers two ways of creating a base RDD, the parallelize() method and loading data from external storage. Parallelize is more for testing since it requires the RDD be created in memory on one machine. The more common method for creating RDDs is to load data from external storage. Here is an example Scala command for loading a text file. (Karau, Konwinski, Wendell & Zaharia 2015 02)

```
val input = sc.textFile(file:///homedir/reports/largeReport.md)
```

When a Spark application references an RDD, Spark returns an iterator to all of the objects in the RDD. For the base RDD, each object represents one line in the source file. For new RDDs created through transformations, each object can contain, for example, a single word or a key-value pair. RDD objects are *typed* which allows Spark, at compile time, to do logic checking for non-existent methods or mismatched data types as well as syntax checking.

The Spark application example below steps through the relationships between transformations, RDDs, objects contained in each RDD, and an action. The transformation code corresponding to the associated RDD in each step is highlighted.

Text file poem.txt:
Two roads diverged in a yellow wood
And sorry I could not travel both
And be one traveler long I stood
And looked down one as far as I could

1. Spark command, written in Scala, to create the Base RDD called wordCount and resulting HadoopRDD:

```
val wordCount = sc.textFile(“poem.txt”)
```

HadoopRDD (created from the text file)

Two roads diverged in a yellow wood

And sorry I could not travel both
And be one traveler long I stood
And looked down one as far as I could

Adapted from: (Cloudera, 2014 07)

2. The following Spark command, written in Scala, divides the lines into words, counts the words, and returns the counts to the Spark Driver.

```
wordCount.flatMap(line => line.split(" ")).map(word =>(word,
1)).reduceByKey(_ + _).collect()
```

The following list further breaks down the Spark application by highlighting each transformation and the resulting intermediate RDD.

a. The flatmap transformation creates an RDD of individual words by splitting strings at white spaces and results in the following MapPartitionsRDD.

```
wordCount.flatMap(line => line.split(" ")).map(word =>
(word, 1)).reduceByKey(_ + _).collect()
```

MapPartitionsRDD (created from flatmap)

Two	sorry	One	down
Roads	I	traveler	one
diverged	could	long	as
In	not	I	far
A	travel	stood	as
yellow	both	And	I
Wood	And	looked	could
And	Be		

Adapted from: (Cloudera, 2014 07)

b. The map transformation creates an RDD of key-value pairs. The '1' value paired with the occurrence of each word represents a count of one and results in the following MapPartitionsRDD.

```
wordCount.flatMap(line => line.split(" ")).map(word =>
(word, 1)).reduceByKey(_ + _).collect()
```

MapPartitionsRDD (created from mapping to (word, 1))

(Two, 1)	(sorry, 1)	(one, 1)	(down, 1)
(roads, 1)	(I, 1)	(traveler, 1)	(one, 1)
(diverged, 1)	(could, 1)	(long, 1)	(as, 1)
(In, 1)	(not, 1)	(I, 1)	(far, 1)
(a, 1)	(travel, 1)	(stood, 1)	(as, 1)
(yellow, 1)	(both, 1)	(And, 1)	(I, 1)
(wood, 1),	(And, 1)	(looked, 1)	(could, 1)
(And, 1)	(be, 1)		

Adapted from: (Cloudera, 2014 07)

c. The reduceByKey transformation sums the occurrences of each key (word in this case) and creates an RDD of key-value pairs and results in the following ShuffledRDD.

```
wordCount. flatMap(line => line.split(" ")).map(word =>
(word, 1)).reduceByKey(_ + _).collect()
```

ShuffledRDD (created when summing the word counts)

(Two, 1)
(roads, 1)
(diverged, 1)
(In, 1)
(a, 1)
(yellow, 1)
(wood, 1),
(And, 3)
(sorry, 1)
(I, 3)

```
...
```

Adapted from: (Cloudera, 2014 07)

d. The .collect() action instructs Spark to run the application and return the results to the Spark Driver console.

```
wordCount.flatmap(lambda line: line.split()).map(lambda
word: (word, 1)).reduceByKey(lambda v1,v2: v1+v2).collect()
```

Spark output to console:

```
res10: Array[(String, Int)] = Array((diverged,1), (long,1),
(Two,1), (down,1), (one,2), (as,2), (travel,1), (looked,1),
(yellow,1), (And,3), (not,1), (traveler,1), (a,1),
(sorry,1),(be,1), (could,2), (I,3), (wood,1), (roads,1),
(in,1), (far,1), (both,1), (stood,1))
```

Spark characterizes RDDs by the five main properties listed in Table 1 below. Spark uses these properties to determine how to schedule and execute the Spark application. A use case for the optional partitioner property would be to control the partitioning of a key-value pair RDD.

Table 2: RDD Properties

Property	Meaning
partitions()	Return a list of Partition objects
iterator(p, parentIters)	Compute the elements of partition p given iterators for its parent partitions (computation of partition splits)
dependencies()	Return a list of dependencies
partitioner()	(Optional) Return metadata specifying whether the RDD is hash/range partitioned
preferredLocations(p)	(Optional) List nodes where partition p can be accessed faster due to data locality

Adapted from: (Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma. J., McCauley, M., Franklin, M., Shenker, S. & Stoica, I. (2012, 04). *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.* Retrieved from: https://amplab.cs.berkeley.edu/publication/resilient- distributed-datasets-a-fault-tolerant-abstraction-for-in-memory-cluster-computing/)

Spark applications create RDDs from data in durable storage or through transformations on other RDDs. Spark users can configure persistence levels for RDDs. In general, the original or base RDD for a Spark application should be stored on disk only. The base RDD is typically very large. Spark applications typically only perform analysis on subsets of the base RDD. Child RDDs that a Spark application references more than once should be persisted in memory. The cache() method is short for the default MEMORY_ONLY storage level (Apache Spark, n.d.-c). With the persist() method, Spark users can set a specific storage level. Spark can store RDDs as serialized or deserialized. Serialized data is compressed for efficient storage and distribution but is not in a format for processing. Deserialized data is larger than serialized data and is in a processing-ready format. Table 2 below summarizes Spark RDD storage levels.

Table 3: Spark RDD Storage Levels

Storage Level	Meaning
MEMORY_ONLY(Default)	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they are needed.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the remaining partitions on disk.
MEMORY_ONLY_SER (Java and Scala)	Store the RDD as serialized Java objects. This is generally more space-efficient but also more CPU-intensive than storing deserialized objects. Spark must deserialize objects before processing them. Like the MEMORY_ONLY level, partitions that are not cached are recomputed on the fly.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, except that partitions that do not fit in memory will be saved to disk instead of recomputing them on the fly.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires enabling off-heap memory.

Adapted from: Apache Spark. (n.d.-c). Programming Guide. Retrieved from: http://spark.apache.org/docs/latest/programming-guide.html

Spark tracks each RDD's lineage so Spark can rebuild a lost RDD partition if an Executor or Worker fails. RDD lineages consist of *narrow* and *wide dependencies*. Narrow dependencies occur when each partition of a parent RDD has at most one child RDD partition dependence. Wide dependencies occur when partitions in the parent RDD have multiple child RDD partition dependencies. Transformations like map use narrow dependencies and transformations like join typically use wide dependencies. (Zaharia et al., 2012)

When cluster hardware fails, RDDs can lose some of their partitions. In most cases, it is more efficient to rebuild the lost RDD partitions from lineage rather than to replicate all partitions. For RDDs with long lineages and wide dependencies, it is faster to recover the lost partitions from durable storage. For those complex RDDs, Spark can checkpoint them to durable storage for fault tolerance.

Here are the first couple of lines from a sample Spark application, written in Scala, to filter lines from a log file stored in a Hadoop Distributed File System (HDFS).

1. Create a base RDD from a log file

```
val lines = sc.textFile("hdfs://<path to log file>")
```

2. Create a filtered RDD from the base RDD, lines

```
val errors = lines.filter(_.startsWith("ERROR"))
```

3. Cache the errors filtered RDD for iterative key word search filtering

```
errors.cache()
```

Broadcast Variables

Broadcast Variables are immutable objects that Spark originates from the Driver node and replicates across the cluster to all Worker nodes. Pre-staging Broadcast Variables saves time during application execution. One common example of this is joining a very large dataset with a small dataset. Spark replicates the small dataset across the cluster. When each Executor executes the join, it uses the Broadcast Variable and the local partition of the very large dataset (Zaharia & Chambers, 2017 04)

Accumulators

Accumulators can be described as the opposite of a Broadcast Variable. They create mutable variables across nodes in the cluster that each Executor can modify. Executors update their local Accumulator during application execution. The Driver tracks Accumulator updates and reports its current value. (Zaharia & Chambers, 2017 04)

Optional Technical Detail: Spark Structured APIs

Spark developers added two higher-level APIs, DataFrames (introduced with Spark 1.3 in August 2015) and Datasets (introduced with Spark 1.6 in January 2016). DataFrames and Datasets, as with RDDs, are immutable, distributed, fault-tolerant collections of data. DataFrames and Datasets allow Spark to increase processing efficiency and reduce data size for structured data. With Spark 2.0 (introduced in July 2016), developers consolidated the DataFrame API under the Dataset API as Dataset[Row]. Row indicates an element of type Row which is evaluated by Spark at runtime.

Although RDDs are no longer the primary data abstraction API, they are still a key piece of the Spark ecosystem. DataFrames and Datasets are built on RDDs. Spark compiles DataFrames and Datasets down to RDDs before processing them (Zaharia & Chambers, 2017 04). Programmers will continue to use RDDs for unstructured data like video, audio, and text streams, for granular control of physical data distribution, or when data does not require a columnar format. (Damji, 2016 07)

DataFrame

Spark developers introduced DataFrames, SQL, *Catalyst* optimization, and structure in Spark 1.3. DataFrames include a schema and query optimization but they lack type safety. Spark evaluates DataFrame syntax at compile time but evaluates DataFrame logic at runtime.
DataFrames organize data into named columns with schema information similar to a table in a relational database or a data frame in R or the Python Pandas library. DataFrames are available in Scala, Java, Python, and R.

Structured data falls into two categories – structured and semi-structured. Semi-structured data refers to a CSV file, JSON file, or other similar layout where they define rows and columns but not data types. Spark users can use *type inference* to let Spark specify the data types. Spark users can also choose to specify the schema and column types before reading the data file into Spark. Structured data, like a database table or a Parquet file, will define the rows, columns, and data types.

Spark processes data using its own data types. At compile time, Spark sees each DataFrame row as a generic Row type. At runtime, Spark converts from language specific types to Spark types. The table below shows data type comparisons between Spark, Scala, Java, and Python.

Table 4. Comparison of Spark, Scala, Java, and Python Data Types

Spark Type	Scala Value Type	Java Value Type	Python Value Type
ByteType	Byte	byte or Byte	int or long
ShortType	Short	short or Short	int or long
IntegerType	Int	int or Integer	int or long
LongType	Long	long or Long	int or long

FloatType	Float	float or Float	float
DoubleType	Double	double or Double	float
DecimalType	java.math.BigDecimal	java.math.BigDecimal	decimal.Decimal
StringType	String	String	str
BinaryType	Array[Byte]	byte[]	bytearray
TimestampType	java.sql.Timestamp	java.sql.Timestamp	datetime.datetime
DateType	java.sql.Date	java.sql.Date	datetime.date
ArrayType	scala.collection.Seq	java.util.List	list, tuple, or array
MapType	scala.collection.Map	java.util.Map	dict
StructType	org.apache.spark.sql.Row	org.apache.spark.sql.Row	list or tuple
StructField	StructField with DataType contents.	StructField with DataType contents.	StructField with DataType contents.

Here are some examples of DataFrames in action.

For the examples below, this is a JSON representation of a list of open houses. For Spark, JSON files should list one JSON object per line. The file name is listings.json.

listings.json

{"Neighborhood": "Beacon Hill", "Listing Price": "650,000", "Address": "28 Myrtle St., Apt 4", "Date": "4/23/2017", "Time": "12:30-2:30", "Bedrooms": "2", "Bathrooms": "1", "Parking": "Street", "OutdoorSpace": "Patio", "Square Feet": "980"}
{"Neighborhood": "Fenway", "Listing Price": "425,000", "Address": "144 Hemenway St., Apt 2B", "Date": "4/22/2017", "Time": "1:30-3:30", "Bedrooms": "1", "Bathrooms": "1", "Parking": "Street", "OutdoorSpace": "none", "Square Feet": "650"}
{"Neighborhood": "South End", "Listing Price": "750,000", "Address": "64 Concord St, Apt 5", "Date": "4/22/2017", "Time": "12:00-3:00", "Bedrooms": "2", "Bathrooms": "2", "Parking": "Deeded", "OutdoorSpace": "Roof Deck", "Square Feet": "1280"}
{"Neighborhood": "Dorchester", "Listing Price": "450,000", "Address": "23 Leroy St., Apt 7", "Date": "4/23/2017", "Time": "12:00-2:00", "Bedrooms": "2", "Bathrooms": "1", "Parking": "Street", "OutdoorSpace": "Patio", "Square Feet": "780"}

1. This application uses Scala and *SparkSession* (represented as spark) to read in a JSON file. SparkSession includes a DataFrame reader method (.read) and a JSON format specifier (.json).

```
val openHouses = spark
                .read
                .json( “<path>/listings.json” )
```

Output:

openHouses: org.apache.spark.sql.DataFrame = [Address: string, Bathrooms: string ... 8 more fields]

2. Like with RDDs, DataFrame transformations are lazy. Spark will not execute anything until the application calls an action. Here is a simple line of code to display the top two lines of the JSON file.

```
openHouses.take(2)
```

Output:

res11: Array[org.apache.spark.sql.Row] = Array([28 Myrtle St., Apt 4,1,2,4/23/2017,650,000,Beacon Hill,Patio,Street,980,12:30-2:30], [144 Hemenway St., Apt 2B,1,1,4/22/2017,425,000,Fenway,none,Street,650,1:30-3:30])

3. The code below is an example of sorting a DataFrame. After executing the sort, Spark will create a new, sorted DataFrame. DataFrames are immutable so Spark will not alter an existing DataFrame.

```
val sortedOpenHouses = openHouses.sort("Bedrooms", "Bathrooms")
```

Output:

sortedOpenHouses: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Address: string, Bathrooms: string ... 8 more fields]

4. To show the sorted table, use the display() method.

```
display(sortedOpenHouses)
```

Figure 4. Screen Capture of Output

```
1 %scala
2 display(sortedOpenHouses)
```

▸ (1) Spark Jobs

Address	Bathrooms	Bedrooms	Date	Listir
144 Hemenway St., Apt 2B	1	1	4/22/2017	425,0
28 Myrtle St., Apt 4	1	2	4/23/2017	650,0
23 Leroy St., Apt 7	1	2	4/23/2017	450,0
64 Concord St, Apt 5	2	2	4/22/2017	750,0

5. The explain() method shows the execution plan for transforming listings.json into the DataFrame created for sortedOpenHouses. Explain plans are similar to a DAG from RDD transformations.

```
sortedOpenHouses.explain()
```

Output:

== Physical Plan ==

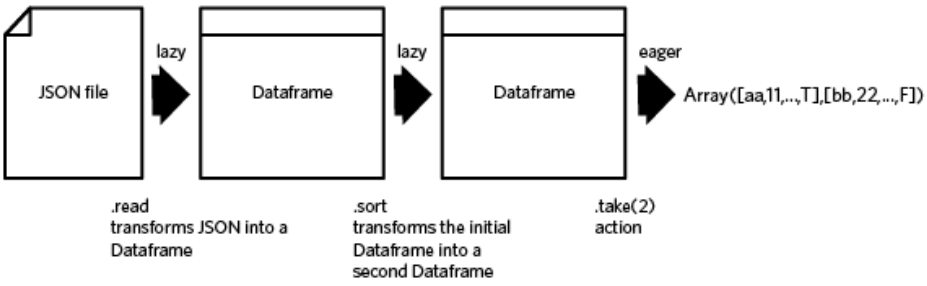
*Sort [Bedrooms#195 ASC NULLS FIRST, Bathrooms#194 ASC NULLS FIRST], true, 0

+- Exchange rangepartitioning(Bedrooms#195 ASC NULLS FIRST, Bathrooms#194 ASC NULLS FIRST, 200)

+- *FileScan json [Address#193,Bathrooms#194,Bedrooms#195,Date#196,Listing Price#197,Neighborhood#198,OutdoorSpace#199,Parking#200,Square Feet#201,Time#202] Batched: false, Format: JSON, Location: InMemoryFileIndex[dbfs:/FileStore/tables/xf7c7p0w1497450711820/listings.json], PartitionFilters: [], PushedFilters: [], ReadSchema:struct<Address:string,Bathrooms:string,Bedrooms:string,Date:string, Listing Price:string,Neighborh...

6. This graphic Illustrates the execution plan for transforming listings.json into the DataFrame created for sortedOpenHouses and running an action.

Figure 5.



Adapted from: (Zaharia & Chambers, 2017 04)

7. Spark integrates Spark SQL into the DataFrame API. Spark users can query DataFrames using SQL code through any language-integrated API (Scala, Python, Java, R). Spark SQL queries benefit from the same query optimizations that DataFrame operations do. Here is an example of querying a DataFrame using Spark SQL.

First, create a table or view from the DataFrame. In this case, the code creates a view.

```
openHouses.createOrReplaceTempView("openHouses")
```

Then, run a Spark SQL query to match the DataFrame command (val sortedOpenHouses = openHouses.sort("Bedrooms", "Bathrooms")).

```
val SQL_sortedOpenHouses = spark.sql("""
        Select * from openHouses
        Order By Bedrooms, Bathrooms
      """)
```

8. To view the resulting table, use the display() method.

```
display(SQL_sortedOpenHouses)
```

Figure 6.

Dataset

Datasets are an evolution of DataFrames and include schemas or *Case Classes* for Scala and *JavaBeans* for Java. Datasets require additional overhead to compute, are *type-safe*, and, like DataFrames, have a schema. The benefit of Datasets is that Spark can leverage something called *encoders* to process them extremely efficiently. Datasets are only available in Scala and Java because of their compile-time type safety. Compile-time type safety refers to syntax and logic checking at compile-time instead of at runtime.

For scenarios involving large amounts of data, Spark users can utilize DataFrames to sort and filter the data. Spark users can then convert the smaller, filtered and sorted DataFrames into Datasets for further processing (Zaharia & Chambers, 2017 04). The concept of large amounts of data can be difficult to quantify. It essentially indicates where the performance hit for converting a large amount of data into a Dataset is substantial, use a DataFrame as an intermediate step.

Typed and Un-typed APIs

Scala and Java both have compile-time type-safety so they can use Datasets. Dataset is the most restrictive API. The Dataset API will catch syntax and logic errors at compile time. The DataFrame API will detect errors like a non-existent function at compile time but will not catch something like a non-existent column until runtime. The benefit of using DataFrames is that all four language-integrated APIs (Scala, Python, Java, and R) can access them.

Table 5.

Language	Main Abstraction
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python	DataFrame (no compile-time type safety)
R	DataFrame (no compile-time type safety)

- The [T] in Dataset[T] refers to the data structure. Scala uses the Case Class to describe the data.
- Dataset[Row] refers to a collection of rows in the form of a relation(table)

Adapted from: (Databricks: 7 Steps for a Developer…, 2017; KDnuggets:Apache Spark Key Terms, Explained, 2016 06; Damji, 2016 07)

Optional Technical Detail: Spark Programming

Spark Operations

Spark Shell

The Spark Shell is the Driver program that runs the main() function of the Spark Application and creates the Spark Context. To open the Scala shell, type

```
bin/spark-shell
```

To open a Python shell, type

```
bin/pyspark
```

SparkSession

SparkSession replaces the Spark Context in Spark versions 2.0 and later. It provides a command line interface for submitting Spark commands. The advantage over Spark Context is that SparkSession connects to the SQL context and Hive contexts automatically. SparkSession also contains information about the environment so SparkConf does not have to be explicitly created. (Damji, J., 2016 08 15)

Here is an example for creating a SparkSession.

```
val spark = SparkSession
        .builder()
        .appName("SparkSessionZipsExample")
        .config("spark.sql.warehouse.dir", warehouseLocation)
        .enableHiveSupport()
        .getOrCreate()
```

(Damji, J., 2016 08 15)

SparkContext

The SparkContext represents the connection to the Spark cluster. It provides an interface for running Spark commands. SparkConf must be specified first, before creating a SparkContext. For SQL or Hive commands, the SQL or Hive contexts must also be loaded. (Zaharia & Chambers, 2017 04)

Code Example:

```
SparkContext(sparkConf).set("spark.some.config.option", "some-value")

val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

(Damji, J., 2016 08 15)

SparkConf

SparkConf sets configurations for the environment.

Code Example:

```
val sparkConf = new SparkConf().setAppName

("SparkSessionZipsExample").setMaster("local"))
```

(Damji, J., 2016 08 15)

Transformations and Actions

Spark executes two types of operations, Transformations and Actions. Transformations only update the *Directed Acyclic Graph (DAG)* and do not perform any computations. The DAG is essentially a plan for how the application is going to be run once an action is called. Transformations are lazy operations and Actions are eager operations.

The table below lists some common Transformations. A more complete list is maintained at http://spark.apache.org/docs/latest/programming-guide.html#transformations.

Table 6. Common Spark Transformations

Transformation	Meaning
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
distinct([numTasks]))	Return a new dataset that contains the distinct elements of the source dataset.
reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.

join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

(Apache Spark. (n.d.-c). Spark Programming Guide. Retrieved from: http://spark.apache.org/docs/ latest/programming-guide.html)

Actions perform processing based on the DAG and return results. The table below lists some common Actions. A more complete list is maintained at http://spark.apache.org/docs/latest/programming-guide.html#actions.

Table 7. Common Spark Actions

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

Directed Acyclic Graph (DAG)

A DAG is similar to an execution plan in an RDBMS. Prior to executing actions, check the DAG to make sure the application will operate efficiently. For instance, shuffle operations like join or reduceByKey should happen after filter operations to minimize network traffic and processing time. To display the DAG using Scala, add .toDebugString to the end of the transformations as in the word count example below.

```
%scala
val txtFileSc = sc.textFile(“<location of the text file>”)

%scala
txtFileSc.flatMap(line=>line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _).toDebugString
```

The output below illustrates the DAG from the word count example. DAG output is read from the bottom up. The bottom two lines show how Spark creates a HadoopRDD from the txt file and then transforms the HadoopRDD to a MapPartitionsRDD for further processing. Then the flatMap command transformation creates a second MapPartitionsRDD. The map transformation creates a third MapPartitionsRDD. Finally, the reduceByKey transformation sums keys from all RDD partitions, and results in a ShuffledRDD.

```
(2) ShuffledRDD[50] at reduceByKey at <console>:39 []
    +-(2) MapPartitionsRDD[49] at map at <console>:39 []
        | MapPartitionsRDD[48] at flatMap at <console>:39 []
        | /FileStore/tables/0fbmvtrd1496106271742/reLsTrv.txt
         MapPartitionsRDD[44] at textFile at <console>:35 []
        | /FileStore/tables/0fbmvtrd1496106271742/reLsTrv.txt
         HadoopRDD[43] at textFile at <console>:35 []
```

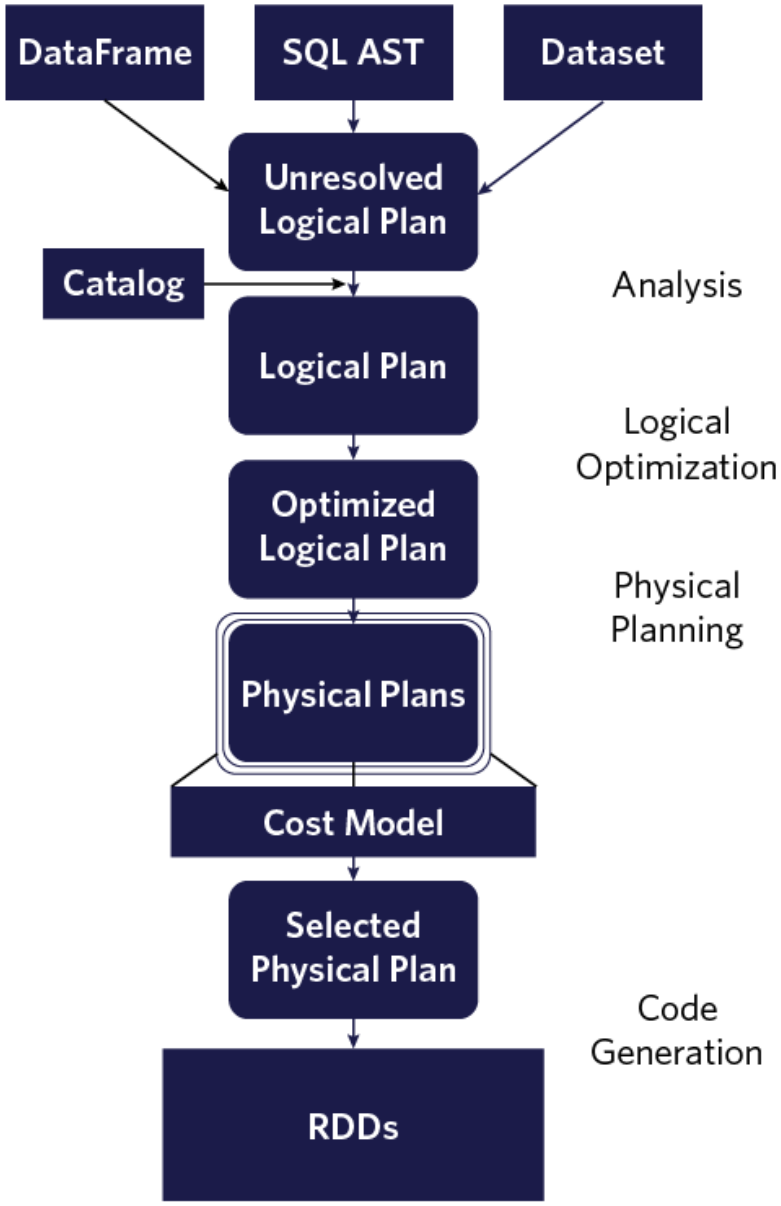
Tungsten

Tungsten is Spark's technology for improving memory and CPU efficiency for Spark applications. Tungsten bypasses a lot of unnecessary JVM overhead. It creates very efficient bytecode for Spark Executors to process.

Catalyst

Catalyst is Spark's query optimizer. It is able to optimize Spark jobs submitted using structured DataFrame APIs or Dataset APIs. Catalyst decomposes structured data into RDDs for Spark cluster operations. The graphic below illustrates how Catalyst evaluates Spark jobs and breaks them down into logical and then physical execution plans.

Figure 7. Catalyst Optimization



Spark Environments

Spark executes applications in clustered and non-clustered environments. In a non-clustered environment (Local Mode), all roles are on the same machine. For a clustered environment, Spark users submit Spark jobs to a cluster, either in interactive mode or in cluster mode. In interactive mode, the Spark user's machine is the Driver and must stay connected to the cluster during job execution. Spark returns results to the user's machine. In cluster mode, Spark users submit a .jar, .py, or similar file to the cluster. The cluster identifies a Driver and Executors, runs the application, and outputs results to durable storage.

Table 8.

Mode	Driver	Worker	Executor	Master
Local	Runs on a single JVM	Runs on the same JVM as the driver	Runs on the same JVM as the driver	Runs on the same JVM as the driver
Standalone	Can run on any node in the cluster	Runs on its own JVM on	Each worker in the cluster will	Can be allocated arbitrarily where the master is started

		each node	launch its own JVM	
Yarn(client)	On a client, not part of the cluster	YARN NodeManager	YARN's NodeManager's Container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for Executors.
YARN(cluster)	Runs within the YARN's Application Master	Same as YARN client mode	Same as YARN client mode	Same as YARN client mode
Mesos(client)	Runs on a client machine, not part of Mesos cluster	Runs on Mesos Slave	Container within Mesos Slave	Mesos' master
Mesos(cluster)	Runs within one of Mesos' masters	Same as client mode	Same as client mode	Mesos' master

Adapted from Damji, J. & Farooqui, S. (2016, 09). 7 Steps to Mastering Apache Spark 2.0.

Non-Clustered Environment (Local Mode)

With Local Mode, Spark users can install a fully functional Spark environment on a single machine. Spark users can run the same code in Local mode as they would on a Spark cluster. Spark developers provide the non-clustered environment for proof-of-concept and testing.

In Local Mode, the Spark Driver, Spark Executors, and Spark Master run in the same JVM. To start Spark in local mode, run

```
./bin/spark-shell
```

on a machine with Spark installed. This will launch a Spark console where the Spark user can enter Spark commands. (Karau, Konwinski, Wendell & Zaharia 2015 02)

Clustered Environments

To take advantage of the full power of Spark's distributed computing engine, Spark users can submit jobs to a cluster. Spark users configure *spark-submit* options for the specific application and environment. Spark-submit launches the Spark application on the cluster and results are written to durable storage.

Spark Submit

Spark users process Spark applications on a cluster using spark-submit. Spark-submit has several options to specify what should run on the cluster: how to reach the cluster, what type of cluster will be used, what resources are needed, and how to identify the Spark application. For complex applications with many dependencies, Spark users can use build tools for Scala (sbt) and Java (Maven). For complex Python scripts, Spark users can add supporting libraries using the --py-files option with spark-submit. Spark users can also install Python supporting libraries to machines in the cluster by using a Python package manager.

This is an example of submitting a Python script to a Spark Standalone cluster.

```
bin/spark-submit --master spark://host:7077 --executor-memory 10g
my_script.py
```

The following table shows --master **options**

Table 9.

Value	Explanation
spark://host:port	Connect to a Spark Standalone cluster at the specified port. By default Spark Standalone masters use port 7077.
mesos://host:port	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050.
yarn	Connect to a YARN cluster. Set the HADOOP_CONF_DIR environment variable to point the location of your Hadoop configuration directory, which contains information about the cluster.
local	Run in local mode with a single core.
local[N]	Run in local mode with N cores.
local[*]	Run in local mode and use as many cores as the machine has.

Adapted from: (Karau, Konwinski, Wendell & Zaharia 2015 02)

The following is an example of submitting a Spark application, written in Python, in YARN client mode. Each line of code contains an explanation. Client mode indicates the Spark user's machine must remain connected to the cluster for the duration of the application. It also indicates that the results will be collected at the user's machine.

Table 10. Submitting a Spark application written in Python in YARN client mode

Spark Line of Code	Explanation
$ export HADOOP_CONF_DIR=/opt/hadoop/conf	Connection information about the YARN cluster for Spark.
$ ./bin/spark-submit \	Spark-submit command with options below
--master yarn \	Indicates application will run on YARN
--py-files somelib-1.2.egg,otherlib-4.4.zip,other-file.py \	Supporting Python libraries

`--deploy-mode client \`	Spark user's machine will be the Driver. Spark will return results to the Spark users's machine. Options are client(interactive) and cluster
`--name "Example Program" \`	Name for the application for monitoring
`--num-executors 40 \`	Total number of executors for the application
`--executor-memory 10g \`	Memory allocated per executor
`my_script.py`	Application code

Adapted from: (Karau, Konwinski, Wendell & Zaharia 2015 02)

Standalone

Standalone mode is Spark's built in resource scheduler. This is quickest and easiest method for setting up a Spark cluster to run distributed applications. To specify Standalone mode, set the --master option to spark://host:port.

To set up a Standalone cluster, use the following steps.

- Download appropriate Apache Spark release from: http://spark.apache.org/downloads.html
- Extract the contents to each machine designated for the cluster
- Create a file in the conf directory called slaves on the Master
    - Add the hostname for each Worker to the slaves file
- Run sbin/start-master.sh on the Master to start the Master instance
- Run sbin/start-slaves.sh on the Master to start a slave instance on each host listed in the slaves file
- Further options and configuration information is available on: https://spark.apache.org/docs/latest/spark-standalone.html#spark-standalone-mode.

YARN

YARN is the Apache Hadoop cluster manager. For Spark installations on YARN, the YARN Resource Manager replaces the Spark Master. YARN clusters have a couple of advantages over the Spark Standalone cluster. YARN has dynamic resource allocation to make sure submitted jobs are using the appropriate amount of cluster resources. If machines allocated to a certain application are near capacity limits, YARN will allocate additional machines. If machines allocated to a certain application remain idle, YARN will re-allocate the machines for other requests. Another advantage of running Spark jobs on a YARN cluster is that it can allocate tasks next to HDFS blocks. This greatly speeds data access.

For additional information regarding running Spark on YARN, refer to Apache Spark documentation: https://spark.apache.org/docs/latest/running-on-yarn.html.

Mesos

Spark users can deploy Spark to an Apache Mesos cluster. In this configuration, the Mesos master replaces the Spark Master as the cluster manager. Mesos can run applications in fine-grained or course-grained mode. Fine-grained applications scale up and down depending on how busy the application is. Fine-grained applications work well in shared clusters because cluster managers can re-allocate resources where they are needed. The downside is the time required to spin up new resources for a rapidly scaling application. Course-grained applications acquire a fixed set of resources and retain the resources. This is helpful for some streaming applications that need the resources instantly available.

For additional information regarding running Spark on Mesos, refer to Apache Spark documentation: https://spark.apache.org/docs/latest/running-on-mesos.html.

Amazon EMR

Amazon Elastic MapReduce (EMR) clusters support Spark installations. Spark can leverage the EMR file system to access data in Amazon S3. To build a Spark cluster on EMR, access the Amazon EMR console at: https://console.aws.amazon.com/elasticmapreduce/. (Amazon Web Services)

For additional details on running Spark in EMR, refer to the Amazon AWS Spark documentation: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark.html.

References

Amazon Web Services. Apache Spark. Retrieved from: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark.html

Apache Spark. (2015, 04). Advanced Apache Spark Training - Sameer Farooqui (Databricks). Retrieved from: https://www.youtube.com/watch?v=7ooZ4S7Ay6Y

Apache Spark. Retrieved from: http://spark.apache.org/

Apache Spark Overview. Retrieved from: https://spark.apache.org/docs/latest/

Cloudera. (2014, 7 23). Introduction to Apache Spark Developer Training. Retrieved from: https://www.slideshare.net/cloudera/spark-devwebinarslides-final

Damji, J. & Farooqui, S. (2016, 09). 7 Steps to Mastering Apache Spark 2.0. Retrieved from: http://www.kdnuggets.com/2016/09/7-steps-mastering-apache-spark.html

Damji, J. (2016, 07 14). A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets: When to use them and why. Retrieved from: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Damji, J. (2016, 08 15). How to use SparkSession in Apache Spark 2.0: A unified entry point for manipulating data with Spark. Retrieved from: https://databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html

Databricks. (2017). 7 Steps for a Developer to Learn Apache® Spark™: Highlights from Databricks' Technical Content

Databricks. Retrieved from: https://docs.databricks.com

IBM Big Data University. (2016, 05 02). Course BD0211E: Spark Fundamentals I. Retrieved from: https://cognitiveclass.ai/courses/what-is-spark/

Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015, 02). *Learning Spark*. Sebastopol, CA: O'Reilly Media, Inc.

Lee, D. & Damji, J. (2016, 06). Apache Spark Key Terms, Explained. Retrieved from: http://www.kdnuggets.com/2016/06/spark-key-terms-explained.html

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma. J., McCauley, M., Franklin, M., Shenker, S. & Stoica, I. (2012, 04). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Retrieved from: https://amplab.cs.berkeley.edu/publication/resilient-distributed-datasets-a-fault-tolerant-abstraction-for-in-memory-cluster-computing/

Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S. & Stoica, I. (2010, 06). Spark: Cluster Computing with Working Sets. Retrieved from: https://amplab.cs.berkeley.edu/publication/spark-cluster-computing-with-working-sets-paper/

Zaharia, M. & Chambers, B. (2017, 04). *Spark: The Definitive Guide, 1st Edition*. Sebastopol, CA: O'Reilly Media, Inc.

Kinds of Big Data DBMS

Some of the most common models for Big Data include key-value stores, where data is stored in key-value pairs, column-oriented data stores, where data is stored in columns rather than in rows, and graph databases, which enables the data to be stored in relationships with other data.

BigTable

According to its creators at Google, “A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.”

BigTable is essentially a map, built on top of the Google File System. A BigTable file is made up of rows and columns, much like a table in a relational database. It maintains column keys in groups called column families, and orders the data by row key. BigTable exhibits some features of a column-oriented database, and some features of a key/value data model.

Data Structure - Rows and Columns

Since the data is sorted by row key, choosing the right row keys so that frequently accessed rows are stored together will help to maximize data locality and thereby increase performance. Each range of row keys and its corresponding data is kept in a file called a “tablet.” These tablets are replicated on different machines for parallel processing and fault tolerance. Reads and writes on a single row are atomic, although transactions on ranges of rows are not supported.

Columns are stored by row, similar to a row structure in a DBMS. Unlike a row structure in a standard relational database, different rows can have different sets of columns and values.

Column keys are stored separately in groups called column families. The column key consists of *family name: identifier* and can consist of any number of columns greater than one. A column family must be established before columns can be added to it.

All the data in the column family is usually of the same data type. In this way, the columns can easily be compressed to save on storage and retrieval time. Access control is stored at the column family level, and disk and memory accounting also occurs at this level.

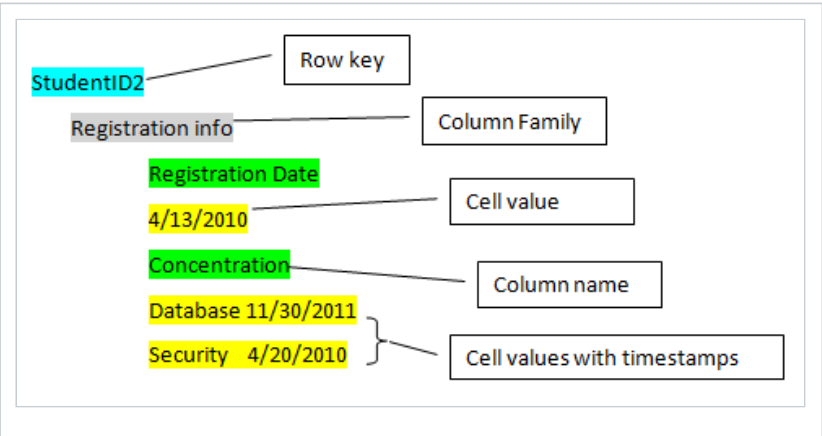
Rows and columns in BigTable meet in cells, much like the cells in a relational database. But unlike a relational database table, each cell can contain multiple versions of the same data. To make this work, BigTable assigns timestamps to each version of the same data in a cell, and different versions are stored by the most recent version first. The way a timestamp is structured can be set either by BigTable, or by the application itself, and the length of time to keep a time stamped data version can also be configured. With the timestamp feature in place, it is possible to query for all values of a given cell that changed in the last week, for example.

Locality Groups

Groups of column families can form locality groups. Column families, which are normally accessed together, can be assigned as a locality group, which can improve read performance. Some tuning parameters can be specified at the locality-group level. Compression also works well on locality groups. Some implementations of BigTable, like Apache’s HBase, lack the locality group feature.

So the access to the data follows this pattern:
Tablet -> Row Key -> Locality Group -> Column family -> Column -> Timestamp -> Value

As an example, if the Bigtable is storing student data, it might look something like this:



Sources:

- Chang F. , Dean, Ghemawat, Hsieh, & Wallach, 2006
- Harris, 2006

- (modeled after) George, HBase vs. BigTable Comparison, 2011, p. 20

Internal Structure

Internally, data is stored in files called a “Sorted Strings Table” (SSTable) which is a key/value map structure developed by Google. At the physical level, the SSTable is a series of blocks, with a block index. The block index is read into memory when the SSTable is opened, and read from memory when the SSTable needs to be accessed. This reduces disk reads, and improves performance.

A BigTable implementation includes a library linked into every client to cache tablet locations, a master server and several tablet servers. The master server manages the tablet servers. It registers a tablet server when it comes on, and deactivates it when it is unavailable or no longer needed. The master also manages load-balancing of the tablet servers.

The tablet server manages access to its tablets, which could range from just a few to many thousands. The tablet server also controls read and write access to the tablets. When a client wishes to read or write data, the client communicates directly with the tablet server, not the master. This prevents the bottleneck with the one master that limited the performance of early implementations of the Google File System.

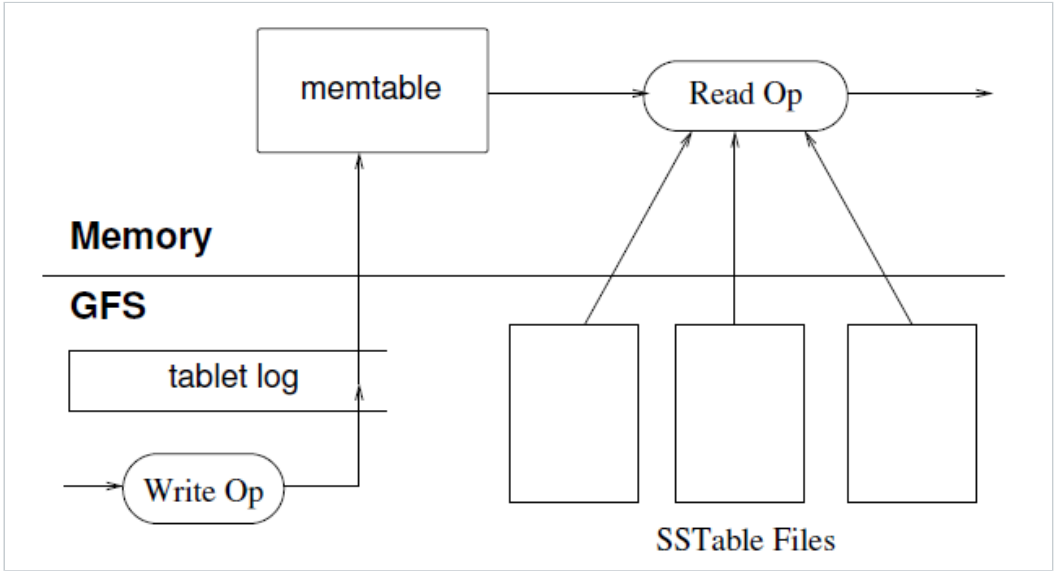
Tablet servers shard the tablets when they grow too large. Since the tablets are organized by row key, sharding is a relatively simple process. The tablet server “breaks” the set of row keys into two separate tablets.

Caching with BigTable

Bigtable uses two levels of caching – the Scan Cache and the Block Cache.

The Scan Cache caches the actual key/value pairs from the SSTable interface, and is useful for applications that read the same data repeatedly.

The Block Cache caches the physical blocks read in from GFS when an SSTable is opened. Queries that read data that has already been read can take advantage of the Block Cache to minimize disk transfers.



Chang F. , Dean, Ghemawat, Hsieh, & Wallach, 2006

Reads and Writes

A write operation, or mutation, is immediately written to a commit log file, and then written to an in-memory memtable. When the memtable reaches a certain size, the updates are written to disk, and the memtable is cleared.

A data read occurs on a merged view of the SSTables and the memtable, so the reads reflect the latest updates.

Updates to multiple tablets are included in the same commit log file. The commit log file contains the record of updates, and “redo points” that point to places in the commit log that involve the data on the tablet. Both the commit log file and memtable are sorted lexicographically so finding the data for the tablet is relatively simple. These operations are illustrated in the following diagram.

Harizopoulos S. A., 2009

In traditional row-store architecture data is stored in tables by rows. The primary key is likely to be the first entry in the row, and there is one value for each attribute in the row. Rows are physically stored together as a unit, and they may be sorted by the primary key. This row-store architecture works well for OLTP databases, where queries usually need to return many attributes from a few rows. Updates on the entire row are the most efficient because they involve only a single storage access.

In a column-store data model, data is physically stored by columns, usually in the order of the original rows. Many queries still need to process the data by the row, so most column-oriented databases offer some relational database type interface. From a logical view, the data appears to be the same as it would with a row-store.

For instance, this is the way data might appear in a row-oriented database, with each row stored as a unit:

Date	Store	Product	Customer	Price
2/1/2013	Walmart	Pickles	Harry	$2.69
2/5/2013	Target	Jacket	Katie	$35.99
2/28/2013	Sears	Toolkit	Joe	$25.99

With a column-store architecture, each column is stored in its own data structure.

Date	Store	Product	Customer	Price
2/1/2013 2/5/2013 2/28/2013	Walmart Target Sears	Pickles Jacket Toolkit	Harry Katie Joe	$2.69 $35.99 $25.99

Column-stores are excellent for data analytics and business intelligence. They work well for ad hoc queries that read large amounts of data. To get just the values of *Product* in a row-store, the scanner would read through each of the rows to get the values. With a column-store, the scanner only has to read the *Product* column.

Because queries in data warehousing environments tend to aggregate values, processing can occur within just a single column or just a few columns of values. Once accessed, entire columns can reside in memory, optimizing performance. For this type of query, it is generally more efficient to loop through values of an individual column than it would be to go through the data row-by-row looking for individual column values, and then aggregating those values.

For some queries it is useful to reconstruct the original row or instance. In the example, the row can be easily reconstructed because the order of the rows is preserved. For instance, the row for *2/5/2013*, which is the second value for date, matches up to the second values of *Target* in the store column file, *Jacket* in the product column file, etc. If necessary, the columns can store NULL values to maintain the order.

Many implementations of column-store architecture, such as Vertica, rely on “projections” which create column subsets of the original row. One such projection for our sample data might include “Date,” “Product,” and “Price.” Another might include “Store” and “Customer.” Vertica also requires a “super projection” for each original table that consists of every column in the row.

The point at which the row is reconstructed during the execution of a query can make a big difference in performance. During a query plan, attributes may need to be added to the intermediate results to be operated on at a different point. As the query plan finishes, these intermediate results are output to the user. This process is called “materialization.”

“Early materialization” constructs the tuples as early as possible in the query, so the column-store needs to retain some information about the original instance. Sometimes the same tuple may need to be accessed at different points in the query. For instance, some attributes may need to be located in a “where” clause, but their values may be needed later on in the query. If “early materialization” is not used, the attributes may need to be accessed again, which could impact performance.

“Late materialization” constructs the tuples as late as possible within the query after some processing has occurred. This usually improves performance because the operations can be performed on the compressed data and may result in fewer rows that need to be reconstructed.

Because the data values in an individual column, such as price, are all of the same type, compression algorithms work well to compress the data in column store architectures. Once compressed, more of this data can be read in from the disk, which means fewer disk accesses, and better read performance.

All commercial column-oriented database implementations now support some form of SQL, although at different levels. Some, such as Sybase IQ, are totally ANSI-compliant.

Sources:

1. Abadi, Myers, DeWitt, & Madden, 2007
2. Lamb, et al., 2012

Document Stores

Document-oriented database implementations are more sophisticated than basic key-value stores and are somewhat similar to relational databases. Rather than just a key/value structure, data is stored in a table-like document, with different types of documents either within a single database, or within table-like structures within a database. A document-oriented database may be schema-less, like CouchDB and SimpleDB, or it can have a flexible schema, like MongoDB. Most implementations allow for secondary indexes.

Document-oriented database implementations assume that documents encapsulate and encode data in standard formats or encodings such as XML, YAML, JSON, and BSON, or industry formats such as PDF or the Microsoft Office document formats.

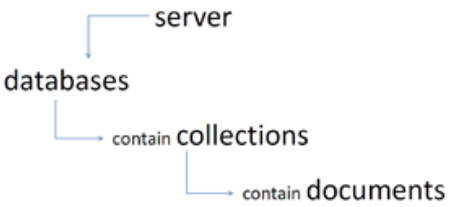
Different document stores organize documents in different ways. Some organize documents in collections. This is similar to the way that relational database tables organize tables as unordered collections of rows. It is different than relational database tables, because the rows of a relational table must have the same structure, while the documents in a collection can have very different structure.

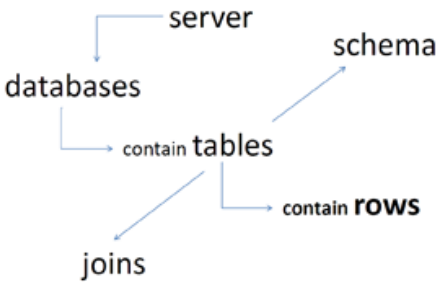
Document stores index and retrieve documents using unique document keys. Most document-oriented databases also provide application programming interfaces (APIs) that support retrieval of documents based on their content. The three major vendors of document-oriented databases are CouchDB by Apache, MongoDB, supported by 10gen, and Amazon’s SimpleDB.

MongoDB

MongoDB is a document-oriented data store with some of the features of a relational database, though not all. MongoDB supports indexes, and has its own query optimizer, but does not support joins, or ACID transactions. Writes on a document are atomic, much like writes on a row are atomic in BigTable. MongoDB runs on a publisher/subscriber architecture, similar to BigTable, and uses range partitioning to horizontally scale.

A *document* in MongoDB corresponds roughly to a row in a database table, and contains many *fields*, which are comparable to *columns*. A group of documents is called a *collection*. A collection is conceptually equivalent to a table in an RDBMS, except that it does not enforce a schema. A group of collections is contained within a single *database*. The following illustration compares the elements of an RDBMS structure with that of MongoDB.





MongoDB Structure

RDBMS Structure

In a typical RDBMS structure, databases contain tables that contain rows. The tables implement a schema, so any row in the table has to contain all the same fields as any other row in the table. Identical fields are defined by identical data types. If there is no value for a particular field in a row, NULL values may be inserted. A field contains a single value. Joins are permitted to locate data more easily, and to make queries more efficient.

In the MongoDB structure, databases contain collections that contain documents. Schemas are not enforced, but dynamic, meaning fields can be added to one document in a collection, but not all. An example will help to illustrate this.

For example, suppose that a collection contains student records. One document from that collection could look like this, with fields for _id, name and grade point average (gpa):

```
var thisdoc = {
_id: ObjectId("5099803df3f4948bd2f98391"),
name: { first: 'Mitt', last: 'Romney' },
gpa: NumberLong(40)
```

But for another document in the same collection, a graduation date attribute is added. The original document remains the same. The modified document now includes the yearOfGraduation:

```
var thisdoc = {
_id: ObjectId("5099803df3f4948bd2f98391"),
name: { first: 'Barack', last: 'Obama' },
gpa: NumberLong(40)
yearOfGraduation: new Date('Jun 03, 1979'),
}
```

Not only can documents in a collection have different elements or groups of elements, but also elements that are the same can have different data types between documents. The data types contained in a document can be a string or number data type, but can also be types like arrays and objects. It is possible to create a document that contains other documents, as in this example from the MongoDB manual.

```
var mydoc = {
_id: ObjectId("5099803df3f4948bd2f98391"),
name: { first: 'John', last: 'Backus' },
birth: new Date('Dec 03, 1924'),
death: new Date('Mar 17, 2007'),
contribs: [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
awards: [
{ award: 'National Medal of Science',
     year: 1975,
     by: 'National Science Foundation' },
{ award: 'Turing Award',
     year: 1977,
     by: 'ACM' }
     ]
     }
```

This document has as its first field, *_id*. This is the first field on every MongoDB document. It is the primary key field, and must contain a unique value that is never updated. The value (but not the name of the field) can be user-specified, and can contain a natural identifier, like a *studentID*, or it can be specified as an *ObjectID* as in the example. If it is specified as an *ObjectID* or if it is not included, MongoDB will generate it automatically. The *_id* field can also be specified as a sequence number and incremented automatically, like a sequence in Oracle or SQL Server.

Other fields in the example include:

> *name,* which contains another document with the fields "*first*" and "*last*", and their values.
> *birth* and *death,* which are simple Date formatted dates.
> *Contribs*, which contains an array of strings and
> *awards*, which contains an array of documents. (MongoDB, 2013, p.191)

Since a collection has a dynamic schema, and since it can have any number of elements, in theory, it would be possible to put all the data into a single collection. Although MongoDB will allow this, but it is not considered to be good design because:

1. Logically it makes more sense to separate the data.
2. It is less efficient to locate particular data fields if all the data is in one collection.
3. Documents are indexed *per collection*, so if all the data is together in one collection, the index created would be huge, and not very efficient.
4. Storing similar documents together is better for data locality.

Create, Read, Update, Delete

MongoDB supports the basic operations Create, Read, Update and Delete.

The *insert()* command is used to insert documents into collections and is conceptually equivalent to the SQL INSERT command. If an *_id* field is not explicitly specified in the command, MongoDB will generate the _id field and supply its value with a unique ObjectID. The *insert ()* command can also be used for batch inserts.

Reads are carried out using either the *find()* command or the *findOne()* command. The *find()* command is equivalent to a SQL SELECT command, and returns a result set, or cursor, containing all the documents that satisfied the query conditions. The *findOne()* command returns one and only one element that satisfies the query.

The *update()* command updates documents or fields in documents that are contained in the result set of a query. This command can also be used to add or delete fields from a document.

MongoDB has an interesting feature, not found in the standard SQL UPDATE command, which is an *upsert* flag. If the *upsert* flag is set, and the document is not found during the update, a new document will be inserted.

The *remove()* command will delete documents from a collection.

Indexes

Indexes in MongoDB are defined per collection. It is possible to create a secondary index on a single field, or on a composite collection of fields. As with any index in any database, there is additional overhead on write operations to create and maintain the index.

Queries use only one index at a time, and MongoDB's query optimizer will find the best one for the query. If, however, the query includes additional fields not contained in the index, MongoDB will perform a table scan, so performance can be an issue.

Data Modeling

MongoDB does not support joins, so to model relationships, it is important to represent the relationships explicitly within MongoDB.

To model a one-to-one relationship in MongoDB, the easiest way is to include all the related fields in a single document.

There are two ways to model a one-to-many relationships. One way to do this is to embed a document or set of documents within another document. In an RDBMS, this would essentially be the same as denormalizing a set of multi-valued child records and including them in the same record as the parent.

Suppose the design was for an online ordering system, and the data model contained orders and line items. In a traditional RDBMS design, that portion of the model might look like the ERD on the left, with one *Order* and many child *OrderItems*:

```
var orders = {
          _id: ObjectId("5099803df3f4948bd2f98391"),
          orderNumber: NumberLong(00003)
          orderDate: new Date('Jun 23, 1912'),
          customer: { first: "John", last: "Doe" },
          orderItems: [
                  {
                    linenumber NumberInt(4),
                    product_id: "53",
                    price: 1200
                   },
                   {
                    linenumber NumberInt(4),
                    product_id: "15",
                    price: 50
                    }],
          }}
          }
```


RDBMS model of a parent-child relationship The same relationship modeled in MongoDB with an embedded collection

Embedded data models are useful for one-to-many relationships, like the one above, where *order items* are usually only relevant in the context of the parent record, *Orders*. All the information is together in one place, and because of this, read performance could be optimized, depending on the application requirements. This model usually involves duplication, which can lead to data anomalies—there is another way.

References

The alternative to embedding documents to model one-to-many relationships is to create a "normalized" data model in MongoDB with *manual references*. Although MongoDB does not support joins as such, it does permit the use of references, and offers two different techniques.

To implement *manual references*, the *_id* field of one document is referenced in the second document. The following example illustrates how to use manual references for a collection that contains courses and room locations.

```
course_id = ObjectId()
          db.courses.insert({
          "_id": course_id
          "name": "CS799 Advanced Database Management"
          "professor": "Schudy"
})

room_id = ObjectId()
          db.roomlocation.insert({
          "_id": room_id
          "building": "Fuller"
          "room":"lab3"
          "occupant_id": course_id
          "class nights": {"Tuesday","Thursday"}
})
```

A query on *room_id* returns the attribute named *occupant_id,* which has the value of *course_id*. A second query on *course_id* would return the document with that *course_id* and with the *name* "CS799AdvanceDatabaseManagement" with *professor* "Schudy".

There is a second method called *DBRefs* that will allow references to a separate document in a separate collection or even separate database. But to resolve these references, extra queries are needed, which impacts performance.

Consistency

MongoDB handles consistency differently in different situations. If the client reads the primary key, consistency is guaranteed. Any reads of the primary key will be consistent with the latest updates. However, if the client reads from a secondary index, they may be reading from a "stale" copy of the data. By default, eventual consistency is provided for everything except reads on the primary key.

Reads and writes on a single document are atomic, and consistency is guaranteed on the read for a single document. However, for writes on multiple documents, it is not strict consistency. MongoDB offers a "two-phase commit" (MongoDB, 2013, p. 567). But this is not automatic; it is controlled by the application, and it still results in eventual consistency.

> Because only single-document operations are atomic with MongoDB, two-phase commits can only offer *transaction-like* semantics. It's possible for applications to return intermediate data at intermediate points during the two-phase commit or rollback (MongoDB, 2013).

Security

MongoDB uses roles to provide access to database objects, and within the roles, privileges are granted. Some privileges are granted at the database level, and some for the entire system. Roles can also be combined.

A Database User role can have either *read* or *readWrite* access to a particular database. *Read* privileges permit read access to the database, search access, and more (MongoDB, 2013). In addition to all the read privileges, readWrite privileges allow the user to insert, update and delete records from the collection.

Database Administration roles are also established at the database level. Here, too, there are two levels of user privileges, *dbAdmin* and *userAdmin*. With a dbAdmin role, a user can create a collection, create and update indexes, monitor performance and other administrative tasks. A userAdmin role permits the user to handle user administration for that database. In other words, a user with a userAdmin role can create, modify or delete users to get access to that database. However, a userAdmin role only applies to user administration.

MongoDB also provides for an *AnyDatabase* role that will permit a user to read or write to any database, or perform *dbAdmin* or *userAdmin* tasks on any database.

A *clusterAdmin* role permits various administrative tasks at the system level, such as repairing or dropping a database, and many tasks related to sharding. It is the only role that applies to the system, rather than to a single database instance.

Replication

MongoDB provides replication in two different ways. For very large clusters, it uses publisher/subscriber replication, much like the Google File System and Hadoop. For smaller-sized clusters, three machines, called *mongods* are designated for replicas. Of those, one becomes the primary, and the others are the secondaries. The client writes to the primary, and updates are asynchronously replicated to the secondaries.

MongoDB Lab

Go to Module 6 "MongoDB Lab" page to access the interactive MongoDB Lab learning object.

Key-Value Stores

A key-value data store is a simple, schema-less data representation, consisting of a primary key and its data value. The most popular products based on key-value structures are Amazon's DynamoDB, Riak, and Voldemort, which are based on Dynamo. Scalaris, Oracle's BerkelyDB, Oracle NoSQL, and Google's LevelDB are also key-value implementations. Key-value stores are excellent choices for web applications, which often do not involve complex queries.

A typical relational database schema has not only a primary key, but also different column names, each of which could have a different data type. All the records in the table contain values (or NULLs) for each of these columns.

In a key-value store, the keys are unique IDs, and the corresponding "value" is a blob of data. It is generally the responsibility of the programmer to parse the data into meaningful columns. Most key-value NoSQL database implementations allow the basic "CRUD" operations ("Create" (or Insert), "Read", "Update" and "Delete").

Queries are generally direct lookups on the key. Key-value data stores are optimized for reads, so read performance is usually very high. Secondary indexes are generally not available, although that is changing. Amazon recently released a limited secondary index capability for Dynamo DB.

Most implementations of key-value data stores use some form of replication and so use an "eventually consistent" model. Scalaris handles their implementation differently. Replications are synchronous, so all copies are updated in a single transaction. Scalaris also handles ACID transactions, but most implementations do not.

DynamoDB

DynamoDB is a key-value schema-less NoSQL distributed database system developed by Amazon. The database designers opted for availability over consistency, reasoning that most applications would be able to give up strict consistency in exchange for more uptime. "Customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados."

Data Model

A DynamoDB database consists of tables. Each table contains items, and each item contains different attributes. A primary key is required for each table.

Each attribute and its corresponding value is a key-value pair. An attribute can be single-valued or multi-valued. An attribute with multiple values is considered to contain a set. No duplicate values are allowed in a multi-valued set, and NULL values are not allowed for any attribute. Unlike a relational database schema, each item in a table can contain a different collection of attributes from any other item in the table.

For instance, if DynamoDB were storing a table of student data, it might look something like this:

```
    {
  ID: 101
      Name: Public, John Q.
      School: Metropolitan College
Registration Date: 4/13/2012
  Concentration: {Security, Networking}
}

    {
  ID: 102
      Name: Smith, Mary
      School: College of Arts and Sciences
Registration Date: 4/13/2012
  Advisor: Jones, William
}
```

The first item, with *ID* 101 contains a multi-valued set for "Concentration", while the second one, with item *ID* 102, did not. And the second item, with *ID* 102, contained an "Advisor" attribute that the first item, with *ID* 101, does not.

Keys and Indexes

A primary key can be one attribute, as *ID* is above in the example with the student data. If the key is only one attribute, Amazon builds a hash index on just that one attribute.

But if the primary key is a composite primary key, Dynamo allows for a "range key." Then data is stored in order by the primary key, but then by the range key, providing the equivalent of a composite primary key.

If a range key exists for the table, Dynamo will also build up to 5 "local secondary indexes" for that table, consisting of a total of 20 attributes spread across all the indexes. If only a primary key is defined for the table, secondary indexes are not allowed. The indexes will allow queries on one particular attribute that is not the primary key. Indexes in Dynamo speed up locating data, but they are not used in joins, as they would be in a relational database.

Amazon's Dynamo DB Developer Guide provides an example of how the data might look for discussion forum data in a table called "Thread." The data in the table might look something like this.

Thread

ForumName (Hash key)	*Subject* (Range key)	*LastPostDateTime*	*Replies*
"S3"	"aaa"	"2013-03-15:17:24:31"	12
"S3"	"bbb"	"2013-01-22:23:18:01"	3
"S3"	"ccc"	"2013-02-31:13:14:21"	4
"S3"	"ddd"	"2013-01-03:09:21:11"	9
"EC2"	"yyy"	"2013-02-12:11:07:56"	18
"EC2"	"zzz"	"2013-01-18:07:33:42"	0
"RDS"	"rrr"	"2013-01-19:01:13:24"	3
"RDS"	"sss"	"2013-03-11:06:53:00"	11
"RDS"	"ttt"	"2013-02-22:12:19:44"	5
...	...	...	...

Amazon, p.224

Suppose you wanted to see how many threads were posted in the last month. Without a local secondary index, you would have to do a scan of the entire table. With an additional range key or local secondary index on *LastPostDateTime*, the index would look like the following:

LastPostDateTime

ForumName (Hash key)	*LastPostDateTime* (Range key)	*Subject*
"S3"	"2013-01-03:09:21:11"	"ddd"
"S3"	"2013-01-22:23:18:01"	"bbb"
"S3"	"2013-02-31:13:14:21"	"ccc"
"S3"	"2013-03-15:17:24:31"	"aaa"
"EC2"	"2013-01-18:07:33:42"	"zzz"
"EC2"	"2013-02-12:11:07:56"	"yyy"
"RDS"	"2013-01-19:01:13:24"	"rrr"
"RDS"	"2013-02-22:12:19:44"	"ttt"
"RDS"	"2013-03-11:06:53:00"	"sss"
...	...	...

Amazon, p.225

This structure would make it relatively easy to query for each thread name/post date combination.

But now suppose you wanted to get the number of replies for a given forum and date period. There are two ways to accomplish this in Dynamo:

1. The "Replies" attribute could be declared as a non-indexed attribute in the local secondary index, LastPostIndex. The additional attribute is called a "projection."

   If "Replies" are included, the index would look like this:

   LastPostIndex

ForumName (Hash key)	LastPostDateTime (Range key)	Subject	Replies
"S3"	"2013-01-03:09:21:11"	"ddd"	9
"S3"	"2013-01-22:23:18:01"	"bbb"	3
"S3"	"2013-02-31:13:14:21"	"ccc"	4
"S3"	"2013-03-15:17:24:31"	"aaa"	12
"EC2"	"2013-01-18:07:33:42"	"zzz"	0
"EC2"	"2013-02-12:11:07:56"	"yyy"	18
"RDS"	"2013-01-19:01:13:24"	"rrr"	3
"RDS"	"2013-02-22:12:19:44"	"ttt"	5
"RDS"	"2013-03-11:06:53:00"	"sss"	11
...	...	...	...

Amazon, p.227

2. If the additional attributes are not included, and are requested in the *Query* command, Dynamo will retrieve the values from the original table, but this will impact the efficiency of the secondary index.

Partitioning

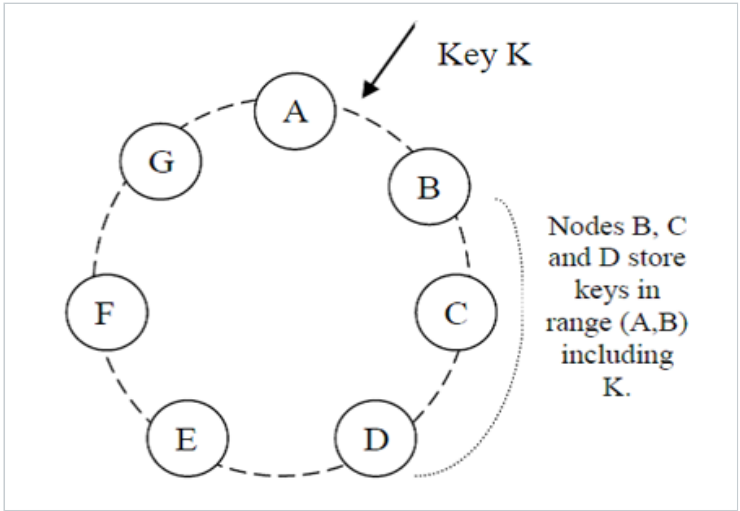
Dynamo is designed to be an "always writeable" data store, meaning that despite any network failures or concurrent update issues, the updates will complete successfully. According to the 2007 paper:

> Dynamo allows read and write operations to continue even during network partitions and resolves update conflicts using different conflict resolution mechanisms.

To accomplish this, rather than implementing a publisher/subscriber architecture, used by both the Google File System and Hadoop, Dynamo uses a peer-to-peer architecture. It assumes that all of the machines are "trusted" so it does not focus on security checking. The machines are all equal with each other.

Dynamo uses a hash function to calculate each node's position on the ring, which is conceptually like the ring below. Based on its hash value, each key is hashed to some position on the ring. It is assigned to the node greater than that position. In the example below, Key K hashes to a value greater than the position of Node A, so it is assigned to Node B, which is called its "coordinator node." The coordinator node is responsible for replicating its data items to the N-1 successor nodes, in this case C and D.

Each node is responsible for the keys that are mapped between itself and its Nth predecessor. So, for instance, node B is responsible for all keys that hash to positions on the ring between A and B. Assuming that N is 3, node B is also responsible for keys in the range (G, A) and (F, G). Node C is responsible for all keys that hash to positions between B and C, A and B, G and A, etc.



Dynamo: amazon's highly available key-value store, 2007

Consistency

DynamoDB is "eventually consistent" which means that a read may not show all the latest updates. The data is replicated across different machines, so one copy of the data may have received and completed the updates, but the updates may not have reached all copies of the data. If the client reads from a copy that has not yet been updated, a "stale" copy, that client may not see the most current updates. Eventually all the copies will receive the updates, and the data will once again be consistent. Amazon’s literature states that consistency of all copies is reached in a second or less.

One of the unique features of DynamoDB is that it allows the designer to select whether or not the reads should be eventually consistent or strongly consistent. The default choice is eventually consistent. When the table is created or updated, the throughput parameters are set. Throughput is defined in terms of "read capacity units" and "write capacity units." Write capacity units are simply the number of 1 kilobyte writes per second.

Strongly consistent reads require more effort and twice as many database resources as eventual consistency reads. So 10 “read capacity units” with eventual consistency would result in a throughput of 20 eventually consistent reads, or 10 strongly consistent reads. When each read costs money, the design is crucial.

If the strongly consistent option is chosen, the read will be executed on the most current data. In other words, the read will not happen until all the outstanding writes have updated all the copies of the data.

Specifying any secondary indexes also impacts the total throughput, since Dynamo has to maintain the indexes as part of any write transaction.

Versions and Vector Clocks

Each write operation creates a new version of the data. Because of this, several versions of the data can be present at any one time. Usually the newest version is assumed to be the best, and the older ones are discarded. To determine which version is older, Dynamo uses a system of "vector clocks."

A "vector clock" consists of a node/counter pair. Each version of the data has its own unique vector clock. The read operation returns the vector clock information along with the object to be located. When a client is updating an object, it passes the vector clock information that it obtained from the read operation, the context, to the write command. If there are multiple vector clocks indicating that the data was updated at different times, Dynamo has different ways to reconcile them.

Query and Scan

Amazon offers two commands to locate the data, *Query* and *Scan*.

A Query retrieves the data by the primary key or range key, so is optimized for read performance. The *Query* operation will always return results, although the result set could be empty if there are no results.

The Query will stop if:

- There are no more results
- The number of items returned satisfies the limits of the query.
- The amount of data exceeds Amazon’s maximum limit of 1 MB for a single query.

A *Scan* will scan the entire table, much like a table scan in an RDBMS, and it is generally just as slow. Filters can also be applied to Dynamo's Scan, but it will have to read the entire table and then filter the results before the result set is returned. Since the Query function queries directly on the key or secondary indexes, there is no need to filter out any of the results.

References

- (2007). Dynamo: Amazon's highly available key-value store.
- Vogels (2013). Expanding the Cloud: Faster, More Flexible Queries with DynamoDB.
- For more information on vector clocks, read "Dynamo: amazon's highly available key-value store", 2007.

Graph Databases

A graph is a structure composed of vertices, or nodes, and edges, or links that connect the nodes.

The MBTA subway map is an example of a graph. Here, the nodes are the subway stops, and the edges or links are the routes or paths from one subway stop to the next.

http://www.mbta.com/schedules_and_maps/subway

The edges describe a relationship between different nodes. On this map, there is an edge, or path, between Alewife and Braintree, so the rider knows that he can get to Park Street Station if he starts at Alewife and travels the Red Line until he reaches Park Street. If he wants to get to the Airport, and he is at Park Street Station, he will first have to travel or traverse the Green Line to Government Center, and then travel the Blue Line to get to the Airport.

On the subway map, Alewife is related to Park Street because there is a path on the Red Line that connects Alewife to Park Street. But at Park Street subway lines cross, and there are multiple paths. From there the traveler can either go down the Green Line or the Red Line. From the Park Street station on the Red Line, it's possible to travel in different directions, either to Alewife or to Braintree.

Representing these relationships in a relational database is possible, but because of the recursive nature of the links, it becomes complex, and is difficult to scale. More connections translate to poorer performance in an RDBMS query.

A graph database implements this kind of data model, using mathematical graph theory to traverse the paths.

Interestingly enough, some products, like Twitter's FlockDB, use a relational DBMS product as a backend, usually MySQL. Others, like Neo4j, use "native graph storage" which stores the nodes and relationships. Native graph storage is better for performance of traversals, but not as good for queries that don't involve traversing the links. A database backend is better for this kind of query, but not as good for performance of traversals. As for many tradeoffs, the best choice depends on the business needs.

Graph databases are appropriate when the data and the relationships between the data are naturally represented by a graph structure. They are usually used with OLTP applications. Examples include:

- Data representing the relationships between people or organizations. Graph databases are ideal for modeling a social web. Twitter created its own graph database product, called FlockDB, to track its users.
- Data representing links in a transportation or communications network.

A *property graph* includes not only edges and vertices, but also properties. Properties in a graph database generally consist of key/value pairs, and are used to describe nodes and vertices more completely. In our MBTA subway example the names of the stations and the distances between stations might be represented as properties of the vertices and edges respectively.

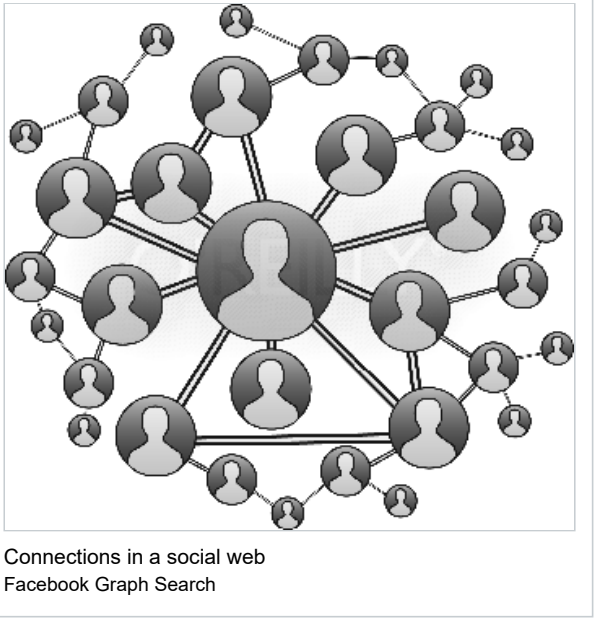
Most graph database implementations allow the Create, Read, Update, Delete (CRUD) operations. There is no “join” as such as there would be in a relational database, because traversing a link *is* essentially the join operation. Traversing a link is very fast since the connections already exist and has a constant time cost.

There are graph database query languages, such as Cypher, which is used by Neo4j, SPARQL and Gremlin. These query languages are used to execute the CRUD operations, and also for graph traversal.

Neo4j

Neo4j is a graph database product, built on Java JVMs, which is designed to model data that has many connections. A social web, for instance, is an ideal candidate for a graph database. Given the number of connections, and recursive relationships it becomes difficult to model this in an RDBMS. Although possible, the resulting number of joins would degrade performance as the database grows. A graph database can model this structure more efficiently.

A graph is made up of a set of vertices and a set of edges. The Neo4j structure is based on a *property graph model*, consisting of *nodes*, which are the vertices, and any number of *relationships*, which are the edges, or paths, between the nodes. There are no tables or rows, but one graph structure.



Connections in a social web
Facebook Graph Search

In the connections graph above, each node represents a person. Each of those people have relationships with one or more other people. Each of the people that those people are connected to can also have relationships with even more people and with each other.
Nodes and relationships can contain *properties* in Neo4j.*Properties* are sets of key-value pairs that describe the node or relationship. For example, in the social graph above, a node can have a property of “person:Professor Schudy”.Any node or relationship can have any number of properties.

Relationships connect the nodes and can also have properties of their own. A property of the relationships in a social graph might be something like “interested in” or “drives.” Relationships are named and have a starting and ending point and a direction.

A Neo4j property graph is a multigraph, which means multiple relationships can exist simultaneously between the same nodes. Different relationships can denote different kinds of relationships, which are identified by properties, like “Interested in” or “likes.” In a social graph, for instance, there might be diverse kinds of relationships that describe connections between people, like business, friendship, hobbies, etc.

Neo4j.com has a “sandbox” on its website that will allow you to enter commands and to see the results immediately on their sample database at this link: http://www.neo4j.org/learn/try.

Cypher’s CREATE command will create a node in a graph database. First the name of the node (COURSE) is specified followed by a property, like the actual course name (CS779).

```
CREATE  (n:Course { name: "CS779"});
```

Now we can create a node for the professor for this course:

```
CREATE (n:Professor { name:"Robert Schudy" });
```

A Cypher MATCH command is similar to a SQL SELECT command. It allows us to find a node and the relationships that exist for that node.

A *MATCH* command matches the node information with the information we’re looking for. In this case, it will find the node we just created for the course name of “CS779”:

```
MATCH (course:Course)
WHERE course.name = 'CS779'
RETURN course
```

We can add a property for course description, and check our changes. We can do this with another MATCH command and a SET clause, similar to a SQL UPDATE command.

```
MATCH (course:Course)
WHERE course.name = 'CS779'
SET course.description = 'Advanced Database Management'
RETURN course.name, course.description
```

This table is returned:

course.name	course.description
CS779	Advanced Database Management

(Pointer, 2010)

To make this more interesting, we’ll add some more courses:

```
CREATE (n:Course { name: "CS669"});
CREATE (n:Course { name: "CS625"});
CREATE (n:Course { name: "CS783"});
CREATE (n:Professor { name:"Robert Schudy" });
CREATE (n:Professor { name:"Warren Mansur" });
CREATE (n:Professor { name:"George Maiewski" });
```

And verify our adds with a simple count of all the nodes:

```
MATCH n
RETURN n
```

Neo4j returns this collection:

n
(8:Course {name:"CS779"})
(9:Course {name:"CS669"})
(10:Course {name:"CS625"})
(11:Course {name:"CS783"})
(12:Professor {name:"Robert Schudy"})
(13:Professor {name:"Warren Mansur"})
(14:Professor {name:"George Maiewski"})

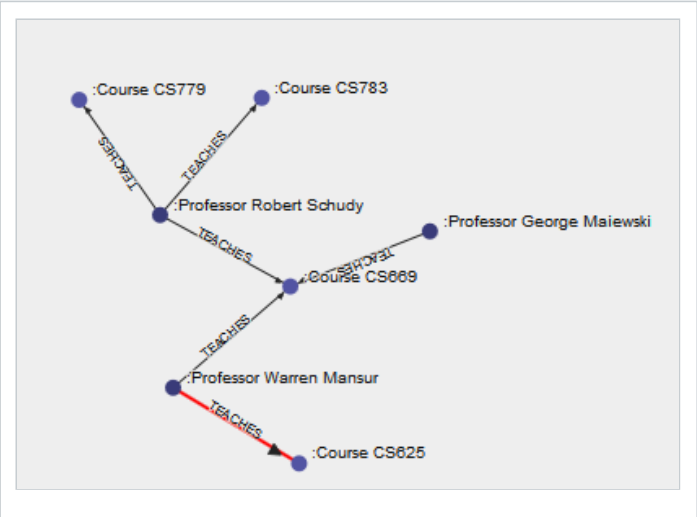
The heart of graph database technologies lies in the relationships it can set up between nodes. So to continue this example, we'll set up a "teaches" relationship between Professor and course.

We'll use the MATCH command to create the relationship, and this time we'll use a UNIQUE clause, which prohibits a duplicate relationship between "CS779" and Professor Schudy.

```
MATCH course:Course, professor:Professor
WHERE course.name = 'CS779' AND professor.name = 'Robert Schudy'
CREATE UNIQUE (course)-[r:TAUGHT_BY]->(professor)
RETURN r

And add more relationships between the other courses and professors.
MATCH (course:Course)
WHERE course.name = 'CS669'
SET course.description = 'Database Design and Implementation'
RETURN course.name, course.description

MATCH (course:Course)
WHERE course.name = 'CS625'
SET course.description = 'Business Data Communication and Networks'
RETURN course.name, course.description

MATCH (course:Course)
WHERE course.name = 'CS783'
SET course.description = 'Enterprise Architecture'
RETURN course.name, course.description

MATCH course:Course, professor:Professor
WHERE course.name = 'CS779' AND professor.name = 'Robert Schudy'
CREATE UNIQUE (professor)-[r:TEACHES]->(course)
RETURN r

MATCH course:Course, professor:Professor
WHERE course.name = 'CS669' AND professor.name = 'Robert Schudy'
CREATE UNIQUE (professor)-[r:TEACHES]->(course)
RETURN r

MATCH course:Course, professor:Professor
WHERE course.name = 'CS669' AND professor.name = 'Warren Mansur'
CREATE UNIQUE (professor)-[r:TEACHES]->(course)
RETURN r

MATCH course:Course, professor:Professor
WHERE course.name = 'CS783' AND professor.name = 'Robert Schudy'
CREATE UNIQUE (professor)-[r:TEACHES]->(course)
RETURN r
```

```
MATCH course:Course, professor:Professor
WHERE course.name = 'CS669' AND professor.name = 'George Maiewski'
CREATE UNIQUE (professor)-[r:TEACHES]->(course)
RETURN r

MATCH course:Course, professor:Professor
WHERE course.name = 'CS625' AND professor.name = 'Warren Mansur'
CREATE UNIQUE (professor)-[r:TEACHES]->(course)
RETURN r
```

Visually, our nodes and relationships will look like this:



This graph was created on the http://www.neo4j.org/learn/try site.

To locate a particular node, instead of using SELECTs and JOINs, Neo4j uses a system of “graph traversal”. A graph traversal starts at a node and follows the paths to get to another node or series of nodes. Traversing a node has a constant time cost, because the traversal looks at one node at a time, rather than looking at the entire data set.

To see all the courses taught by Professor Schudy, we’ll use a MATCH command that follows the course-professor relationship.

```
MATCH (course)--(professor)
WHERE professor.name='Robert Schudy'
RETURN course.description
```

Which would result in this table:

course.description
Advanced Database Management
Database Design and Implementation
Enterprise Architecture

And we could use the START command to start at a given node and follow the relationships for that node.

This simple example will give you some insight into the basic capabilities of Neo4j and other graph databases. Graph databases also support many more powerful features, which you can discover by consulting the material in the references at the end of this lecture.

Conclusion

There are many different kinds of big data and many different data models. There are also many different uses for big data. Consequently there are many different types of databases that can be used successfully in big data applications, including traditional RDBMS. Developers

should choose the data model and database based on business needs the strengths of each model and database. If strict consistency is required, a traditional RDBMS may be your best choice. If scalable performance is vital, and the demands on the database are largely reads, a key-value database with eventual consistency might work out best. If you need a data model that supports multiple relationships and recursive relationships, consider a graph database.

Bibliography

Abadi, D. J., Madden, S. R., & Hachem, N. (2008). Column-Stores vs. Row-Stores: How Different Are They Really? *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (pp. 967-980). Vancouver: SIGMOD '08.

Abadi, D. J., Myers, D. S., DeWitt, D. J., & Madden, S. R. (2007). Materialization Strategies in a Column-Oriented DBMS. *Proceedings of ICDE 2007*, (pp. 466-475). Istanbul.

Amazon. (n.d.). *Amazon Dynamo Developer's guide*. Retrieved from www.aws.com: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

Apache Hadoop. *MapReduce Tutorial*. 02 19, 2010. http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html (accessed 10 20, 2012; no longer available)

apache.org. (n.d.). *Chapter 9: Architecture.* Retrieved from hbase.apache.org: http://hbase.apache.org/book/architecture.html#arch.overview

apache.org. (n.d.). *Zookeeper Overview.* Retrieved from zookeeper.apache.org: http://zookeeper.apache.org/doc/trunk/zookeeperOver.html

Awadallah, A. (n.d.). *Introducing Hadoop: The Modern Data Operating System.* Retrieved 09 20, 2012, from Stanford University: http://www.stanford.edu/class/ee380/Abstracts/111116-slides.pdf

Brewer, E. (2000). Towards More Robust Distributed Systems. *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing* (p. 7). New York: Association for Computing Machinery (ACM).

Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. *Proceedings of the OSDI*, (pp. 335-350).

Capriolo, E., Wampler, D., & Rutherglen, J. (2012). *Programming Hive.* Sebastopol, CA: O'Reilly Media, Inc.

Cattell, R. (2010). "Scalable SQL and NoSQL Data Stores". *SIGMOD*, 12-27.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., & Wallach, D. A. (2006). Bigtable: A Distributed Storage System for Structured Data. *Proceedings of the OSDI*, (pp. 205-218).

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., & Wallach, D. (2006). Bigtable: A Distributed Storage System for Structured Data. *Proceedings of the OSDI*, (pp. 205-218).

Copeland, G. P. (1985). A Decomposition Storage Model. *International Conference on the Management of Data* (pp. 268-279). New York: ACM.

Dean, Jeffrey, and Sanjay Ghemawat. *"MapReduce: Simplified Data Processing on Large Clusters"* OSDI '04: 6th Symposium on Operating Systems Design and Implementation. USENIX, Inc., 2004. 137-149.

Dean, Jeffrey, and Sanjay Ghemawat. "*MapReduce: A Flexible Data Processing Tool*." Communications of the ACM, 2010: 72-77.

Dynamo: amazon's highly available key-value store. (2007). *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 41*(6).

Facebook Graph Search. (n.d.). Retrieved from bounce ideas blog: http://blog.zadrocommunications.com.au/facebook-graph-search-will-it-revolutionise-your-online-experience (no longer available)

George, L. (2011, 11 24). *HBase vs. BigTable Comparison.* Retrieved from Lineland: http://www.larsgeorge.com/2009/11/hbase-vs-bigtable-comparison.html

George, L. (2011). *HBase: The Definitive Guide.* Sebastopol: O'Reilly Media.

Gilbert, S., & Lynch, N. A. (2012). Perspectives on the CAP Theorem. *Computer*, 30-36.

Google. (2011, 07). *LevelDB Benchmarks*. Retrieved from Googlecode.com: http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html

Gray, J., Helland, P., O'Neil, P., & Shasha, D. (n.d.). The Dangers of Replication and a Solution. Association for Computing Machinery, Inc.

Harizopoulos, S. A. (2009). *Column-Oriented Database Systems*. Retrieved from Yale University: http://www.cs.yale.edu/homes/dna/talks/Column_Store_Tutorial_VLDB09.pdf

Harizopoulos, S., Liang, V., Abadi, D. J., & Madden, S. (2006). Performance Tradeoffs in Read-Optimized Databases. *VLDB '06 Proceedings of the 32nd international conference on Very large data bases*, (pp. 487-498).

Harris, R. (2006, 09 06). *Google's Bigtable Distributed Storage System.* Retrieved from StorageMojo: http://storagemojo.com/2006/09/07/googles-bigtable-distributed-storage-system-pt-i/

Holmes, A. (2012). *Hadoop in Practice.* Shelter Island: Manning Publications.

"Joins with plain Map Reduce or MultipleInputs." Kick Start Hadoop. 09 19, 2011. http://kickstarthadoop.blogspot.com/2011/09/joins-with-plain-map-reduce.html (accessed 03 20, 2013).

Lam, C. (2011). *Hadoop in Action.* Stamford, CT: Manning Publications.

Lamb, A., Fuller, M., Varadarajan, R., Tran, n., Vandiver, B., Doshi, L., et al. (2012). The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, (pp. 1790-1801). Istanbul.

Matei, G. (2010). Column-Oriented Databases, an Alternative for Analy. *Database Systems Journal*, 3-16.

MongoDB. (2013). *Mongo DB 2.4 manual.* Retrieved from mongodb.org: http://docs.mongodb.org/manual/

Murthy, A. (n.d.). *Apache Hadoop Yarn Background and an Overview*. Retrieved 12 3, 2012, from http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/

Neubauer, P. (2010, 05 12). *Graph Databases, NoSQL and Neo4j*. Retrieved from InfoQ: http://www.infoq.com/articles/graph-nosql-neo4j

Neubauer, P., & Rodriguez, M. A. (2010). Graph Transversal Pattern. *CoRR, abs/1004.1001*.

Olesker, Alex. Hadoop is an Open Source Revolution: Federal Computer Week Interview. 11 24, 2011. http://ctovision.com/2011/11/hadoop-is-an-open-source-revolution-federal-computer-week-interview/ (accessed 11 21, 2012).

Pointer, R. (2010, 05 03). *Introducing FlockDB*. Retrieved from blog.twitter.com: https://blog.twitter.com/2010/introducing-flockdb

Pritchett, D. (2008, May/June). BASE: An ACID alternative. *Queue - Object-Relational Mapping*, pp. 48-55.

Raab, D. (2008, 10 01). *How to Judge a Columnar Database, Revisited*. Retrieved from David Raab Article Archive: http://archive.raabassociatesinc.com/2008/10/how-to-judge-a-columnar-database-revisited/

Rahien, A. (2010, 03 29). *That NoSql Thing - Key/Value Stores*. Retrieved from Ayende@Rahien: http://ayende.com/blog/4449/that-no-sql-thing-key-value-stores

Robinson, I., Webber, J., & Eifrem, E. (n.d.). *Graph Databases.* O'Reilly Media.

Rodriguez, M. A., & Neubauer, P. (2010). "The Graph Transversal Pattern". *Graph Data Management: Techniques and Applications*.

Schedules and Maps. (2013). Retrieved from mbta.com: http://www.mbta.com/schedules_and_maps/subway/

Shvachko, K., Kuang, H., & Radia, S. (2010). The Hadoop Distributed File System. *MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 1-10.

Thomas, L., & Gupta, A. (2012, 11 30). *HBase vs. Cassandra.* Retrieved 04 08, 2013, from BigDataNoob: http://bigdatanoob.blogspot.com/2012/11/hbase-vs-cassandra.html

van Groningen, Martin. "*Introduction to Hadoop*." Trifork. 08 04, 2009. http://blog.jteam.nl/2009/08/04/introduction-to-hadoop/ (accessed 03 19, 2013).

Vogels, W. (2009, January). Eventually Consistent. *Communications of the ACM*, pp. 40-44.

Vogels, W. (2013, 04 13). *Expanding the Cloud: Faster, More Flexible Queries with DynamoDB*. Retrieved from allthingsdistributed.com: http://www.allthingsdistributed.com/2013/04/dynamdb-local-secondary-indices.html

White, T. (2012). *Hadoop: The Definitive Guide.* Sebastopol: O'Reilly Media.