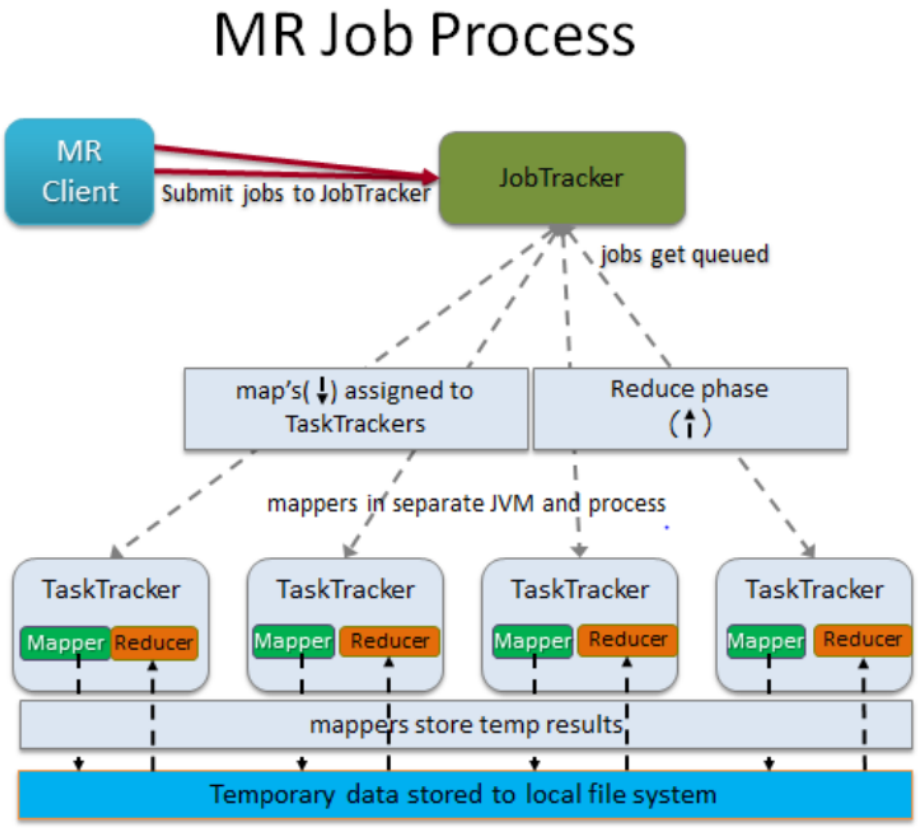
MapReduce

MapReduce is an algorithm design pattern that originated in the functional programming world. It consists of three steps. First, you write a mapper function or script that goes through your input data and outputs a series of keys and values to use calculating the results. The keys are used to cluster together bits of data that will be needed to calculate a single output result. The unordered list of keys and values is then put through a sort step that ensures that all the fragments that have the same key are next to one another in the file. The reducer stage then goes through the sorted output and receives all of the values that have the same key in a contiguous block.

That may sound like a very roundabout way of building your algorithms, but it's prime virtue is that it removes unplanned random accesses, with all scattering and gathering handled in the sorting phase. Even on single machines, this boosts performance, thanks to the increased locality of memory accesses, but it also allows the process to be split across a large number of machines easily, by dealing with the input in many independent chunks and partitioning the data based on the key.

Hadoop is the best-known public system for running Map Reduce algorithms, but many modern databases, such as MongoDB, also support it as an option. It's worthwhile even in a fairly traditional system, since if you can write your query in a MapReduce form, you'll be able to run it efficiently on as many machines as you have available.



NoSQL Databases

A few years ago, web programmers started to use the memory cached system to temporarily store data in RAM, so frequently used values could be retrieved very quickly, rather than relying on a slower path accessing the full database from disk.

This coding pattern required all of the data accesses to be written using only key/value primitives, initially in addition to the traditional SQL queries on the main database. As developers got more comfortable with the approach, they started to experiment with databases that used a key/value interface for the persistent storage as well as the cache, since they already had to express most of their queries in that form anyway. This is a rare example of the removal of an abstraction layer, since the key/value interface is less expressive and lower-level than a query language like SQL.

These systems do require more work from an application developer, but they also offer a lot more flexibility and control over the work the database is performing. The cut-down interface also makes it easier for database developers to create new and experimental systems to try out new solutions to tough requirements like very large-scale, widely distributed data sets or high through put applications. This widespread

demand for solutions, and the comparative ease of developing new systems, has led to a flowering of new databases.

The main thing they have in common is that none of them support the traditional SQL interface, which has led to the movement being dubbed NoSQL. It's a bit misleading, though, since almost every production environment that they're used in also has an SQL-based database for anything that requires flexible queries and reliable transactions, and as the products mature, it's likely that some of them will start supporting the language as an option.

If "NoSQL" seems too combative, think of it as "NotOnlySQL." These are all tools designed to trade the reliability and ease-of-use of traditional databases for the flexibility and performance required by new problems developers are encountering. With so many different systems appearing, such a variety of design tradeoffs, and such a short track record for most, this list is inevitably incomplete and somewhat subjective.

MongoDB

Mongo, whose name comes from "humongous", is a database aimed at developers with fairly large data sets, but who want something that's low maintenance and easy to work with. It's a document-oriented system, with records that look similar to JSON objects with the ability to store and query on nested attributes. From my own experience, a big advantage is the proactive support from the developers employed by 10gen, the commercial company that originated and supports the open source project.

It always had quick and helpful responses both on the IRC channel and mailing list , something that's crucial when you're dealing with comparatively young technologies like these. It supports automatic sharding and MapReduce operations. Queries are written in JavaScript, with an interactive shell available, and bindings for all of the other popular languages.

CouchDB

CouchDB is similar in many ways to MongoDB, as a document-oriented database with a JavaScript interface, but it differs in how it supports querying, scaling, and versioning.

It uses a multiversion concurrency control approach, which helps with problems that require access to the state of data at various times, but it does involve more work on

the client side to handle clashes on writes, and periodic garbage collection cycles have to be run to remove old data. It doesn't have a good built-in method for horizontal scalability, but there are various external solutions like
BigCouch, Lounge, and Pillow to handle splitting data and processing across a cluster of machines.

You query the data by writing JavaScript MapReduce functions called views, an approach that makes it easy for the system to do the processing in a distributed way. Views offer a lot of power and flexibility, but they can be a bit overwhelming for simple queries.

Cassandra

Originally an internal Facebook project, Cassandra was open sourced a few years ago and has become the standard distributed database for situations where it's worth investing the time to learn a complex system in return for a lot of power and flexibility. Traditionally, it was a long struggle just to set up a working cluster, but as the project matures, that has become a lot easier.

It's a distributed key/value system, with highly structured values that are held in a hierarchy similar to the classic database/table levels, with the equivalents being keyspaces and column families. It's very close to the data model used by Google's BigTable. By default, the data is sharded and balanced automatically using consistent hashing on key ranges, though other schemes can be configured. The data structures are optimized for consistent write performance, at the cost of occasionally slow read operations. One very useful feature is the ability to specify how many nodes must agree before a read or write operation completes. Setting the consistency level allows you to tune the CAP tradeoffs for your particular application, to prioritize speed over consistency or vice versa.

The lowest-level interface to Cassandra is through Thrift, but there are friendlier clients available for most major languages. The recommended option for running queries is through Hadoop. You can install Hadoop directly on the same cluster to ensure locality of access, and there's also a distribution of Hadoop integrated with Cassandra available from DataStax.

There is a command-line interface that lets you perform basic administration tasks, but it's quite bare bones. It is recommended that you choose initial tokens when you first set up your cluster, but otherwise the decentralized architecture is fairly low maintenance, barring major problems.

Redis

Two features make Redis stand out: it keeps the entire database in RAM, and its values can be complex data structures. Though the entire dataset is kept in memory, it's also backed up on disk periodically, so you can use it as a persistent database. This approach does offer fast and predictable performance, but speed falls off a cliff if the size of your data expands beyond available memory and the operating system starts paging virtual memory to handle accesses.

This won't be a problem if you have small or predictably sized storage needs, but it does require a bit of forward planning as you're developing applications. You can deal with larger data sets by clustering multiple machines to- gether, but the sharding is currently handled at the client level. There is an experimental branch of the code under active development that supports clustering at the server level.

The support for complex data structures is impressive, with a large number of list and set operations handled quickly on the server side. It makes it easy to do things like appending to the end of a value that's a list, and then trim the list so that it only holds the most recent 100 items. These capabilities do make it easier to limit the growth of your data than it would be in most systems, as well as making life easier for application developers.

BigTable

BigTable is only available to developers outside Google as the foundation of the AppEngine data store. Despite that, as one of the pioneering alternative databases, it's worth looking at.

It has a more complex structure and interface than many NoSQL datastores, with a hierarchy and multidimensional access. The first level, much like traditional relational databases, is a table holding data. Each table is split into multiple rows, with each row addressed with a unique key string. The values inside the row are arranged into cells, with each cell identified by a column family identifier, a column name, and a timestamp, each of which will be explained below.

The row keys are stored in ascending order within file chunks called shards. This ensures that operations accessing continuous ranges of keys are efficient, though it does mean you have to think about the likely order you'll be reading your keys in. In one example, Google reversed the domain names of URLs they were using as keys so that all links from similar domains were nearby; for example, com.google.maps/index.html was near com.google.www/index.html

You can think of a column family as something like a type or a class in a programming language. Each represents a set of data values that all have some common properties; for example, one might hold the HTML content of web pages, while another might be designed to contain a language identifier string. There's only expected to be a small number of these families per table, and they should be altered infrequently, so in practice they're often chosen when the table is created. They can have properties, constraints, and behaviors associated with them.

Column names are confusingly not much like column names in a relational database. They are defined dynamically, rather than specified ahead of time, and they often hold actual data themselves. If a column family represented inbound links to a page, the column name might be the URL of the page that the link is from, with the cell contents holding the link's text.

The timestamp allows a given cell to have multiple versions over time, as well as making it possible to expire or garbage collect old data.
A given piece of data can be uniquely addressed by looking in a table for the full identifier that conceptually looks like row key, then column family, then column name, and finally timestamp. You can easily read all the values for a given row key in a particular column family, so you could actually think of the column family as being the closest comparison to a column in a relational database.

As you might expect from Google, BigTable is designed to handle very large data loads by running on big clusters of commodity hardware. It has per-row transaction guarantees, but it doesn't offer any way to atomically alter larger numbers of rows. It uses the Google File System as its underlying storage, which keeps redundant copies of all the persistent files so that failures can be recovered from.

HBase

HBase was designed as an open source clone of Google's BigTable, so unsurprisingly it has a very similar interface, and it relies on a clone of the Google File System called HDFS. It supports the same data structure of tables, row keys, column families, column names, timestamps, and cell values, though it is recommended that each table have no more than two or three families for performance reasons.

HBase is well integrated with the main Hadoop project, so it's easy to write and read to the database from a MapReduce job running on the system. One thing to watch out for is that the latency on individual reads and writes can be comparatively slow, since

it's a distributed system and the operations will involve some network traffic. HBase is at its best when it's accessed in a distributed fashion by many clients. If you're doing serialized reads and writes you may need to think about a caching strategy.

Hypertable

Hypertable is another open source clone of BigTable. It's written in C++, rather than Java like HBase, and has focused its energies on high performance. Otherwise, its interface follows in BigTable's footsteps, with the same column family and timestamping concepts.

Voldemort

An open source clone of Amazon's Dynamo database created by LinkedIn, Voldemort has a classic three-operation key/value interface, but with a sophisticated backend architecture to handle running on large distributed clusters. It uses consistent hashing to allow fast lookups of the storage locations for particular keys, and it has versioning control to handle inconsistent values.

A read operation may actually return multiple values for a given key if they were written by different clients at nearly the same time. This then puts the burden on the application to take some sensible recovery actions when it gets multiple values, based on its knowledge of the meaning of the data being written. The example that Amazon uses is a shopping cart, where the set of items could be unioned together, losing any deliberate deletions but retaining any added items, which obviously makes sense—from a revenue perspective, at least!

Riak

Like Voldemort, Riak was inspired by Amazon's Dynamo database, and it offers a key/value interface and is designed to run on large distributed clusters. It also uses consistent hashing and a gossip protocol to avoid the need for the kind of centralized index server that BigTable requires, along with versioning to handle update conflicts. Querying is handled using MapReduce functions written in either Erlang or JavaScript. It's open source under an Apache license, but there's also a closed source commercial version with some special features designed for enterprise customers.

ZooKeeper

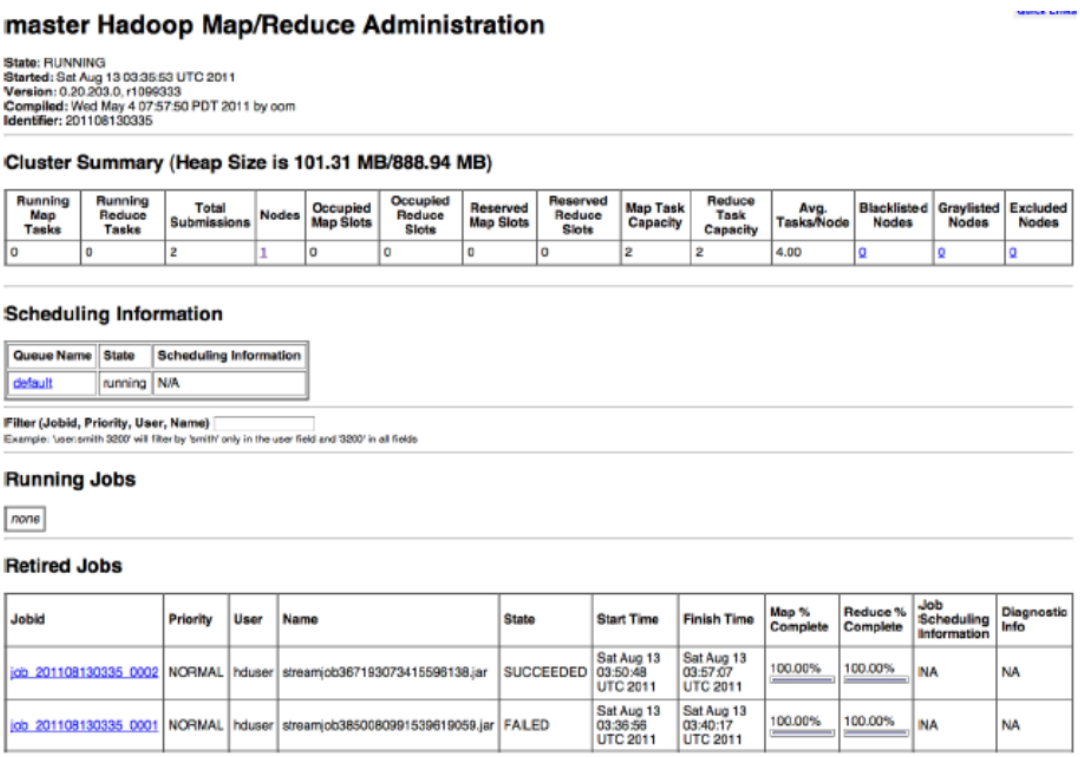
When you're running a service distributed across a large cluster of machines, even tasks like reading configuration information, which are simple on single-machine systems,can be hard to implement reliably. The ZooKeeper framework was originally built at Yahoo! to make it easy for the company's applications to access configuration information in a robust and easy-to-understand way, but it has since grown to offer a lot of features that help coordinate work across distributed clusters.

One way to think of it is as a very specialized key/value store, with an interface that looks a lot like a filesystem and supports operations like watching callbacks, write consensus, and transaction Ids that are often needed for coordinating distributed algorithms.

This has allowed it to act as a foundation layer for services like LinkedIn's Norbert, a flexible framework for managing clusters of machines. ZooKeeper itself is built to run in a distributed way across a number of machines, and it's designed to offer very fast reads, at the expense of writes that get slower the more servers are used to host the service.

MapReduce

In the traditional relational database world, all processing happens after the information has been loaded into the store, using a specialized query language on highly structured and optimized data structures. The approach pioneered by Google, and adopted by many other web companies, is to instead create a pipeline that reads and writes to arbitrary file formats, with intermediate results being passed between stages as files, with the computation spread across many machines. Typically based around the Map-Reduce approach to distributing work, this approach requires a whole new set of tools, which I'll describe below.



master Hadoop Map/Reduce Administration

State: RUNNING
Started: Sat Aug 13 03:35:53 UTC 2011
Version: 0.20.203.0, r1099333
Compiled: Wed May 4 07:57:50 PDT 2011 by oom
Identifier: 201108130335

Cluster Summary (Heap Size is 101.31 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Graylisted Nodes	Excluded Nodes
0	0	2	1	0	0	0	0	2	2	4.00	0	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Retired Jobs

Jobid	Priority	User	Name	State	Start Time	Finish Time	Map % Complete	Reduce % Complete	Job Scheduling Information	Diagnostic Info
job_201108130335_0002	NORMAL	hduser	streamjob3671930734155961 38.jar	SUCCEEDED	Sat Aug 13 03:50:48 UTC 2011	Sat Aug 13 03:57:07 UTC 2011	100.00%	100.00%	NA	NA
job_201108130335_0001	NORMAL	hduser	streamjob3850080991539619059.jar	FAILED	Sat Aug 13 03:36:56 UTC 2011	Sat Aug 13 03:40:17 UTC 2011	100.00%	100.00%	NA	NA

Hadoop

Originally developed by Yahoo! as a clone of Google's MapReduce infrastructure, but subsequently open sourced, Hadoop takes care of running your code across a cluster of machines. Its responsibilities include chunking up the input data, sending it to each machine, running your code on each chunk, checking that the code ran, passing any results either on to further processing stages or to the final output location, performing the sort that occurs between the map and reduce stages and sending each chunk of that sorted data to the right machine, and writing debugging information on each job's progress, among other things.

As you might guess from that list of requirements, it's quite a complex system, but thankfully it has been battle-tested by a lot of users. There's a lot going on under the hood, but most of the time, as a developer, you only have to supply the code and data, and it just works. Its popularity also means that there's a large ecosystem of related tools, some that making writing individual processing steps easier, and others that orchestrate more complex jobs that require many inputs and steps. As a novice user, the best place to get started is by learning to write a streaming job in your favourite scripting language, since that lets you ignore the gory details of what's going on behind the scenes.

As a mature project, one of Hadoop's biggest strengths is the collection of debugging

and reporting tools it has built in. Most of these are accessible through a web interface that holds details of all running and completed jobs and lets you drill down to the error and warning log files.

Hive

With Hive, you can program Hadoop jobs using SQL. It's a great interface for anyone coming from the relational database world, though the details of the underlying implementation aren't completely hidden. You do still have to worry about some differences in things like the most optimal way to specify joins for best performance and some missing language features. Hive does offer the ability to plug in custom code for situations that don't fit into SQL, as well as a lot of tools for handling input and output.

To use it, you set up structured tables that describe your input and output, issue load commands to ingest your files, and then write your queries as you would in any other relational database. Do be aware, though, that because of Hadoop's focus on large-scale processing, the latency may mean that even simple jobs take minutes to complete, so it's not a substitute for a real-time transactional database.

Pig

The Apache Pig project is a procedural data processing language designed for Hadoop. In contrast to Hive's approach of writing logic-driven queries, with Pig you specify a series of steps to perform on the data. It's closer to an everyday scripting language, but with a specialized set of functions that help with common data processing problems.

It's easy to break text up into component ngrams, for example, and then count up how often each occurs. Other frequently used operations, such as filters and joins, are also supported. Pig is typically used when your problem (or your inclination) fits with a procedural approach, but you need to do typical data processing operations, rather than general purpose calculations. Pig has been described as "the duct tape of Big Data" for its usefulness there, and it is often combined with custom streaming code written in a scripting language for more general operations.

Cascading

Most real-world Hadoop applications are built of a series of processing steps, and Cas-

cading lets you define that sort of complex workflow as a program. You lay out the logical flow of the data pipeline you need, rather than building it explicitly out of MapReduce steps feeding into one another. To use it, you call a Java API, connecting objects that represent the operations you want to perform into a graph.

The system takes that definition, does some checking and planning, and executes it on your Hadoop cluster. There are a lot of built-in objects for common operations like sorting, grouping, and joining, and you can write your own objects to run custom processing code.

Cascalog

Cascalog is a functional data processing interface written in Clojure. Influenced by the old Datalog language and built on top of the Cascading framework, it lets you write your processing code at a high level of abstraction while the system takes care of assembling it into a Hadoop job. It makes it easy to switch between local execution on small amounts of data to test your code and production jobs on your real Hadoop cluster.

Cascalog inherits the same approach of input and output taps and processing operations from Cascading, and the functional paradigm seems like a natural way of specifying data flows. It's a distant descendant of the original Clojure wrapper for Cascading, cascading clojure
.

mrjob

mrjob is a framework that lets you write the code for your data processing, and then transparently run it either locally, on Elastic MapReduce, or on your own Hadoop cluster. Written in Python, it doesn't offer the same level of abstraction or built-in operations as the Java-based Cascading.

The job specifications are defined as a series of map and reduce steps, each implemented as a Python function. It is great as a framework for executing jobs, even allowing you to attach a debugger to local runs to really understand what's happening in your code.

Caffeine

Even though no significant technical information has been published on it, I'm including Google's Caffeine project, as there's a lot of speculation that it's a replacement for the MapReduce paradigm. From reports and company comments, it appears that Google is using a new version of the Google File System that supports smaller files and distributed masters.

It also sounds like the company has moved away from the batch processing approach to building its search index, instead using a dynamic database approach to make updating faster. There's no indication that Google's come up with a new algorithmic approach that's as widely applicable as MapReduce, though I am looking forward to hearing more about the new architecture.

S4

Yahoo! initially created the S4 system to make decisions about choosing and positioning ads, but the company open sourced it after finding it useful for processing arbitrary streams of events. S4 lets you write code to handle unbounded streams of events, and runs it distributed across a cluster of machines, using the ZooKeeper framework tobhandle the housekeeping details.

You write data sources and handlers in Java, and S4 handles broadcasting the data as events across the system, load-balancing the work across the available machines. It's focused on returning results fast, with low latency, for applications like building near real-time search engines on rapidly changing content.

This sets it apart from Hadoop and the general MapReduce approach, which involves Synchronization steps within the pipeline, and so some degree of latency. One thing to be aware of is that S4 uses UDP and generally offers no delivery guarantees for the data that's passing through the pipeline. It usually seems possible to adjust queue sizes to avoid data loss, but it does put the burden of tuning to reach the required level of reliability on the application developer.

MapR

MapR is a commercial distribution of Hadoop aimed at enterprises. It includes its own file systems that are a replacement for HDFS, along with other tweaks to the framework, like distributed name nodes for improved reliability. The new file system aims to offer increased performance, as well as easier backups and compatibility with NFS to make it simpler to transfer data in and out. The programming model is still the standard Hadoop one; the focus is on improving the infrastructure surrounding the core framework to make it more appealing to corporate customers.

Acunu

Like MapR, Acunu is a new low-level data storage layer that replaces the traditional file system, though its initial target is Cassandra rather than Hadoop. By writing a kernel-level key/value store called Castle, which has been open-sourced, the creators are able to offer impressive speed boosts in many cases.

The data structures behind the performance gains are also impressive. Acunu also offers some of the traditional benefits of a commercially supported distribution, such as automatic configuration and other administration tools.

Flume

One very common use of Hadoop is taking web server or other logs from a large number of machines, and periodically processing them to pull out analytics information.

The Flume project is designed to make the data gathering process easy and scalable, by running agents on the source machines that pass the data updates to collectors, which then aggregate them into large chunks that can be efficiently written as HDFS files. It's usually set up using a command-line tool that supports common operations, like tailing a file or listening on a network socket, and has tunable reliability guarantees that let you trade off performance and the potential for data loss.

Kafka

Kafka is a comparatively new project for sending large numbers of events from producers to consumers. Originally built to connect LinkedIn's website with its backend systems, it's somewhere between S4 and Flume in its functionality. Unlike S4, it's persistent and offers more safeguards for delivery than Yahoo!'s UDP-based system, but it tries to retain its distributed nature and low latency. It can be used in a very similar way to Flume, keeping its high throughput, but with a more flexible system for creating multiple clients and an underlying architecture that's more focused on parallelization. Kafka relies on ZooKeeper to keep track of its distributed processing.

Azkaban

The trickiest part of building a working system using these new data tools is the integration. The individual services need to be tied together into sequences of operations that are triggered by your business logic, and building that plumbing is surprisingly time consuming. Azkaban is an open source project from LinkedIn that lets you define what you want to happen as a job flow, possibly made up of many dependent steps, and then handles a lot of the messy housekeeping details.

It keeps track of the log outputs, spots errors and emails about errors as they happen, and provides a friendly web interface so you can see how your jobs are getting on. Jobs are created as text files, using a very minimal set of commands, with any complexity expected to reside in the Unix commands or Java programs that the step calls.

Oozie

Oozie is a job control system that's similar to Azkaban, but exclusively focused on Hadoop. This isn't as big a difference as you might think, since most Azkaban uses I've run across have also been for Hadoop, but it does mean that Oozie's integration is a bit tighter, especially if you're using the Yahoo! distribution of both. Oozie also supports a more complex language for describing job flows, allowing you to make runtime decisions about exactly which steps to perform, all described in XML files.

There's also an API that you can use to build your own extensions to the system's functionality. Compared to Azkaban, Oozie's interface is more powerful but also more complex, so which you choose should depend on how much you need Oozie's advanced features.

Greenplum

Though not strictly a NoSQL database, the Greenplum system offers an interesting way of combining a flexible query language with distributed performance. Built on top of the Postgres open source database, it adds in a distributed architecture to run on a cluster of multiple machines, while retaining the standard SQL interface.

It automatically shards rows across machines, by default based on a hash of a table's primary key, and works to avoid data loss both by using RAID drive setups on individual servers and by replicating data across machines. It's normally deployed on clusters of machines with comparatively fast processors and large amounts of RAM, in contrast to the pattern of using commodity hardware that's more common in the web world.

IMPALA

How Impala Fits Into the Hadoop Ecosystem

Impala makes use of many familiar components within the Hadoop ecosystem. Impala can interchange data with other Hadoop components, as both a consumer and a producer, so it can fit in flexible ways into your ETL and ELT pipelines.

Continue reading:

- How Impala Works with Hive
- Overview of Impala Metadata and the Metastore
- How Impala Uses HDFS
- How Impala Uses HBase

How Impala Works with Hive

A major Impala goal is to make SQL-on-Hadoop operations fast and efficient enough to appeal to new categories of users and open up Hadoop to new types of use cases. Where practical, it makes use of existing Apache Hive infrastructure that many Hadoop users already have in place to perform long-running, batch-oriented SQL queries.

In particular, Impala keeps its table definitions in a traditional MySQL or PostgreSQL database known as the **metastore**, the same database where Hive keeps this type of data. Thus, Impala can access tables defined or loaded by Hive, as long as all columns use Impala-supported data types, file formats, and compression codecs.

The initial focus on query features and performance means that Impala can read more types of data with the SELECT statement than it can write with the INSERT statement. To query data using the Avro, RCFile, or SequenceFile file formats, you load the data using Hive.

The Impala query optimizer can also make use of table statistics and column statistics. Originally, you gathered this information with the ANALYZE TABLE statement in Hive; in Impala 1.2.2 and higher, use the Impala COMPUTE STATS statement instead. COMPUTE STATS requires less setup, is more reliable, and does not require switching back and forth between impala-shell and the Hive shell.

Overview of Impala Metadata and the Metastore

As discussed in How Impala Works with Hive, Impala maintains information about table definitions in a central database known as the **metastore**. Impala also tracks other metadata for the low-level characteristics of data files:

- The physical locations of blocks within HDFS.

For tables with a large volume of data and/or many partitions, retrieving all the metadata for a table can be time-consuming, taking minutes in some cases. Thus, each Impala node caches all of this metadata to reuse for future queries against the same table.

If the table definition or the data in the table is updated, all other Impala daemons in the cluster must receive the latest metadata, replacing the obsolete cached metadata, before issuing a query against that table. In Impala 1.2 and higher, the metadata update is automatic, coordinated through the catalogd daemon, for all DDL and DML statements issued through Impala.

For DDL and DML issued through Hive, or changes made manually to files in HDFS, you still use the REFRESH statement (when new data files are added to existing tables) or the INVALIDATE METADATA statement (for entirely new tables, or after dropping a table, performing an HDFS rebalance operation, or deleting data files). Issuing INVALIDATE METADATA by itself retrieves metadata for all the tables tracked by the metastore. If you know that only specific tables have been changed outside of Impala, you can issue REFRESH *table_name* for each affected table to only retrieve the latest metadata for those tables.

How Impala Uses HDFS

Impala uses the distributed filesystem HDFS as its primary data storage medium. Impala relies on the redundancy provided by HDFS to guard against hardware or network outages on individual nodes. Impala table data is physically represented as data files in HDFS, using familiar HDFS file formats and compression codecs. When data files are present in the directory for a new table, Impala reads them all, regardless of file name. New data is added in files with names controlled by Impala.

How Impala Uses HBase

HBase is an alternative to HDFS as a storage medium for Impala data. It is a database storage system built on top of HDFS, without built-in SQL support. Many Hadoop users already have it configured and store large (often sparse) data sets in it. By defining tables in Impala and mapping them to equivalent tables in HBase, you can query the contents of the HBase tables through Impala, and even perform join queries including both Impala and HBase tables.

Twitter Analysis

Social media has gained immense popularity with marketing teams, and Twitter is an effective tool for a company to get people excited about its products. Twitter makes it easy to engage users and communicate directly with them, and in turn, users can provide word-of-mouth marketing for companies by discussing the products. Given limited resources, and knowing we may not be able to talk to everyone we want to target directly, marketing departments can be more efficient by being selective about whom we reach out to.

In this post, we'll learn how we can use Apache Flume, Apache HDFS, Apache Oozie, and Apache Hive to design an end-to-end data pipeline that will enable us to analyze Twitter data. This will be the first post in a series. The posts to follow to will describe, in more depth, how each component is involved and how the custom code operates. All the code and instructions necessary to reproduce this pipeline is available on the Cloudera Github.

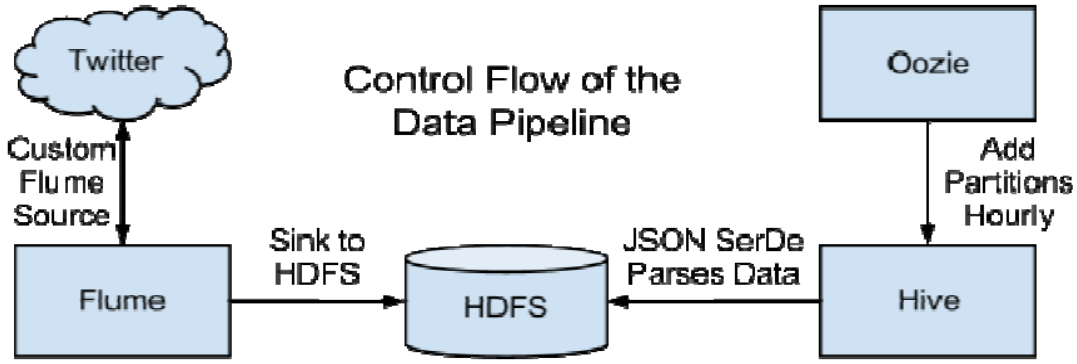
Who is Influential?

To understand whom we should target, let's take a step back and try to understand the mechanics of Twitter. A user – let's call him Joe – follows a set of people, and has a set of followers. When Joe sends an update out, that update is seen by all of his followers. Joe can also retweet other users' updates. A retweet is a repost of an update, much like you might forward an email. If Joe sees a tweet from Sue, and retweets it, all of Joe's followers see Sue's tweet, even if they don't follow Sue. Through retweets, messages can get passed much further than just the followers of the person who sent the original tweet. Knowing that, we can try to engage users whose updates tend to generate lots of retweets. Since Twitter tracks retweet counts for all tweets, we can find the users we're looking for by analyzing Twitter data.

Now we know the question we want to ask: Which Twitter users get the most retweets? Who is influential within our industry?

How Do We Answer These Questions?

SQL queries can be used to answer this question: We want to look at which users are responsible for the most retweets, in descending order of most retweeted. However, querying Twitter data in a traditional RDBMS is inconvenient, since the Twitter Streaming API outputs tweets in a JSON format which can be arbitrarily complex. In the Hadoop ecosystem, the Hive project provides a query interface which can be used to query data that resides in HDFS. The query language looks very similar to SQL, but allows us to easily model complex types, so we can easily query the type of data we have. Seems like a good place to start. So how do we get Twitter data into Hive? First, we need to get Twitter data into HDFS, and then we'll be able to tell Hive where the data resides and how to read it.



The diagram above shows a high-level view of how some of the CDH (Cloudera's Distribution Including Apache Hadoop) components can be pieced together to build the data pipeline we need to answer the questions we have. The rest of this post will describe how these components interact and the purposes they each serve.

Gathering Data with Apache Flume

The Twitter Streaming API will give us a constant stream of tweets coming from the service. One option would be to use a simple utility like curl to access the API and then periodically load the files. However, this would require us to write code to control where the data goes in HDFS, and if we have a secure cluster, we will have to integrate with security mechanisms. It will be much simpler to use components within CDH to automatically move the files from the API to HDFS, without our manual intervention.

Apache Flume is a data ingestion system that is configured by defining endpoints in a data flow called sources and sinks. In Flume, each individual piece of data (tweets, in our case) is called an event; sources produce events, and send the events through a channel, which connects the source to the sink. The sink then writes the events out to a predefined location. Flume supports some standard data sources, such as syslog or netcat. For this use case, we'll need to design a custom source that accesses the Twitter Streaming API, and sends the tweets through a channel to a sink that writes to HDFS files. Additionally, we can use the custom source to filter the tweets on a set of search keywords to help identify relevant tweets, rather than a pure sample of the entire Twitter firehose. The custom Flume source code can be found here.

Partition Management with Oozie

Once we have the Twitter data loaded into HDFS, we can stage it for querying by creating an external table in Hive. Using an external table will allow us to query the table without moving the data from the location where it ends up in HDFS. To ensure scalability, as we add more and more data, we'll need to also partition the table. A partitioned table allows us to prune the files that we read when querying, which results in better performance when dealing with large data sets. However, the Twitter API will continue to stream tweets and Flume will perpetually create new files. We can automate the periodic process of adding partitions to our table as the new data comes in.

Apache Oozie is a workflow coordination system that can be used to solve this problem. Oozie is an extremely flexible system for designing job workflows, which can be scheduled to run based on a set of criteria. We can configure the workflow to run an ALTER TABLE command that adds a partition containing the last hour's worth of data into Hive, and we can instruct the workflow to occur every hour. This will ensure that we're always looking at up-to-date data.

Querying Complex Data with Hive

Before we can query the data, we need to ensure that the Hive table can properly interpret the JSON data. By default, Hive expects that input files use a delimited row format, but our Twitter data is in a JSON format, which will not work with the defaults. This is actually one of Hive's biggest strengths. Hive allows us to flexibly define, and redefine, how the data is represented on disk. The schema is only really enforced when we read the data, and we can use the Hive SerDe interface to specify how to interpret what we've loaded.

SerDe stands for Serializer and Deserializer, which are interfaces that tell Hive how it should translate the data into something that Hive can process. In particular, the Deserializer interface is used when we read data off of disk, and converts the data into objects that Hive knows how to manipulate. We can write a custom SerDe that reads the JSON data in and translates the objects for Hive. Once that's put into place, we can start querying. The SerDe will take a tweet in JSON form, like the following:


```
1  {
2    "retweeted_status": {
3      "contributors": null,
4      "text": "#Crowdsourcing – drivers already generate traffic data for your smartphone to suggest
5  alternative routes when a road is clogged. #bigdata",
6      "geo": null,
7      "retweeted": false,
8      "in_reply_to_screen_name": null,
9      "truncated": false,
10     "entities": {
```

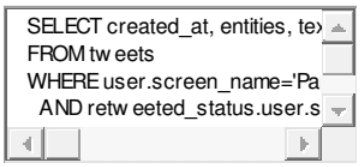
```
        "urls": [],
        "hashtags": [
          {
            "text": "Crowdsourcing",
            "indices": [
              0,
              14
            ]
          },
          {
            "text": "bigdata",
            "indices": [
              129,
              137
            ]
          }
        ],
        "user_mentions": []
      },
      "in_reply_to_status_id_str": null,
      "id": 245255511388336128,
      "in_reply_to_user_id_str": null,
      "source": "SocialOomph",
      "favorited": false,
      "in_reply_to_status_id": null,
      "in_reply_to_user_id": null,
      "retweet_count": 0,
      "created_at": "Mon Sep 10 20:20:45 +0000 2012",
      "id_str": "245255511388336128",
      "place": null,
      "user": {
        "location": "Oregon, ",
        "default_profile": false,
        "statuses_count": 5289,
        "profile_background_tile": false,
        "lang": "en",
        "profile_link_color": "627E91",
        "id": 347471575,
        "following": null,
        "protected": false,
        "favourites_count": 17,
        "profile_text_color": "D4B020",
        "verified": false,
        "description": "Dad, Innovator, Sales Professional. Project Management Professional
(PMP).  Soccer Coach,  Little League Coach  #Agile #PMOT - views are my own -",
        "contributors_enabled": false,
        "name": "Scott Ostby",
        "profile_sidebar_border_color": "404040",
        "profile_background_color": "0F0F0F",
        "created_at": "Tue Aug 02 21:10:39 +0000 2011",
```

```
        "default_profile_image": false,
        "followers_count": 19005,
        "profile_image_url_https": "https://si0.twimg.com/profile_images/1928022765/scott_normal.jpg",
        "geo_enabled": true,
        "profile_background_image_url":
"http://a0.twimg.com/profile_background_images/327807929/xce5b8c5dfff3dc3bbfbdef5ca2a62b4.jpg",
        "profile_background_image_url_https":
"https://si0.twimg.com/profile_background_images/327807929/xce5b8c5dfff3dc3bbfbdef5ca2a62b4.jpg",
        "follow_request_sent": null,
        "url": "http://facebook.com/ostby",
        "utc_offset": -28800,
        "time_zone": "Pacific Time (US &amp;amp; Canada)",
        "notifications": null,
        "friends_count": 13172,
        "profile_use_background_image": true,
        "profile_sidebar_fill_color": "1C1C1C",
        "screen_name": "ScottOstby",
        "id_str": "347471575",
        "profile_image_url": "http://a0.twimg.com/profile_images/1928022765/scott_normal.jpg",
        "show_all_inline_media": true,
        "is_translator": false,
        "listed_count": 45
      },
      "coordinates": null
    },
    "contributors": null,
    "text": "RT @ScottOstby: #Crowdsourcing – drivers already generate traffic data for your smartphone
to suggest alternative routes when a road is  ...",
    "geo": null,
    "retweeted": false,
    "in_reply_to_screen_name": null,
    "truncated": false,
    "entities": {
      "urls": [],
      "hashtags": [
        {
          "text": "Crowdsourcing",
          "indices": [
            16,
            30
          ]
        }
      ],
      "user_mentions": [
        {
          "id": 347471575,
          "name": "Scott Ostby",
          "indices": [
            3,
            14
```

```
          ],
          "screen_name": "ScottOstby",
          "id_str": "347471575"
        }
      ]
    },
    "in_reply_to_status_id_str": null,
    "id": 245270269525123072,
    "in_reply_to_user_id_str": null,
    "source": "web",
    "favorited": false,
    "in_reply_to_status_id": null,
    "in_reply_to_user_id": null,
    "retweet_count": 0,
    "created_at": "Mon Sep 10 21:19:23 +0000 2012",
    "id_str": "245270269525123072",
    "place": null,
    "user": {
      "location": "",
      "default_profile": true,
      "statuses_count": 1294,
      "profile_background_tile": false,
      "lang": "en",
      "profile_link_color": "0084B4",
      "id": 21804678,
      "following": null,
      "protected": false,
      "favourites_count": 11,
      "profile_text_color": "333333",
      "verified": false,
      "description": "",
      "contributors_enabled": false,
      "name": "Parvez Jugon",
      "profile_sidebar_border_color": "C0DEED",
      "profile_background_color": "C0DEED",
      "created_at": "Tue Feb 24 22:10:43 +0000 2009",
      "default_profile_image": false,
      "followers_count": 70,
      "profile_image_url_https":
"https://si0.twimg.com/profile_images/2280737846/ni91dkogtgwp1or5rwp4_normal.gif",
      "geo_enabled": false,
      "profile_background_image_url": "http://a0.twimg.com/images/themes/theme1/bg.png",
      "profile_background_image_url_https": "https://si0.twimg.com/images/themes/theme1/bg.png",
      "follow_request_sent": null,
      "url": null,
      "utc_offset": null,
      "time_zone": null,
      "notifications": null,
      "friends_count": 299,
      "profile_use_background_image": true,
```

```
161       "profile_sidebar_fill_color": "DDEEF6",
162       "screen_name": "ParvezJugon",
163       "id_str": "21804678",
164       "profile_image_url":
      "http://a0.twimg.com/profile_images/2280737846/ni91dkogtgwp1or5rwp4_normal.gif",
          "show_all_inline_media": false,
          "is_translator": false,
          "listed_count": 7
        },
        "coordinates": null
      }
```


and translate the JSON entities into queryable columns:




```
1 SELECT created_at, entities, text, user
2 FROM tweets
3 WHERE user.screen_name='ParvezJugon'
4  AND retweeted_status.user.screen_name='ScottOstby';
```

which will result in:

created_at entities

text use
r
1 Mon Sep 10 21:19:23 +0000
2 2012 {"urls":[],"user_mentions":[{"screen_name":"ScottOstby","name":"Scott Ostby"}],"hashtags":[{"text":"Crowdsourcing"}]} RT @ScottOstby: #Crowdsourcing – drivers already generate traffic data for your smartphone to suggest alternative routes when a road is ... {"screen_name":"ParvezJugon","name":"Parvez Jugon","friends_count":299,"followers_count":70,"statuses_count":1294,"verified":false,"utc_offset":null,"time_zone":null}

We've now managed to put together an end-to-end system, which gathers data from the Twitter Streaming API, sends the tweets to files on HDFS through Flume, and uses Oozie to periodically load the files into Hive, where we can query the raw JSON data, through the use of a Hive SerDe.

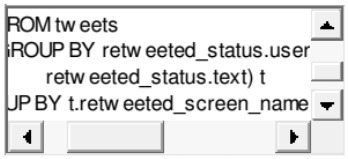
Some Results

In my own testing, I let Flume collect data for about three days, filtering on a set of keywords:

hadoop, big data, analytics, bigdata, cloudera, data science, data scientist, business intelligence, mapreduce, data warehouse, data warehousing, mahout, hbase, nosql, newsql, businessintelligence, cloudcomputing

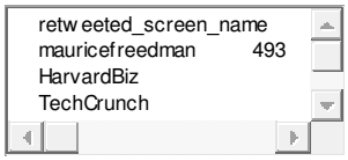
The collected data was about half a GB of JSON data. The data has some structure, but certain fields may or may not exist. The retweeted_status field, for example, will only be present if the tweet was a retweet. Additionally, some of the fields may be arbitrarily complex. The hashtags field is an array of all the hashtags present in the tweets, but most RDBMS's do not support arrays as a column type. This semi-structured quality of the data makes the data very difficult to query in a traditional RDBMS. Hive can handle this data much more gracefully.

The query below will find usernames, and the number of retweets they have generated across all the tweets that we have data for:




```
1  SELECT
2   t.retweeted_screen_name,
3   sum(retweets) AS total_retweets,
4   count(*) AS tweet_count
5  FROM (SELECT
6         retweeted_status.user.screen_name as retweeted_screen_name,
7        retweeted_status.text,
8        max(retweet_count) as retweets
9        FROM tweets
10       GROUP BY retweeted_status.user.screen_name,
11              retweeted_status.text) t
12 GROUP BY t.retweeted_screen_name
13 ORDER BY total_retweets DESC
14 LIMIT 10;
```

For the few days of data, I found that these were the most retweeted users for the industry:




```
1   retweeted_screen_name total_retweets tweet_count
2   mauricefreedman       493       1
```

3	HarvardBiz	362	6
4	TechCrunch	314	7
5	googleanalytics	244	10
6	BigDataBorat	201	6
7	stephen_wolfram	182	1
8	CloudExpo	153	28
9	TheNextWeb	150	1
10	GonzalezCarmen	121	10
11	IBMbigdata	100	37

From these results, we can see whose tweets are getting heard by the widest audience, and also determine whether these people are communicating on a regular basis or not. We can use this information to more carefully target our messaging in order to get them talking about our products, which, in turn, will get other people talking about our products.