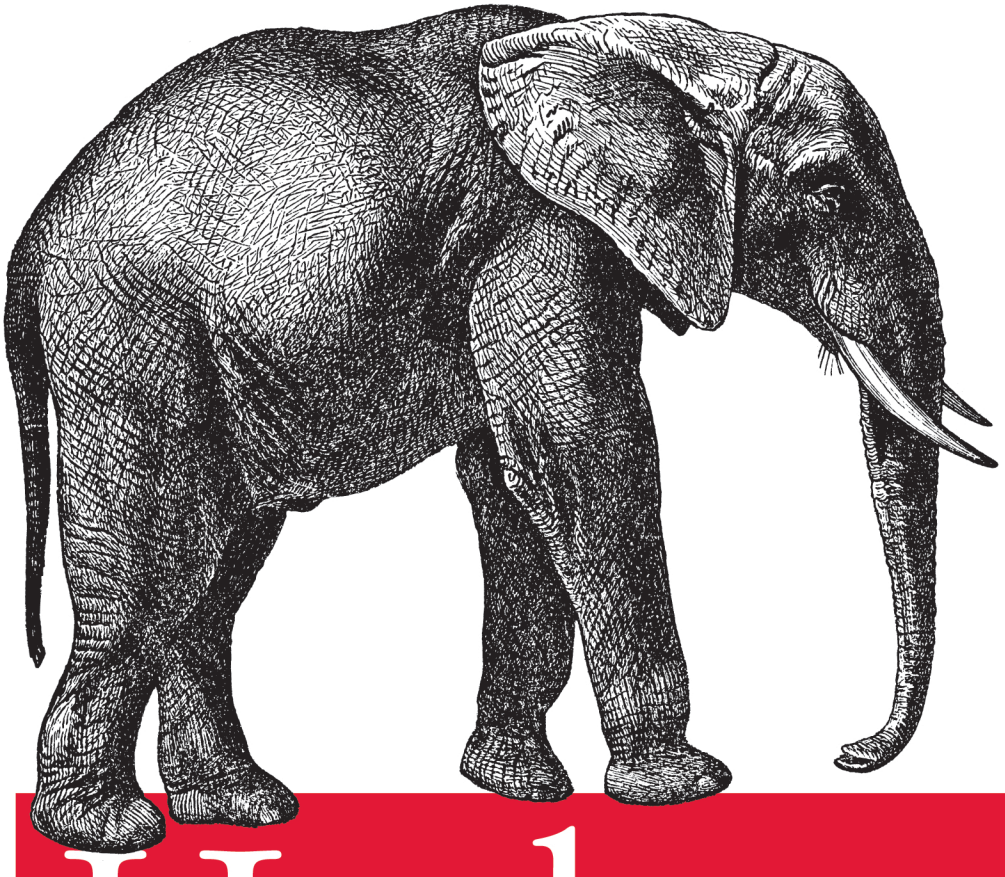
Storage and Analysis at Internet Scale

3rd Edition
Revised & Updated



Hadoop

The Definitive Guide

O'REILLY®

Tom White

Hadoop: The Definitive Guide

Ready to unlock the power of your data? With this comprehensive guide, you'll learn how to build and maintain reliable, scalable, distributed systems with Apache Hadoop. This book is ideal for programmers looking to analyze datasets of any size, and for administrators who want to set up and run Hadoop clusters.

You'll find illuminating case studies that demonstrate how Hadoop is used to solve specific problems. This third edition covers recent changes to Hadoop, including material on the new MapReduce API, as well as MapReduce 2 and its more flexible execution model (YARN).

- Store large datasets with the Hadoop Distributed File System (HDFS)
- Run distributed computations with MapReduce
- Use Hadoop's data and I/O building blocks for compression, data integrity, serialization (including Avro), and persistence
- Discover common pitfalls and advanced features for writing real-world MapReduce programs
- Design, build, and administer a dedicated Hadoop cluster—or run Hadoop in the cloud
- Load data from relational databases into HDFS, using Sqoop
- Perform large-scale data processing with the Pig query language
- Analyze datasets with Hive, Hadoop's data warehousing system
- Take advantage of HBase for structured and semi-structured data, and ZooKeeper for building distributed systems

"Now you have the opportunity to learn about Hadoop from a master—not only of the technology, but also of common sense and plain talk."

—**Doug Cutting,** *Cloudera*

Tom White, an engineer at Cloudera and member of the Apache Software Foundation, has been an Apache Hadoop committer since February 2007. He has written numerous articles for oreilly.com, java.net, and IBM's developerWorks, and speaks regularly about Hadoop at industry conferences.

cloudera

Cloudera is a leading provider of Hadoop-based software and services. Cloudera's Distribution for Hadoop (CDH) is a comprehensive Apache Hadoop-based data management platform and Cloudera Enterprise includes the tools, platform, and support necessary to use Hadoop in production.

Strata
Making Data Work

Strata is the emerging ecosystem of people, tools, and technologies that turn big data into smart decisions. Find information and resources at oreilly.com/data.

US $49.99 CAN $52.99
ISBN: 978-1-449-31152-0

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

Hadoop: The Definitive Guide, Third Edition
by Tom White

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Rachel Steely
Copyeditor: Genevieve d'Entremont
Proofreader: Kevin Broccoli

Indexer: Kevin Broccoli
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

May 2012: Third Edition.

Revision History for the Third Edition:
2012-01-27 Early release revision 1
2012-05-07 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449311520* for release details.

ISBN: 978-1-449-31152-0

[LSI]

1336503003

CHAPTER 16

Case Studies

Hadoop Usage at Last.fm

Last.fm: The Social Music Revolution

Founded in 2002, Last.fm is an Internet radio and music community website that offers many services to its users, such as free music streams and downloads, music and event recommendations, personalized charts, and much more. There are about 25 million people who use Last.fm every month, generating huge amounts of data that need to be processed. One example of this is users transmitting information indicating which songs they are listening to (this is known as "scrobbling"). This data is processed and stored by Last.fm, so the user can access it directly (in the form of charts), and it is also used to make decisions about users' musical tastes and compatibility, and artist and track similarity.

Hadoop at Last.fm

As Last.fm's service developed and the number of users grew from thousands to millions, storing, processing, and managing all the incoming data became increasingly challenging. Fortunately, Hadoop was quickly becoming stable enough and was enthusiastically adopted as it became clear how many problems it solved. It was first used at Last.fm in early 2006 and was put into production a few months later. There were several reasons for adopting Hadoop at Last.fm:

- The distributed filesystem provided redundant backups for the data stored on it (e.g., web logs, user listening data) at no extra cost.
- Scalability was simplified through the ability to add cheap commodity hardware when required.
- The cost was right (free) at a time when Last.fm had limited financial resources.

- The open source code and active community meant that Last.fm could freely modify Hadoop to add custom features and patches.
- Hadoop provided a flexible framework for running distributed computing algorithms with a relatively easy learning curve.

Hadoop has now become a crucial part of Last.fm's infrastructure, currently consisting of two Hadoop clusters spanning over 50 machines, 300 cores, and 100 TB of disk space. Hundreds of daily jobs are run on the clusters performing operations, such as logfile analysis, evaluation of A/B tests, ad hoc processing, and charts generation. This case study will focus on the process of generating charts, as this was the first usage of Hadoop at Last.fm and illustrates the power and flexibility that Hadoop provides over other approaches when working with very large datasets.

Generating Charts with Hadoop

Last.fm uses user-generated track listening data to produce many different types of charts, such as weekly charts for tracks, per country and per user. A number of Hadoop programs are used to process the listening data and generate these charts, and these run on a daily, weekly, or monthly basis. Figure 16-1 shows an example of how this data is displayed on the site, in this case, the weekly top tracks.

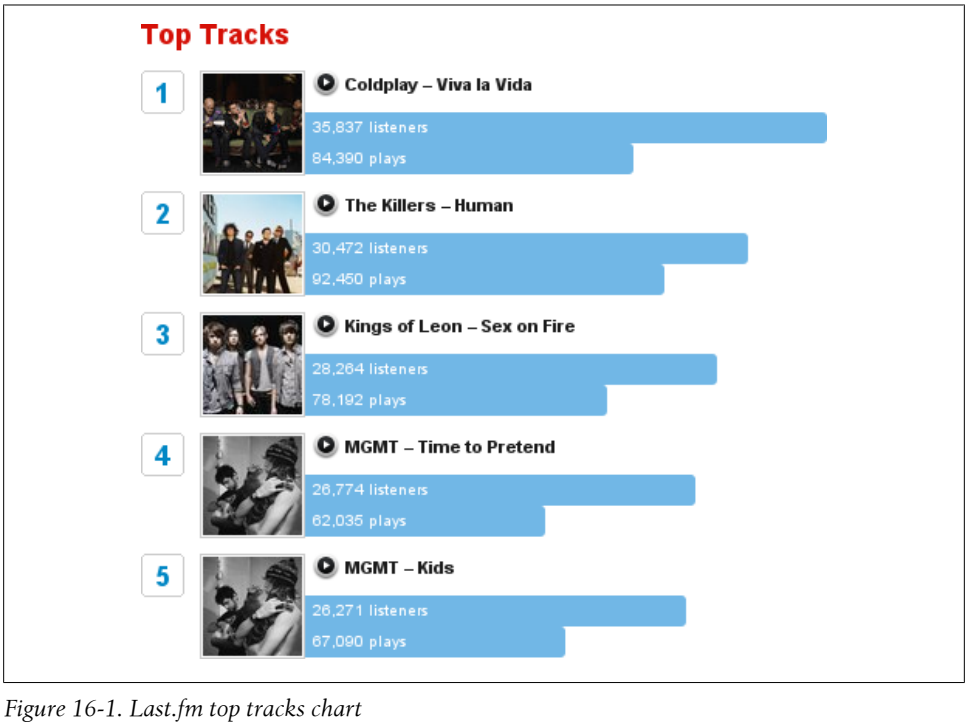


Figure 16-1. Last.fm top tracks chart

Listening data typically arrives at Last.fm from one of two sources:

- A user plays a track of his own (e.g., listening to an MP3 file on a PC or other device), and this information is sent to Last.fm using either the official Last.fm client application or one of many hundreds of third-party applications.
- A user tunes into one of Last.fm's Internet radio stations and streams a song to his computer. The Last.fm player or website can be used to access these streams and extra functionality is made available to the user, allowing him to love, skip, or ban each track that he listens to.

When processing the received data, we distinguish between a track listen submitted by a user (the first source in the previous list, referred to as a *scrobble* from here on) and a track listened to on the Last.fm radio (the second source, mentioned earlier, referred to as a *radio listen* from here on). This distinction is very important in order to prevent a feedback loop in the Last.fm recommendation system, which is based only on scrobbles. One of the most fundamental Hadoop jobs at Last.fm takes the incoming listening data and summarizes it into a format that can be used for display purposes on the Last.fm website as well as for input to other Hadoop programs. This is achieved by the Track Statistics program, which is the example described in the following sections.

The Track Statistics Program

When track listening data is submitted to Last.fm, it undergoes a validation and conversion phase, the end result of which is a number of space-delimited text files containing the user ID, the track ID, the number of times the track was scrobbled, the number of times the track was listened to on the radio, and the number of times it was skipped. Table 16-1 contains sample listening data, which is used in the following examples as input to the Track Statistics program (the real data is gigabytes in size and includes many more fields that have been omitted here for simplicity's sake).

Table 16-1. Listening data

UserId	TrackId	Scrobble	Radio	Skip
111115	222	0	1	0
111113	225	1	0	0
111117	223	0	1	1
111115	225	1	0	0

These text files are the initial input provided to the Track Statistics program, which consists of two jobs that calculate various values from this data and a third job that merges the results (see Figure 16-2).

The Unique Listeners job calculates the total number of unique listeners for a track by counting the first listen by a user and ignoring all other listens by the same user. The Sum job accumulates the total listens, scrobbles, radio listens, and skips for each track

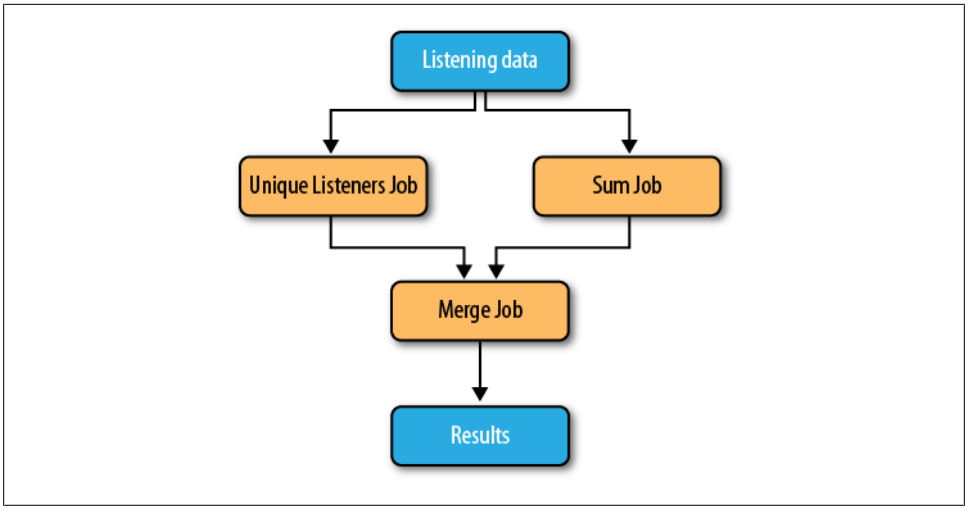


Figure 16-2. TrackStats jobs

by counting these values for all listens by all users. Although the input format of these two jobs is identical, two separate jobs are needed, as the Unique Listeners job is responsible for emitting values per track per user, and the Sum job emits values per track. The final Merge job is responsible for merging the intermediate output of the two other jobs into the final result. The end results of running the program are the following values per track:

- Number of unique listeners
- Number of times the track was scrobbled
- Number of times the track was listened to on the radio
- Number of times the track was listened to in total
- Number of times the track was skipped on the radio

Each job and its MapReduce phases are described in more detail next. Please note that the provided code snippets have been simplified due to space constraints; for download details for the full code listings, refer to the Preface.

Calculating the number of unique listeners

The Unique Listeners job calculates, per track, the number of unique listeners.

UniqueListenersMapper. The `UniqueListenersMapper` processes the space-delimited raw listening data and emits the user ID associated with each track ID:

```
public void map(LongWritable position, Text rawLine, OutputCollector<IntWritable,
    IntWritable> output, Reporter reporter) throws IOException {

  String[] parts = (rawLine.toString()).split(" ");
```

```
    int scrobbles = Integer.parseInt(parts[TrackStatisticsProgram.COL_SCROBBLES]);
    int radioListens = Integer.parseInt(parts[TrackStatisticsProgram.COL_RADIO]);
    // if track somehow is marked with zero plays - ignore
    if (scrobbles <= 0 && radioListens <= 0) {
      return;
    }
    // if we get to here then user has listened to track,
    // so output user id against track id
    IntWritable trackId = new IntWritable(
      Integer.parseInt(parts[TrackStatisticsProgram.COL_TRACKID]));
    IntWritable userId = new IntWritable(
      Integer.parseInt(parts[TrackStatisticsProgram.COL_USERID]));
    output.collect(trackId, userId);
  }
```

UniqueListenersReducer. The `UniqueListenersReducers` receives a list of user IDs per track ID and puts these IDs into a `Set` to remove any duplicates. The size of this set is then emitted (i.e., the number of unique listeners) for each track ID. Storing all the reduce values in a `Set` runs the risk of running out of memory if there are many values for a certain key. This hasn't happened in practice, but to overcome this, an extra MapReduce step could be introduced to remove all the duplicate values, or a secondary sort could be used (for more details, see "Secondary Sort" on page 277):

```
  public void reduce(IntWritable trackId, Iterator<IntWritable> values,
      OutputCollector<IntWritable, IntWritable> output, Reporter reporter)
      throws IOException {

    Set<Integer> userIds = new HashSet<Integer>();
    // add all userIds to the set, duplicates automatically removed (set contract)
    while (values.hasNext()) {
      IntWritable userId = values.next();
      userIds.add(Integer.valueOf(userId.get()));
    }
    // output trackId -> number of unique listeners per track
    output.collect(trackId, new IntWritable(userIds.size()));
  }
```

Table 16-2 shows the sample input data for the job. The map output appears in Table 16-3 and the reduce output in Table 16-4.

Table 16-2. Job input

Line of file	UserId	TrackId	Scrobbled	Radio play	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	11115	222	0	1	0
1	11113	225	1	0	0
2	11117	223	0	1	1
3	11115	225	1	0	0

Table 16-3. Mapper output

TrackId	UserId
IntWritable	IntWritable
222	11115
225	11113
223	11117
225	11115

Table 16-4. Reducer output

TrackId	#listeners
IntWritable	IntWritable
222	1
225	2
223	1

Summing the track totals

The Sum job is relatively simple; it just adds up the values we are interested in for each track.

SumMapper. The input data is again the raw text files, but in this case, it is handled quite differently. The desired end result is a number of totals (unique listener count, play count, scrobble count, radio listen count, skip count) associated with each track. To simplify things, we use an intermediate `TrackStats` object generated using Hadoop Record I/O, which implements `WritableComparable` (so it can be used as output) to hold these values. The Mapper creates a `TrackStats` object and sets the values on it for each line in the file, except for the unique listener count, which is left empty (it will be filled in by the final merge job):

```
public void map(LongWritable position, Text rawLine,
    OutputCollector<IntWritable, TrackStats> output, Reporter reporter)
    throws IOException {

  String[] parts = (rawLine.toString()).split(" ");
  int trackId = Integer.parseInt(parts[TrackStatisticsProgram.COL_TRACKID]);
  int scrobbles = Integer.parseInt(parts[TrackStatisticsProgram.COL_SCROBBLES]);
  int radio = Integer.parseInt(parts[TrackStatisticsProgram.COL_RADIO]);
  int skip = Integer.parseInt(parts[TrackStatisticsProgram.COL_SKIP]);
  // set number of listeners to 0 (this is calculated later)
  // and other values as provided in text file
  TrackStats trackstat = new TrackStats(0, scrobbles + radio, scrobbles, radio, skip);
  output.collect(new IntWritable(trackId), trackstat);
}
```

SumReducer. In this case, the reducer performs a very similar function to the Mapper, summing the statistics per track and returning an overall total:

```
public void reduce(IntWritable trackId, Iterator<TrackStats> values,
    OutputCollector<IntWritable, TrackStats> output, Reporter reporter)
    throws IOException {

  TrackStats sum = new TrackStats(); // holds the totals for this track
  while (values.hasNext()) {
    TrackStats trackStats = (TrackStats) values.next();
    sum.setListeners(sum.getListeners() + trackStats.getListeners());
    sum.setPlays(sum.getPlays() + trackStats.getPlays());
    sum.setSkips(sum.getSkips() + trackStats.getSkips());
    sum.setScrobbles(sum.getScrobbles() + trackStats.getScrobbles());
    sum.setRadioPlays(sum.getRadioPlays() + trackStats.getRadioPlays());
  }
  output.collect(trackId, sum);
}
```

Table 16-5 shows the input data for the job (the same as for the Unique Listeners job). The map output appears in Table 16-6 and the reduce output in Table 16-7.

Table 16-5. Job input

Line	UserId	TrackId	Scrobbled	Radio play	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	11115	222	0	1	0
1	11113	225	1	0	0
2	11117	223	0	1	1
3	11115	225	1	0	0

Table 16-6. Map output

TrackId	#listeners	#plays	#scrobbles	#radio plays	#skips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
225	0	1	1	0	0
223	0	1	0	1	1
225	0	1	1	0	0

Table 16-7. Reduce output

TrackId	#listeners	#plays	#scrobbles	#radio plays	#skips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
225	0	2	2	0	0
223	0	1	0	1	1

Merging the results

The final job needs to merge the output from the two previous jobs: the number of unique listeners per track and the statistics per track. In order to merge these different inputs, two different mappers (one for each type of input) are used. The two intermediate jobs are configured to write their results to different paths, and the `MultipleInputs` class is used to specify which Mapper will process which files. The following code shows how the `JobConf` for the job is set up to do this:

```
MultipleInputs.addInputPath(conf, sumInputDir,
  SequenceFileInputFormat.class, IdentityMapper.class);

MultipleInputs.addInputPath(conf, listenersInputDir,
    SequenceFileInputFormat.class, MergeListenersMapper.class);
```

It is possible to use a single Mapper to handle different inputs, but the example solution is more convenient and elegant.

MergeListenersMapper. This Mapper is used to process the `UniqueListenerJob`'s output of unique listeners per track. It creates a `TrackStats` object in a similar manner to the `SumMapper`, but this time, it fills in only the unique listener count per track and leaves the other values empty:

```
public void map(IntWritable trackId, IntWritable uniqueListenerCount,
    OutputCollector<IntWritable, TrackStats> output, Reporter reporter)
    throws IOException {
  TrackStats trackStats = new TrackStats();
  trackStats.setListeners(uniqueListenerCount.get());
  output.collect(trackId, trackStats);
}
```

Table 16-8 shows some input for the Mapper; the corresponding output is shown in Table 16-9.

Table 16-8. MergeListenersMapper input

TrackId	#listeners
IntWritable	IntWritable
222	1
225	2
223	1

Table 16-9. MergeListenersMapper output

TrackId	#listeners	#plays	#scrobbles	#radio	#skips
222	1	0	0	0	0
225	2	0	0	0	0
223	1	0	0	0	0

IdentityMapper. The `IdentityMapper` is configured to process the `SumJob`'s output of `TrackStats` objects and, as no additional processing is required, directly emits the input data (see Table 16-10).

Table 16-10. IdentityMapper input and output

TrackId	#listeners	#plays	#scrobbles	#radio	#skips
`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`
222	0	1	0	1	0
225	0	2	2	0	0
223	0	1	0	1	1

SumReducer. The two Mappers emit values of the same type: a `TrackStats` object per track, with different values filled in. The final reduce phase can reuse the `SumReducer` described earlier to create a `TrackStats` object per track, sum up all the values, and emit it (see Table 16-11).

Table 16-11. Final SumReducer output

TrackId	#listeners	#plays	#scrobbles	#radio	#skips
`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`
222	1	1	0	1	0
225	2	2	2	0	0
223	1	1	0	1	1

The final output files are then accumulated and copied to a server, where a web service makes the data available to the Last.fm website for display. An example of this is shown in Figure 16-3, where the total number of listeners and plays are displayed for a track.

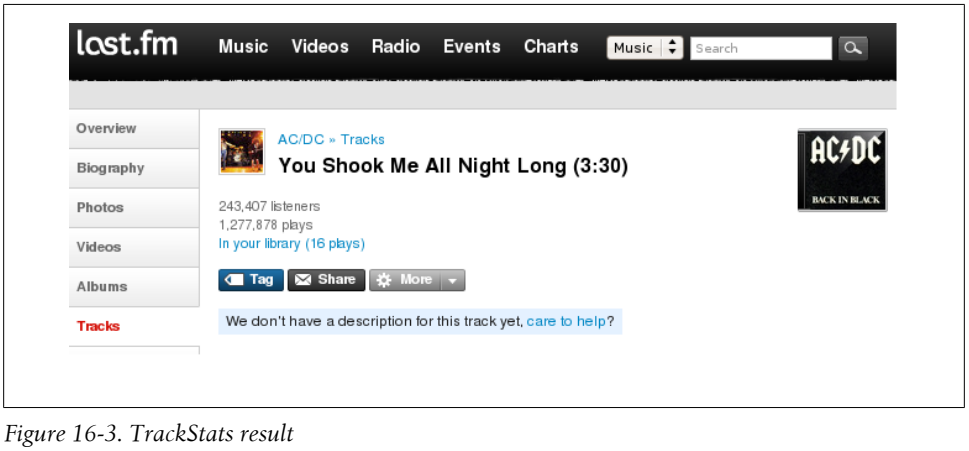


Figure 16-3. TrackStats result

Summary

Hadoop has become an essential part of Last.fm's infrastructure and is used to generate and process a wide variety of datasets ranging from web logs to user listening data. The example covered here has been simplified considerably in order to get the key concepts across; in real-world usage, the input data has a more complicated structure and the code that processes it is more complex. Hadoop itself, although mature enough for production use, is still in active development, and new features and improvements are added by the Hadoop community every week. We at Last.fm are happy to be part of this community as a contributor of code and ideas, and as end users of a great piece of open source technology.

—Adrian Woodhead and Marc de Palol

Hadoop and Hive at Facebook

Hadoop can be used to form core backend batch and near real-time computing infrastructures. It can also be used to store and archive massive datasets. In this case study, we will explore backend data architectures and the role Hadoop can play in them. We will describe hypothetical Hadoop configurations, potential uses of Hive—an open source data warehousing and SQL infrastructure built on top of Hadoop—and the different kinds of business and product applications that have been built using this infrastructure.

Hadoop at Facebook

History

The amount of log and dimension data in Facebook that needs to be processed and stored has exploded as the usage of the site has increased. A key requirement for any data processing platform for this environment is the ability to scale rapidly. Further, with limited engineering resources, the system should be very reliable and easy to use and maintain.

Initially, data warehousing at Facebook was performed entirely on an Oracle instance. After we started hitting scalability and performance problems, we investigated whether there were open source technologies that could be used in our environment. As part of this investigation, we deployed a relatively small Hadoop instance and started publishing some of our core datasets into this instance. Hadoop was attractive because Yahoo! was using it internally for its batch processing needs and because we were familiar with the simplicity and scalability of the MapReduce model as popularized by Google.

Our initial prototype was very successful: the engineers loved the ability to process massive amounts of data in reasonable timeframes, an ability that we just did not have

before. They also loved being able to use their favorite programming language for processing (using Hadoop streaming). Having our core datasets published in one centralized data store was also very convenient. At around the same time, we started developing Hive. This made it even easier for users to process data in the Hadoop cluster by being able to express common computations in the form of SQL, a language with which most engineers and analysts are familiar.

As a result, the cluster size and usage grew by leaps and bounds, and today Facebook is running the second-largest Hadoop cluster in the world. As of this writing, we hold more than 2 PB of data in Hadoop and load more than 10 TB of data into it every day. Our Hadoop instance has 2,400 cores and about 9 TB of memory, and runs at 100% utilization at many points during the day. We are able to scale out this cluster rapidly in response to our growth, and we have been able to take advantage of open source by modifying Hadoop where required to suit our needs. We have contributed back to open source, both in the form of contributions to some core components of Hadoop as well as by open-sourcing Hive, which is now a Hadoop top-level project.

Use cases

There are at least four interrelated but distinct classes of uses for Hadoop at Facebook:

- Producing daily and hourly summaries over large amounts of data. These summaries are used for a number of different purposes within the company:
  - Reports based on these summaries are used by engineering and nonengineering functional teams to drive product decisions. These summaries include reports on growth of the users, page views, and average time spent on the site by the users.
  - Providing performance numbers about advertisement campaigns that are run on Facebook.
  - Backend processing for site features such as people you may like and applications you may like.
- Running ad hoc jobs over historical data. These analyses help answer questions from our product groups and executive team.
- As a de facto long-term archival store for our log datasets.
- To look up log events by specific attributes (where logs are indexed by such attributes), which is used to maintain the integrity of the site and protect users against spambots.

Data architecture

Figure 16-4 shows the basic components of our architecture and the data flow within these components.

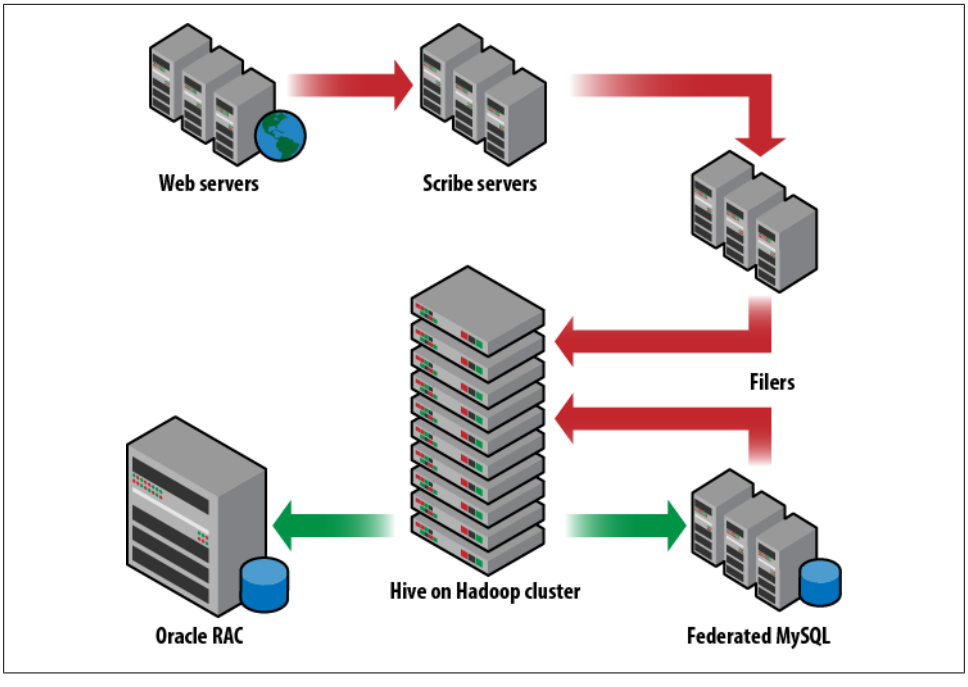


Figure 16-4. Data warehousing architecture at Facebook

As shown in Figure 16-4, the following components are used in processing data:

Scribe
: Log data is generated by web servers as well as internal services such as the Search backend. We use Scribe, an open source log collection service developed in Facebook that deposits hundreds of log datasets with a daily volume in the tens of terabytes into a handful of NFS servers.

HDFS
: A large fraction of this log data is copied into one central HDFS instance. Dimension data is also scraped from our internal MySQL databases and copied over into HDFS daily.

Hive/Hadoop
: We use Hive, a Hadoop subproject developed in Facebook, to build a data warehouse over all the data collected in HDFS. Files in HDFS, including log data from Scribe and dimension data from the MySQL tier, are made available as tables with logical partitions. A SQL-like query language provided by Hive is used in conjunction with MapReduce to create/publish a variety of summaries and reports, as well as to perform historical analysis over these tables.

Tools
: Browser-based interfaces built on top of Hive allow users to compose and launch Hive queries (which in turn launch MapReduce jobs) using just a few mouse clicks.

Traditional RDBMS
: We use Oracle and MySQL databases to publish these summaries. The volume of data here is relatively small, but the query rate is high and needs real-time response.

DataBee
: An in-house Extract, Transform, Load (ETL) workflow software that is used to provide a common framework for reliable batch processing across all data processing jobs.

Data from the NFS tier storing Scribe data is continuously replicated to the HDFS cluster by copier jobs. The NFS devices are mounted on the Hadoop tier, and the copier processes run as Map-only jobs on the Hadoop cluster. This makes it easy to scale the copier processes and makes them fault-resilient. Currently, we copy over 6 TB per day from Scribe to HDFS in this manner. We also download up to 4 TB of dimension data from our MySQL tier to HDFS every day. These are also conveniently arranged on the Hadoop cluster as Map-only jobs that copy data out of MySQL boxes.

Hadoop configuration

The central philosophy behind our Hadoop deployment is consolidation. We use a single HDFS instance, and a vast majority of processing is done in a single MapReduce cluster (running a single jobtracker). The reasons for this are fairly straightforward:

- We can minimize the administrative overheads by operating a single cluster.
- Data does not need to be duplicated. All data is available in a single place for all the use cases described previously.
- By using the same compute cluster across all departments, we get tremendous efficiencies.
- Our users work in a collaborative environment, so requirements in terms of quality of service are not onerous (yet).

We also have a single shared Hive metastore (using a MySQL database) that holds metadata about all the Hive tables stored in HDFS.

Hypothetical Use Case Studies

In this section, we will describe some typical problems that are common for large websites, which are difficult to solve through traditional warehousing technologies, simply because the costs and scales involved are prohibitively high. Hadoop and Hive can provide a more scalable and more cost-effective solution in such situations.

Advertiser insights and performance

One of the most common uses of Hadoop is to produce summaries from large volumes of data. It is very typical of large ad networks, such as the Facebook ad network, Google AdSense, and many others, to provide advertisers with standard aggregated statistics

about their ads that help the advertisers tune their campaigns effectively. Computing advertisement performance numbers on large datasets is a very data-intensive operation, and the scalability and cost advantages of Hadoop and Hive can really help in computing these numbers in a reasonable time frame and at a reasonable cost.

Many ad networks provide standardized CPC- and CPM-based ad units to the advertisers. The CPC ads are cost-per-click ads: the advertiser pays the ad network amounts that are dependent on the number of clicks that the particular ad gets from the users visiting the site. The CPM ads (short for *cost per mille*, that is, the cost per thousand impressions), on the other hand, bill the advertisers amounts that are proportional to the number of users who see the ad on the site. Apart from these standardized ad units, in the last few years ads with more dynamic content that is tailored to each individual user have also become common in the online advertisement industry. Yahoo! does this through SmartAds, whereas Facebook provides its advertisers with Social Ads. The latter allows the advertisers to embed information from a user's network of friends; for example, a Nike ad may refer to a friend of the user who recently fanned Nike and shared that information with her friends on Facebook. In addition, Facebook also provides Engagement Ad units to the advertisers, wherein the users can more effectively interact with the ad, be it by commenting on it or by playing embedded videos. In general, a wide variety of ads are provided to the advertisers by the online ad networks, and this variety adds yet another dimension to the various kinds of performance numbers that the advertisers are interested in getting about their campaigns.

At the most basic level, advertisers are interested in knowing the total number of users, as well as the number of unique users, who have seen the ad or have clicked on it. For more dynamic ads, they may even be interested in getting the breakdown of these aggregated numbers by the kind of dynamic information shown in the ad unit or the kind of engagement action undertaken by the users on the ad. For example, a particular advertisement may have been shown 100,000 times to 30,000 unique users. Similarly, a video embedded inside an Engagement Ad may have been watched by 100,000 unique users. In addition, these performance numbers are typically reported for each ad, campaign, and account. An account may have multiple campaigns, with each campaign running multiple ads on the network. Finally, these numbers are typically reported for different time durations by the ad networks. Typical durations are daily, rolling week, month to date, rolling month, and sometimes even for the entire lifetime of the campaign. Moreover, advertisers also look at the geographic breakdown of these numbers, among other ways of slicing and dicing this data, such as what percentage of the total viewers or clickers of a particular ad are in the Asia Pacific region.

As is evident, there are four predominant dimension hierarchies: the account, campaign, and ad dimension; the time period; the type of interaction; and the user dimension. The last of these is used to report unique numbers, whereas the other three are the reporting dimensions. The user dimension is also used to create aggregated geographic profiles for the viewers and clickers of ads. All this information in totality allows the advertisers to tune their campaigns to improve their effectiveness on any given ad

network. Aside from the multidimensional nature of this set of pipelines, the volumes of data processed and the rate at which this data is growing on a daily basis make this difficult to scale without a technology such as Hadoop for large ad networks. As of this writing, for example, the ad log volume that is processed for ad performance numbers at Facebook is approximately 1 TB per day of (uncompressed) logs. This volume has seen a 30-fold increase since January 2008, when the volumes were in the range of 30 GB per day. Hadoop's ability to scale with hardware has been a major factor behind the ability of these pipelines to keep up with this data growth with only minor tweaking of job configurations. Typically, these configuration changes involve increasing the number of reducers for the Hadoop jobs that are processing the intensive portions of these pipelines. The largest of these stages currently runs with 400 reducers (an increase of eight times from the 50 reducers that were being used in January 2008).

Ad hoc analysis and product feedback

Apart from regular reports, another primary use case for a data warehousing solution is to support ad hoc analysis and product feedback solutions. Any typical website, for example, makes product changes, and product managers or engineers often need to understand the impact of a new feature, based on user engagement as well as on the click-through rate on that feature. The product team may even wish to perform a deeper analysis on the impact of the change based on various regions and countries, such as whether this change increases the click-through rate of the users in the US or whether it reduces the engagement of users in India. A lot of this type of analysis could be done with Hadoop by using Hive and regular SQL. The measurement of click-through rate can be easily expressed as a join of the impressions and clicks for the particular link related to the feature. This information can be joined with geographic information to compute the effect of product changes on different regions. Subsequently, one can compute average click-through rate for different geographic regions by performing aggregations over them. All of these are easily expressible in Hive using a couple of SQL queries (that would, in turn, generate multiple Hadoop jobs). If only an estimate were required, the same queries can be run for a sample set of the users using the sampling functionality natively supported by Hive. Some of this analysis needs the use of custom map and reduce scripts in conjunction with the Hive SQL, and that is also easy to plug into a Hive query.

A good example of a more complex analysis is estimating the peak number of users logging into the site per minute for the entire past year. This would involve sampling page view logs (because the total page view data for a popular website is huge), grouping it by time, and then finding the number of new users at different time points via a custom reduce script. This is a good example where both SQL and MapReduce are required for solving the end user problem, and something that is possible to achieve easily with Hive.

Data analysis

Hive and Hadoop are easy to use for training and scoring for data analysis applications. These data analysis applications can span multiple domains, such as popular websites, bioinformatics companies, and oil exploration companies. A typical example of such an application in the online ad network industry would be the prediction of what features of an ad make it more likely to be noticed by the user. The training phase typically would involve identifying the response metric and the predictive features. In this case, a good metric to measure the effectiveness of an ad could be its click-through rate. Some interesting features of the ad could be the industry vertical that it belongs to, the content of the ad, the placement of the ad on the page, and so on. Hive is useful for assembling training data and then feeding the same into a data analysis engine (typically R or user programs written in MapReduce). In this particular case, different ad performance numbers and features can be structured as tables in Hive. One can easily sample this data (sampling is required, as R can handle only limited data volume) and perform the appropriate aggregations and joins using Hive queries to assemble a response table containing the most important ad features that determine the effectiveness of an advertisement. However, because sampling loses information, some of the more important data analysis applications use parallel implementations of popular data analysis kernels using the MapReduce framework.

Once the model has been trained, it may be deployed for scoring on a daily basis. The bulk of the data analysis tasks do not perform daily scoring, though. Many of them are ad hoc in nature and require one-time analysis that can be used as input into the product design process.

Hive

When we started using Hadoop, we very quickly became impressed by its scalability and availability. However, we were worried about widespread adoption, primarily because of the complexity involved in writing MapReduce programs in Java (as well as the cost of training users to write them). We were aware that a lot of engineers and analysts in the company understood SQL as a tool to query and analyze data, and that many of them were proficient in a number of scripting languages, such as PHP and Python. As a result, it was imperative for us to develop software that could bridge this gap between the languages that the users were proficient in and the languages required to program Hadoop.

It was also evident that a lot of our datasets were structured and could be easily partitioned. The natural consequence of these requirements was a system that could model data as tables and partitions and that could also provide a SQL-like language for query and analysis. Also essential was the ability to plug in customized MapReduce programs written in the programming language of the user's choice into the query. This system was called Hive. Hive is a data warehouse infrastructure built on top of Hadoop and

serves as the predominant tool that is used to query the data stored in Hadoop at Facebook. In the following sections, we describe this system in more detail.

Data organization

Data is organized consistently across all datasets and is compressed, partitioned, and sorted:

Compression
: Almost all datasets are stored as sequence files using the gzip codec. Older datasets are recompressed to use the bzip codec, which affords substantially more compression than gzip. Bzip is slower than gzip, but older data is accessed much less frequently, and this performance hit is well worth the savings in terms of disk space.

Partitioning
: Most datasets are partitioned by date. Individual partitions are loaded into Hive, which loads each partition into a separate HDFS directory. In most cases, this partitioning is based simply on datestamps associated with Scribe logfiles. However, in some cases, we scan data and collate them based on the timestamp available inside a log entry. Going forward, we are also going to be partitioning data on multiple attributes (for example, country and date).

Sorting
: Each partition within a table is often sorted (and hash-partitioned) by unique ID (if one is present). This has a few key advantages:

  - It is easy to run sampled queries on such datasets.
  - We can build indexes on sorted data.
  - Aggregates and joins involving unique IDs can be done very efficiently on such datasets.

Loading data into this long-term format is done by daily MapReduce jobs (and is distinct from the near real-time data import processes).

Query language

The Hive Query language is very SQL-like. It has traditional SQL constructs such as joins, group bys, where, select, from clauses, and from clause subqueries. It tries to convert SQL commands into a set of MapReduce jobs. Apart from the normal SQL clauses, it has a bunch of other extensions, such as the ability to specify custom Mapper and reducer scripts in the query itself; the ability to insert into multiple tables, partitions, HDFS, or local files while doing a single scan of the data; and the ability to run the query on data samples rather than the full dataset (this ability is fairly useful when testing queries). The Hive metastore stores the metadata for a table and provides this metadata to the Hive compiler for converting SQL commands to MapReduce jobs. Through partition pruning, map-side aggregations, and other features, the compiler tries to create plans that can optimize the runtime for the query.

Data pipelines using Hive

Additionally, the ability provided by Hive in terms of expressing data pipelines in SQL can and has provided the much-needed flexibility in putting these pipelines together in an easy and expedient manner. This is especially useful for organizations and products that are still evolving and growing. Many of the operations needed in processing data pipelines are the well-understood SQL operations such as join, group by, and distinct aggregations. With Hive's ability to convert SQL into a series of Hadoop MapReduce jobs, it becomes fairly easy to create and maintain these pipelines. We illustrate these facets of Hive in this section by using an example of a hypothetical ad network and showing how some typical aggregated reports needed by the advertisers can be computed using Hive. As an example, assuming that an online ad network stores information on ads in a table named `dim_ads` and stores all the impressions served to that ad in a table named `impression_logs` in Hive, with the latter table being partitioned by date, the daily impression numbers (both unique and total by campaign, which are numbers that ad networks routinely give to the advertisers) for 2008-12-01 are expressible as the following SQL in Hive:

```
SELECT a.campaign_id, count(1), count(DISTINCT b.user_id)
FROM dim_ads a JOIN impression_logs b ON(b.ad_id = a.ad_id)
WHERE b.dateid = '2008-12-01'
GROUP BY a.campaign_id;
```

This would also be the typical SQL statement that one could use in other RDBMSs, such as Oracle, DB2, and so on.

In order to compute the daily impression numbers by ad and account from the same joined data as earlier, Hive provides the ability to do multiple group bys simultaneously, as shown in the following query (which is SQL-like, but not strictly SQL):

```
FROM(
  SELECT a.ad_id, a.campaign_id, a.account_id, b.user_id
  FROM dim_ads a JOIN impression_logs b ON (b.ad_id = a.ad_id)
  WHERE b.dateid = '2008-12-01') x
INSERT OVERWRITE DIRECTORY 'results_gby_adid'
  SELECT x.ad_id, count(1), count(DISTINCT x.user_id) GROUP BY x.ad_id
INSERT OVERWRITE DIRECTORY 'results_gby_campaignid'
  SELECT x.campaign_id, count(1), count(DISTINCT x.user_id) GROUP BY x.campaign_id
INSERT OVERWRITE DIRECTORY 'results_gby_accountid'
  SELECT x.account_id, count(1), count(DISTINCT x.user_id) GROUP BY x.account_id;
```

In one of the optimizations that is being added to Hive, the query can be converted into a sequence of Hadoop MapReduce jobs that are able to scale with data skew. Essentially, the join is converted into one MapReduce job ,and the three group bys are converted into four MapReduce jobs, with the first one generating a partial aggregate on `unique_id`. This is especially useful because the distribution of `impression_logs` over `unique_id` is much more uniform compared to `ad_id` (in an ad network, it is typical for a few ads to dominate in that they are shown more uniformly to the users). As a result, computing the partial aggregation by `unique_id` allows the pipeline to distribute the work more uniformly to the reducers. The same template can be used to compute

performance numbers for different time periods by simply changing the date predicate in the query.

Computing the lifetime numbers can be more tricky, though, because using the strategy described previously, one would have to scan all the partitions of the `impression_logs` table. Therefore, in order to compute the lifetime numbers, a more viable strategy is to store the lifetime counts on a per `ad_id`, `unique_id` grouping every day in a partition of an intermediate table. The data in this table combined with the next day's `impression_logs` can be used to incrementally generate the lifetime ad performance numbers. As an example, in order to get the impression numbers for 2008-12-01, the intermediate table partition for 2008-11-30 is used. The Hive queries that can be used to achieve this are as follows:

```
INSERT OVERWRITE lifetime_partial_imps PARTITION(dateid='2008-12-01')
SELECT x.ad_id, x.user_id, sum(x.cnt)
FROM (
  SELECT a.ad_id, a.user_id, a.cnt
  FROM lifetime_partial_imps a
  WHERE a.dateid = '2008-11-30'
  UNION ALL
  SELECT b.ad_id, b.user_id, 1 as cnt
  FROM impression_log b
  WHERE b.dateid = '2008-12-01'
) x
GROUP BY x.ad_id, x.user_id;
```

This query computes the partial sums for 2008-12-01, which can be used for computing the 2008-12-01 numbers as well as the 2008-12-02 numbers (not shown here). The SQL is converted to a single Hadoop MapReduce job that essentially computes the group by statement on the combined stream of inputs. This SQL can be followed by the next Hive query, which computes the actual numbers for different groupings (similar to the one in the daily pipelines):

```
FROM(
  SELECT a.ad_id, a.campaign_id, a.account_id, b.user_id, b.cnt
     FROM dim_ads a JOIN lifetime_partial_imps b ON (b.ad_id = a.ad_id)
     WHERE b.dateid = '2008-12-01') x
INSERT OVERWRITE DIRECTORY 'results_gby_adid'
  SELECT x.ad_id, sum(x.cnt), count(DISTINCT x.user_id) GROUP BY x.ad_id
INSERT OVERWRITE DIRECTORY 'results_gby_campaignid'
  SELECT x.campaign_id, sum(x.cnt), count(DISTINCT x.user_id) GROUP BY x.campaign_id
INSERT OVERWRITE DIRECTORY 'results_gby_accountid'
  SELECT x.account_id, sum(x.cnt), count(DISTINCT x.user_id) GROUP BY x.account_id;
```

Hive and Hadoop are batch processing systems that cannot serve the computed data with the same latency as a standard RDBMS, such as Oracle or MySQL. Therefore, on many occasions, it is still useful to load the summaries generated through Hive and Hadoop to a more traditional RDBMS for serving this data to users through different business intelligence (BI) tools, or even though a web portal.

Problems and Future Work

Fair sharing

Hadoop clusters typically run a mix of daily production jobs that need to finish computation within a reasonable time frame as well as ad hoc jobs that may be of different priorities and sizes. In typical installations, these production jobs tend to run overnight, when interference from ad hoc jobs run by users is minimal. However, overlap between large ad hoc and production jobs is often unavoidable and, without adequate safeguards, can impact the latency of production jobs. ETL processing also contains several near real-time jobs that must be performed at hourly intervals (these include processes to copy Scribe data from NFS servers, as well as hourly summaries computed over some datasets). It also means that a single rogue job can bring down the entire cluster and put production processes at risk.

The fair-sharing Hadoop job scheduler, developed at Facebook and contributed back to Hadoop, provides a solution to many of these issues. It reserves guaranteed compute resources for specific pools of jobs while at the same time letting idle resources be used by everyone. It also prevents large jobs from hogging cluster resources by allocating compute resources in a fair manner across these pools. Memory can become one of the most contended resources in the cluster. We have made some changes to Hadoop so that if the job tracker is low on memory, Hadoop job submissions are throttled. This can allow the user processes to run with reasonable per-process memory limits, and it is possible to put in place some monitoring scripts in order to prevent MapReduce jobs from impacting HDFS daemons (due primarily to high memory consumption) running on the same node. Log directories are stored in separate disk partitions and cleaned regularly, and we think it can also be useful to put MapReduce intermediate storage in separate disk partitions as well.

Space management

Capacity management continues to be a big challenge. Utilization is increasing at a fast rate with the growth of data, and many growing companies with expanding datasets have the same pain. In many situations, much of this data is temporary in nature. In such cases, one can use retention settings in Hive and also recompress older data in bzip format to save on space. Although configurations are largely symmetrical from a disk storage point of view, adding a separate tier of high-storage-density machines to hold older data may prove beneficial. This will make it cheaper to store archival data in Hadoop. However, access to such data should be transparent. We are currently working on a data archival layer to make this possible and to unify all the aspects of dealing with older data.

Scribe-HDFS integration

Currently, Scribe writes to a handful of NFS filers from where data is picked up and delivered to HDFS by custom copier jobs, as described earlier. We are working on making Scribe write directly to another HDFS instance. This will make it very easy to scale and administer Scribe. Due to the high uptime requirements for Scribe, its target HDFS instance is likely to be different from the production HDFS instance (so that it is isolated from any load/downtime issues due to user jobs).

Improvements to Hive

Hive is still under active development. We are working on a number of key features, such as support for order by and having clause supports, more aggregate functions, more built-in functions, a datetime data type, and so on. At the same time, a number of performance optimizations are being worked on, such as predicate pushdown and common subexpression elimination. On the integration side, JDBC and ODBC drivers are being developed in order to integrate with OLAP and BI tools. With all these optimizations, we hope that we can unlock the power of MapReduce and Hadoop, and bring it closer to nonengineering communities as well as within Facebook. For more information on this project, please visit *http://hadoop.apache.org/hive/*.

—Joydeep Sen Sarma and Ashish Thusoo

Nutch Search Engine

Nutch is a framework for building scalable Internet crawlers and search engines. It's an Apache Software Foundation project and a subproject of Lucene, and it's available under the Apache 2.0 license.

We won't go deeply into the anatomy of a web crawler as such; instead, the purpose of this case study is to show how Hadoop can be used to implement various complex processing tasks typical for a search engine. Interested readers can find plenty of Nutch-specific information on the official site of the project (*http://lucene.apache.org/nutch*). Suffice it to say that in order to create and maintain a search engine, one needs the following subsystems:

Database of pages
: This database keeps track of all pages known to the crawler and their status, such as the last time it visited the page, its fetching status, refresh interval, content checksum, etc. In Nutch terminology, this database is called *CrawlDb*.

List of pages to fetch
: As crawlers periodically refresh their view of the Web, they download new pages (previously unseen) or refresh pages that they think have already expired. Nutch calls such a list of candidate pages prepared for fetching a *fetchlist*.

Raw page data
: Page content is downloaded from remote sites and stored locally in the original uninterpreted format, as a byte array. This data is called the *page content* in Nutch.

Parsed page data
: Page content is then parsed using a suitable parser. Nutch provides parsers for documents in many popular formats, such as HTML, PDF, Open Office and Microsoft Office, RSS, and others.

Link graph database
: This database is necessary to compute link-based page ranking scores, such as PageRank. For each URL known to Nutch, it contains a list of other URLs pointing to it and their associated anchor text (from HTML `<a href="..">anchor text</a> elements`). This database is called *LinkDb*.

Full-text search index
: This is a classical inverted index, built from the collected page metadata and from the extracted plain-text content. It is implemented using the excellent Lucene library.

We briefly mentioned before that Hadoop began its life as a component in Nutch, intended to improve its scalability and to address clear performance bottlenecks caused by a centralized data processing model. Nutch was also the first public proof-of-concept application ported to the framework that would later become Hadoop, and the effort required to port Nutch algorithms and data structures to Hadoop proved to be surprisingly small. This probably encouraged the following development of Hadoop as a separate subproject with the aim of providing a reusable framework for applications other than Nutch.

Currently, nearly all Nutch tools process data by running one or more MapReduce jobs.

Data Structures

There are several major data structures maintained in Nutch, and they all make use of Hadoop I/O classes and formats. Depending on the purpose of the data and the way it's accessed once it's created, the data is kept using either Hadoop map files or sequence files.

Since the data is produced and processed by MapReduce jobs, which in turn run several map and reduce tasks, its on-disk layout corresponds to the common Hadoop output formats, that is, `MapFileOutputFormat` and `SequenceFileOutputFormat`. So to be precise, we should say that data is kept in several partial map files or sequence files, with as many parts as there were reduce tasks in the job that created the data. For simplicity, we omit this distinction in the following sections.

CrawlDb

CrawlDb stores the current status of each URL as a map file of `<url, CrawlDatum>`, where keys use `Text` and values use a Nutch-specific `CrawlDatum` class (which implements the `Writable` interface).

In order to provide quick random access to the records (sometimes useful for diagnostic reasons, when users want to examine individual records in CrawlDb), this data is stored in map files and not in sequence files.

CrawlDb is initially created using the Injector tool, which simply converts a plain-text representation of the initial list of URLs (called the seed list) to a map file in the format described earlier. Subsequently, it is updated with the information from the fetched and parsed pages—more on that shortly.

LinkDb

This database stores the incoming link information for every URL known to Nutch. It is a map file of `<url, Inlinks>`, where `Inlinks` is a list of URL and anchor text data. It's worth noting that this information is not immediately available during page collection, but the reverse information is available, namely that of outgoing links from a page. The process of inverting this relationship is implemented as a MapReduce job, described shortly.

Segments

Segments in Nutch parlance correspond to fetching and parsing a batch of URLs. Figure 16-5 presents how segments are created and processed.

A segment (which is really a directory in a filesystem) contains the following parts (which are simply subdirectories containing `MapFileOutputFormat` or `SequenceFileOutputFormat` data):

content
: Contains the raw data of downloaded pages as a map file of `<url, Content>`. Nutch uses a map file here because it needs fast random access in order to present a cached view of a page.

crawl_generate
: Contains the list of URLs to be fetched, together with their current status retrieved from CrawlDb, as a sequence file of `<url, CrawlDatum>`. This data uses the sequence file format, first because it's processed sequentially, and second because we couldn't satisfy the map file invariants of sorted keys. We need to spread URLs that belong to the same host as far apart as possible to minimize the load per target host, and this means that records are sorted more or less randomly.

crawl_fetch
: Contains status reports from the fetching, that is, whether it was successful, what was the response code, etc. This is stored in a map file of `<url, CrawlDatum>`.

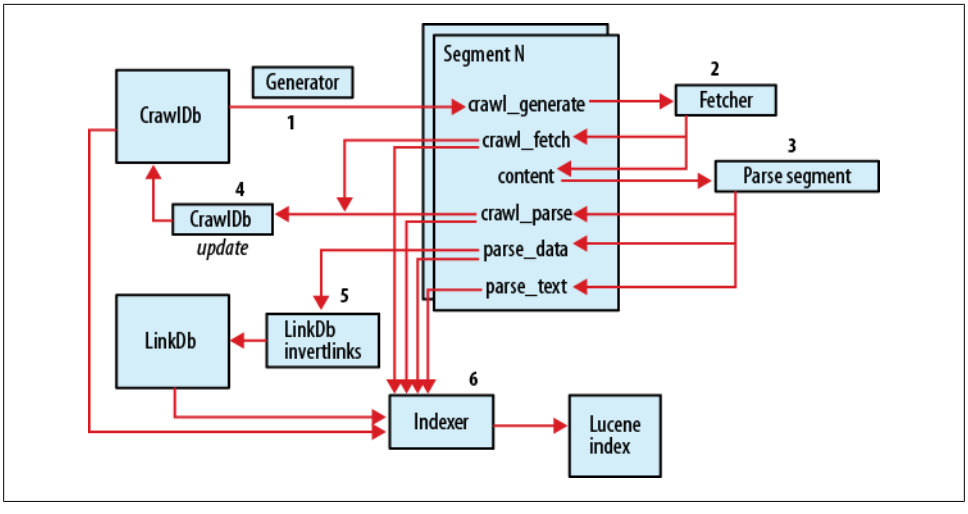


Figure 16-5. Segments

crawl_parse
: The list of outlinks for each successfully fetched and parsed page is stored here so that Nutch can expand its crawling frontier by learning new URLs.

parse_data
: Metadata collected during parsing; this includes, among others, the list of outgoing links (outlinks) for a page. This information is crucial later on to build an inverted graph (of incoming links, or inlinks).

parse_text
: Plain-text version of the page, suitable for indexing in Lucene. These are stored as a map file of `<url, ParseText>` so that Nutch can access them quickly when building summaries (snippets) to display the list of search results.

New segments are created from CrawlDb when the Generator tool is run (labeled as 1 in Figure 16-5), and initially contain just a list of URLs to fetch (the *crawl_generate* subdirectory). As this list is processed in several steps, the segment collects output data from the processing tools in a set of subdirectories.

For example, the *content* part is populated by a tool called Fetcher, which downloads raw data from URLs on the fetchlist (2). This tool also saves the status information in *crawl_fetch* so that this data can be used later on for updating the status of the page in CrawlDb.

The remaining parts of the segment are populated by the Parse segment tool (3), which reads the content section, selects appropriate content parser based on the declared (or detected) MIME type, and saves the results of parsing in three parts: *crawl_parse*, *parse_data*, and *parse_text*. This data is then used to update the CrawlDb with new information (4) and to create the LinkDb (5).

Segments are kept around until all pages present in them are expired. Nutch applies a configurable maximum time limit, after which a page is forcibly selected for refetching; this helps the operator phase out all segments older than this limit (because she can be sure that by that time, all pages in this segment would have been refetched).

Segment data is used to create Lucene indexes (primarily the *parse_text* and *parse_data* parts; 6 in the figure), but it also provides a data storage mechanism for quick retrieval of plain-text and raw content data. The former is needed so that Nutch can generate snippets (fragments of document text best matching a query); the latter provides the ability to present a "cached view" of the page. In both cases, this data is accessed directly from map files in response to requests for snippet generation or for cached content. In practice, even for large collections the performance of accessing data directly from map files is quite sufficient.

Selected Examples of Hadoop Data Processing in Nutch

The following sections present relevant details of some Nutch tools to illustrate how the MapReduce paradigm is applied to a concrete data processing task in Nutch.

Link inversion

HTML pages collected during crawling contain HTML links, which may point either to the same page (internal links) or to other pages. HTML links are directed from source page to target page. See Figure 16-6.

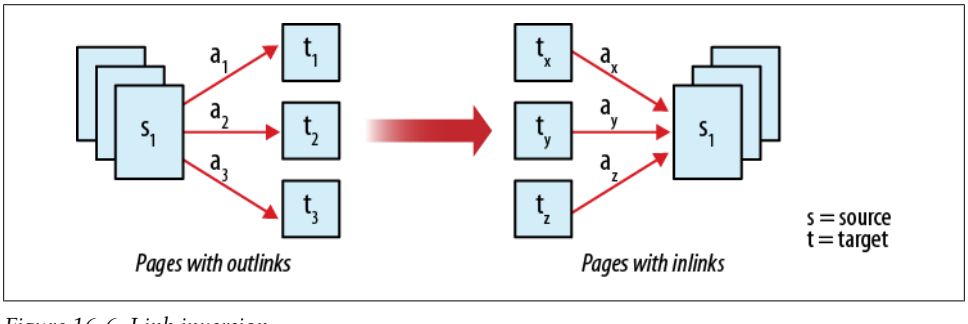


Figure 16-6. Link inversion

However, most algorithms for calculating a page's importance (or quality) need the opposite information, that is, what pages contain outlinks that point to the current page. This information is not readily available when crawling. Also, the indexing process benefits from taking into account the anchor text on inlinks so that this text may semantically enrich the text of the current page.

As mentioned earlier, Nutch collects the outlink information and then uses this data to build a LinkDb, which contains this reversed link data in the form of inlinks and anchor text.

This section presents a rough outline of the implementation of the LinkDb tool. Many details have been omitted (such as URL normalization and filtering) in order to present a clear picture of the process. What's left gives a classical example of why the MapReduce paradigm fits so well with the key data transformation processes required to run a search engine. Large search engines need to deal with massive web graph data (many pages with a lot of outlinks/inlinks), and the parallelism and fault tolerance offered by Hadoop make this possible. Additionally, it's easy to express the link inversion using the Map-sort-Reduce primitives, as illustrated next.

The snippet here presents the job initialization of the LinkDb tool:

```
JobConf job = new JobConf(configuration);
FileInputFormat.addInputPath(job, new Path(segmentPath, "parse_data"));
job.setInputFormat(SequenceFileInputFormat.class);
job.setMapperClass(LinkDb.class);
job.setReducerClass(LinkDb.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Inlinks.class);
job.setOutputFormat(MapFileOutputFormat.class);
FileOutputFormat.setOutputPath(job, newLinkDbPath);
```

As we can see, the source data for this job is the list of fetched URLs (keys) and the corresponding `ParseData` records that contain, among others, the outlink information for each page as an array of outlinks. An outlink contains both the target URL and the anchor text.

The output from the job is again a list of URLs (keys), but the values are instances of inlinks, which is simply a specialized set of inlinks that contain target URLs and anchor text.

Perhaps surprisingly, URLs are typically stored and processed as plain text and not as `java.net.URL` or `java.net.URI` instances. There are several reasons for this: URLs extracted from downloaded content usually need normalization (e.g., converting hostnames to lowercase, resolving relative paths), are often broken or invalid, or refer to unsupported protocols. Many normalization and filtering operations are better expressed as text patterns that span several parts of a URL. Also, for the purpose of link analysis, we may still want to process and count invalid URLs.

Let's take a closer look now at the `map()` and `reduce()` implementations. In this case, they are simple enough to be implemented in the body of the same class:

```
public void map(Text fromUrl, ParseData parseData,
    OutputCollector<Text, Inlinks> output, Reporter reporter) {
  ...
  Outlink[] outlinks = parseData.getOutlinks();
  Inlinks inlinks = new Inlinks();
  for (Outlink out : outlinks) {
    inlinks.clear(); // instance reuse to avoid excessive GC
    String toUrl = out.getToUrl();
    String anchor = out.getAnchor();
    inlinks.add(new Inlink(fromUrl, anchor));
    output.collect(new Text(toUrl), inlinks);
```

```
    }
  }
```

You can see from this listing that for each `Outlink`, our `map()` implementation produces a pair of `<toUrl, Inlinks>`, where `Inlinks` contains just a single `Inlink` containing `fromUrl` and the anchor text. The direction of the link has been inverted.

Subsequently, these one-element-long `Inlinks` are aggregated in the `reduce()` method:

```
public void reduce(Text toUrl, Iterator<Inlinks> values,
    OutputCollector<Text, Inlinks> output, Reporter reporter) {
  Inlinks result = new Inlinks();
  while (values.hasNext()) {
    result.add(values.next());
  }
  output.collect(toUrl, result);
}
```

From this code, it's obvious that we got exactly what we wanted—that is, a list of all `fromUrls` that point to our `toUrl`, together with their anchor text. The inversion process has been accomplished.

This data is then saved using the `MapFileOutputFormat` and becomes the new version of LinkDb.

Generation of fetchlists

Let's take a look now at a more complicated use case. Fetchlists are produced from the CrawlDb (which is a map file of `<url, crawlDatum>`, with the `crawlDatum` containing a status of this URL), and they contain URLs ready to be fetched, which are then processed by the Nutch Fetcher tool. Fetcher is itself a MapReduce application (described shortly). This means that the input data (partitioned in *N* parts) will be processed by *N* map tasks; the Fetcher tool enforces that `SequenceFileInputFormat` should not further split the data into more parts than there are already input partitions. Earlier, we mentioned briefly that fetchlists need to be generated in a special way so that the data in each part of the fetchlist (and consequently processed in each map task) meets certain requirements:

1. All URLs from the same host need to end up in the same partition. This is required so that Nutch can easily implement in-JVM host-level blocking to avoid overwhelming target hosts.
2. URLs from the same host should be as far apart as possible (i.e., well mixed with URLs from other hosts) in order to minimize the host-level blocking.
3. There should be no more than *x* URLs from any single host so that large sites with many URLs don't dominate smaller sites (and URLs from smaller sites still have a chance to be scheduled for fetching).
4. URLs with high scores should be preferred over URLs with low scores.

5. There should be, at most, *y* URLs in total in the fetchlist.
6. The number of output partitions should match the optimum number of fetching map tasks.

In this case, two MapReduce jobs are needed to satisfy all these requirements, as illustrated in Figure 16-7. Again, in the following listings, we are going to skip some details of these steps for the sake of brevity.

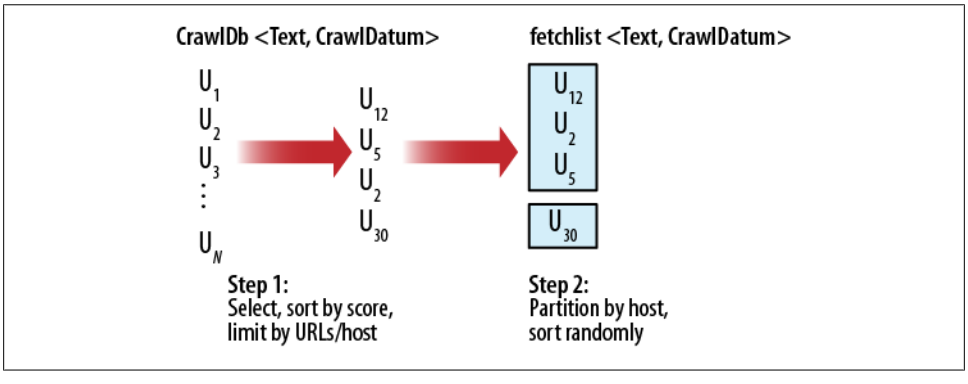


Figure 16-7. Generation of fetchlists

Step 1: Select, sort by score, limit by URL count per host. In this step, Nutch runs a MapReduce job to select URLs that are considered eligible for fetching and to sort them by their score (a floating-point value assigned to each URL, e.g., a PageRank score). The input data comes from CrawlDb, which is a map file of `<url, datum>`. The output from this job is a sequence file with `<score, <url, datum>>`, sorted in descending order by score.

First, let's look at the job setup:

```
FileInputFormat.addInputPath(job, crawlDbPath);
job.setInputFormat(SequenceFileInputFormat.class);
job.setMapperClass(Selector.class);
job.setPartitionerClass(Selector.class);
job.setReducerClass(Selector.class);
FileOutputFormat.setOutputPath(job, tempDir);
job.setOutputFormat(SequenceFileOutputFormat.class);
job.setOutputKeyClass(FloatWritable.class);
job.setOutputKeyComparatorClass(DecreasingFloatComparator.class);
job.setOutputValueClass(SelectorEntry.class);
```

The `Selector` class implements three functions: mapper, reducer, and partitioner. The last function is especially interesting: `Selector` uses a custom `Partitioner` to assign URLs from the same host to the same reduce task so that we can satisfy criteria 3–5 from the previous list. If we didn't override the default partitioner, URLs from the same host would end up in different partitions of the output, and we wouldn't be able to track and limit the total counts, because MapReduce tasks don't communicate between themselves. As it is now, all URLs that belong to the same host will end up being

processed by the same reduce task, which means we can control how many URLs per host are selected.

It's easy to implement a custom partitioner so that data that needs to be processed in the same task ends up in the same partition. Let's take a look first at how the `Selector` class implements the `Partitioner` interface (which consists of a single method):

```
/** Partition by host. */
public int getPartition(FloatWritable key, Writable value, int numReduceTasks) {
  return hostPartitioner.getPartition(((SelectorEntry)value).url, key,
    numReduceTasks);
}
```

The method returns an integer number from `0` to `numReduceTasks - 1`. It simply replaces the key with the original URL from `SelectorEntry` to pass the URL (instead of the score) to an instance of `PartitionUrlByHost`, where the partition number is calculated:

```
/** Hash by hostname. */
public int getPartition(Text key, Writable value, int numReduceTasks) {
  String urlString = key.toString();
  URL url = null;
  try {
    url = new URL(urlString);
  } catch (MalformedURLException e) {
    LOG.warn("Malformed URL: '" + urlString + "'");
  }
  int hashCode = (url == null ? urlString : url.getHost()).hashCode();
  // make hosts wind up in different partitions on different runs
  hashCode ^= seed;

  return (hashCode & Integer.MAX_VALUE) % numReduceTasks;
}
```

As you can see from the code snippet, the partition number is a function of only the host part of the URL, which means that all URLs that belong to the same host will end up in the same partition.

The output from this job is sorted in decreasing order by score. Since there are many records in CrawlDb with the same score, we couldn't use `MapFileOutputFormat` because we would violate the map file's invariant of strict key ordering.

Observant readers will notice that we had to use something other than the original keys, but we still want to preserve the original key-value pairs. We use here a `SelectorEntry` class to pass the original key-value pairs to the next step of processing.

`Selector.reduce()` keeps track of the total number of URLs and the maximum number of URLs per host, and simply discards excessive records. Please note that the enforcement of the total count limit is necessarily approximate. We calculate the limit for the current task as the total limit divided by the number of reduce tasks. But we don't know for sure from within the task that it is going to get an equal share of URLs;

indeed, in most cases it doesn't, because of the uneven distribution of URLs among hosts. However, for Nutch this approximation is sufficient.

Step 2: Invert, partition by host, sort randomly. In the previous step, we ended up with a sequence file of `<score, selectorEntry>`. Now we have to produce a sequence file of `<url, datum>` and satisfy criteria 1, 2, and 6 just described. The input data for this step is the output data produced in step 1.

The following is a snippet showing the setup of this job:

```
FileInputFormat.addInputPath(job, tempDir);
job.setInputFormat(SequenceFileInputFormat.class);
job.setMapperClass(SelectorInverseMapper.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(SelectorEntry.class);
job.setPartitionerClass(PartitionUrlByHost.class);
job.setReducerClass(PartitionReducer.class);
job.setNumReduceTasks(numParts);
FileOutputFormat.setOutputPath(job, output);
job.setOutputFormat(SequenceFileOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(CrawlDatum.class);
job.setOutputKeyComparatorClass(HashComparator.class);
```

The `SelectorInverseMapper` class simply discards the current key (the score value), extracts the original URL and uses it as a key, and uses the `SelectorEntry` as the value. Careful readers may wonder why we don't go one step further, extracting also the original `CrawlDatum` and using it as the value—more on this shortly.

The final output from this job is a sequence file of `<Text, CrawlDatum>`, but our output from the map phase uses `<Text, SelectorEntry>`. We have to specify that we use different key-value classes for the map output, using the `setMapOutputKeyClass()` and `setMapOutputValueClass()` setters. Otherwise, Hadoop assumes that we use the same classes as declared for the reduce output (this conflict usually would cause a job to fail).

The output from the map phase is partitioned using the `PartitionUrlByHost` class so that it again assigns URLs from the same host to the same partition. This satisfies requirement 1.

Once the data is shuffled from map to reduce tasks, it's sorted by Hadoop according to the output key comparator, in this case the `HashComparator`. This class uses a simple hashing scheme to mix URLs in a way that is least likely to put URLs from the same host close to each other.

In order to meet requirement 6, we set the number of reduce tasks to the desired number of `Fetcher` map tasks (the `numParts` mentioned earlier), keeping in mind that each reduce partition will be used later on to create a single `Fetcher` map task.

The `PartitionReducer` class is responsible for the final step, that is, to convert `<url, selectorEntry>` to `<url, crawlDatum>`. A surprising side effect of using `HashComparator` is that several URLs may be hashed to the same hash value, and Hadoop will call

the `reduce()` method, passing only the first such key; all other keys considered equal will be discarded. Now it becomes clear why we had to preserve all URLs in `SelectorEntry` records, because now we can extract them from the iterated values. Here is the implementation of this method:

```
public void reduce(Text key, Iterator<SelectorEntry> values,
    OutputCollector<Text, CrawlDatum> output, Reporter reporter) throws IOException {
  // when using HashComparator, we get only one input key in case of hash collisions
  // so use only URLs extracted from values
  while (values.hasNext()) {
    SelectorEntry entry = values.next();
    output.collect(entry.url, entry.datum);
  }
}
```

Finally, the output from reduce tasks is stored as a `SequenceFileOutputFormat` in a *crawl_generate* subdirectory of a Nutch segment directory. This output satisfies all criteria from 1 to 6.

Fetcher: A multithreaded MapRunner in action

The Fetcher application in Nutch is responsible for downloading the page content from remote sites. As such, it is important that the process uses every opportunity for parallelism, in order to minimize the time it takes to crawl a fetchlist.

There is already one level of parallelism present in Fetcher; multiple parts of the input fetchlists are assigned to multiple map tasks. However, in practice this is not sufficient: sequential download of URLs, from different hosts (see the earlier section on `HashComparator`), would be a tremendous waste of time. For this reason, the Fetcher map tasks process this data using multiple worker threads.

Hadoop uses the `MapRunner` class to implement the sequential processing of input data records. The `Fetcher` class implements its own `MapRunner` that uses multiple threads to process input records in parallel.

Let's begin with the setup of the job:

```
job.setSpeculativeExecution(false);
FileInputFormat.addInputPath(job, "segment/crawl_generate");
job.setInputFormat(InputFormat.class);
job.setMapRunnerClass(Fetcher.class);
FileOutputFormat.setOutputPath(job, segment);
job.setOutputFormat(FetcherOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(NutchWritable.class);
```

First, we turn off speculative execution. We can't run several map tasks to download content from the same hosts, because it would violate the host-level load limits (such as the number of concurrent requests and the number of requests per second).

Next, we use a custom `InputFormat` implementation that prevents Hadoop from splitting partitions of input data into smaller chunks (splits), thus creating more map tasks

than there are input partitions. This again ensures that we control host-level access limits.

Output data is stored using a custom `OutputFormat` implementation, which uses data contained in `NutchWritable` values to create several output map files and sequence files. The `NutchWritable` class is a subclass of `GenericWritable`, able to pass instances of several different `Writable` classes declared in advance.

The Fetcher class implements the `MapRunner` interface, and we set this class as the job's `MapRunner` implementation. The relevant parts of the code are listed here:

```
public void run(RecordReader<Text, CrawlDatum> input,
    OutputCollector<Text, NutchWritable> output,
                Reporter reporter) throws IOException {
  int threadCount = getConf().getInt("fetcher.threads.fetch", 10);
  feeder = new QueueFeeder(input, fetchQueues, threadCount * 50);
  feeder.start();

  for (int i = 0; i < threadCount; i++) {       // spawn threads
    new FetcherThread(getConf()).start();
  }
  do {                                          // wait for threads to exit
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {}
    reportStatus(reporter);
  } while (activeThreads.get() > 0);
}
```

`Fetcher` reads many input records in advance, using the `QueueFeeder` thread that puts input records into a set of per-host queues. Then several `FetcherThread` instances are started, which consume items from per-host queues, while `QueueFeeder` keeps reading input data to keep the queues filled. Each `FetcherThread` consumes items from any nonempty queue.

In the meantime, the main thread of the map task spins around, waiting for all threads to finish their job. Periodically, it reports the status to the framework to ensure that Hadoop doesn't consider this task to be dead and kill it. Once all items are processed, the loop is finished and the control is returned to Hadoop, which considers this map task to be completed.

Indexer: Using custom OutputFormat

This is an example of a MapReduce application that doesn't produce sequence file or map file output; instead, the output from this application is a Lucene index. Again, as MapReduce applications may consist of several reduce tasks, the output from this application may consist of several partial Lucene indexes.

The Nutch Indexer tool uses information from CrawlDb, LinkDb, and Nutch segments (fetch status, parsing status, page metadata, and plain-text data), so the job setup section involves adding several input paths:

```
FileInputFormat.addInputPath(job, crawlDbPath);
FileInputFormat.addInputPath(job, linkDbPath);
// add segment data
FileInputFormat.addInputPath(job, "segment/crawl_fetch");
FileInputFormat.addInputPath(job, "segment/crawl_parse");
FileInputFormat.addInputPath(job, "segment/parse_data");
FileInputFormat.addInputPath(job, "segment/parse_text");
job.setInputFormat(SequenceFileInputFormat.class);
job.setMapperClass(Indexer.class);
job.setReducerClass(Indexer.class);
FileOutputFormat.setOutputPath(job, indexDir);
job.setOutputFormat(OutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LuceneDocumentWrapper.class);
```

All corresponding records for a URL dispersed among these input locations need to be combined to create Lucene documents for addition to the index.

The `Mapper` implementation in `Indexer` simply wraps input data, whatever its source and implementation class, in a `NutchWritable` so that the reduce phase may receive data from different sources, using different classes, and still be able to consistently declare a single output value class (as `NutchWritable`) from both map and reduce steps.

The `Reducer` implementation iterates over all values that fall under the same key (URL), unwraps the data (fetch `CrawlDatum`, CrawlDb `CrawlDatum`, LinkDb `Inlinks`, `ParseData`, and `ParseText`) and, using this information, builds a Lucene document, which is then wrapped in a `Writable LuceneDocumentWrapper` and collected. In addition to all textual content (coming either from the plain-text data or from metadata), this document also contains PageRank-like score information (obtained from CrawlDb data). Nutch uses this score to set the boost value of the Lucene document.

The `OutputFormat` implementation is the most interesting part of this tool:

```
public static class OutputFormat extends
    FileOutputFormat<WritableComparable, LuceneDocumentWrapper> {

  public RecordWriter<WritableComparable, LuceneDocumentWrapper>
      getRecordWriter(final FileSystem fs, JobConf job,
                      String name, final Progressable progress) throws IOException {
    final Path out = new Path(FileOutputFormat.getOutputPath(job), name);
    final IndexWriter writer = new IndexWriter(out.toString(),
                      new NutchDocumentAnalyzer(job), true);

    return new RecordWriter<WritableComparable, LuceneDocumentWrapper>() {
      boolean closed;
      public void write(WritableComparable key, LuceneDocumentWrapper value)
          throws IOException {                  // unwrap & index doc
        Document doc = value.get();
        writer.addDocument(doc);
        progress.progress();
      }

      public void close(final Reporter reporter) throws IOException {
        // spawn a thread to give progress heartbeats
```

```
            Thread prog = new Thread() {
                public void run() {
                  while (!closed) {
                    try {
                      reporter.setStatus("closing");
                      Thread.sleep(1000);
                    } catch (InterruptedException e) { continue; }
                      catch (Throwable e) { return; }
                  }
                }
              };

            try {
              prog.start();
              // optimize & close index
              writer.optimize();
              writer.close();
            } finally {
              closed = true;
            }
          }
        };
    }
```

When an instance of `RecordWriter` is requested, the `OutputFormat` creates a new Lucene index by opening an `IndexWriter`. Then, for each new output record collected in the reduce method, it unwraps the Lucene document from the `LuceneDocumentWrapper` value and adds it to the index.

When a reduce task is finished, Hadoop will try to close the `RecordWriter`. In this case, the process of closing may take a long time because we would like to optimize the index before closing. During this time, Hadoop may conclude that the task is hung, since there are no progress updates, and it may attempt to kill it. For this reason, we first start a background thread to give reassuring progress updates, and then proceed to perform the index optimization. Once the optimization is completed, we stop the progress updater thread. The output index is now created, optimized, and closed, after which it is ready for use in a searcher application.

Summary

This short overview of Nutch necessarily omits many details, such as error handling, logging, URL filtering and normalization, dealing with redirects or other forms of "aliased" pages (such as mirrors), removing duplicate content, calculating PageRank scoring, etc. You can find this and much more information on the official page of the project and on the wiki (*http://wiki.apache.org/nutch*).

Today, Nutch is used by many organizations and individual users. Still, operating a search engine requires nontrivial investments in hardware, integration, customization, and the maintenance of the index, so in most cases, Nutch is used to build commercial vertical- or field-specific search engines.

Nutch is under active development, and the project closely follows new releases of Hadoop. As such, it will continue to be a practical example of a real-life application that uses Hadoop at its core, with excellent results.

—Andrzej Białecki

Log Processing at Rackspace

Rackspace Hosting has always provided managed systems for enterprises, and in that vein, Mailtrust became Rackspace's mail division in Fall 2007. Rackspace currently hosts email for over 1 million users and thousands of companies on hundreds of servers.

Requirements/The Problem

Transferring the mail generated by Rackspace customers through the system generates a considerable "paper" trail, in the form of around 150 GB per day of logs in various formats. It is extremely helpful to aggregate that data for growth planning purposes and to understand how customers use our applications, and the records are also a boon for troubleshooting problems in the system.

If an email fails to be delivered, or a customer is unable to log in, it is vital that our customer support team is able to find enough information about the problem to begin the debugging process. To make it possible to find that information quickly, we cannot leave the logs on the machines that generated them or in their original format. Instead, we use Hadoop to do a considerable amount of processing, with the end result being Lucene indexes that customer support can query.

Logs

Two of our highest-volume log formats are produced by the Postfix mail transfer agent and Microsoft Exchange Server. All mail that travels through our systems touches Postfix at some point, and the majority of messages travel through multiple Postfix servers. The Exchange environment is independent by necessity, but one class of Postfix machines acts as an added layer of protection and uses SMTP to transfer messages between mailboxes hosted in each environment.

The messages travel through many machines, but each server knows only enough about the destination of the mail to transfer it to the next responsible server. Thus, in order to build the complete history of a message, our log processing system needs to have a global view of the system. This is where Hadoop helps us immensely: as our system grows, so does the volume of logs. For our log processing logic to stay viable, we had to ensure that it would scale, and MapReduce was the perfect framework for that growth.

Brief History

Earlier versions of our log processing system were based on MySQL, but as we gained more and more logging machines, we reached the limits of what a single MySQL server could process. The database schema was already reasonably denormalized, which would have made it less difficult to shard, but MySQL's partitioning support was still very weak at that point in time. Rather than implementing our own sharding and processing solution around MySQL, we chose to use Hadoop.

Choosing Hadoop

As soon as you shard the data in a RDBMS system, you lose a lot of the advantages of SQL for performing analysis of your dataset. Hadoop gives us the ability to easily process all of our data in parallel using the same algorithms we would for smaller datasets.

Collection and Storage

Log collection

The servers generating the logs we process are distributed across multiple data centers, but we currently have a single Hadoop cluster, located in one of those data centers (see Figure 16-8). In order to aggregate the logs and place them into the cluster, we use the Unix syslog replacement syslog-ng and some simple scripts to control the creation of files in Hadoop.

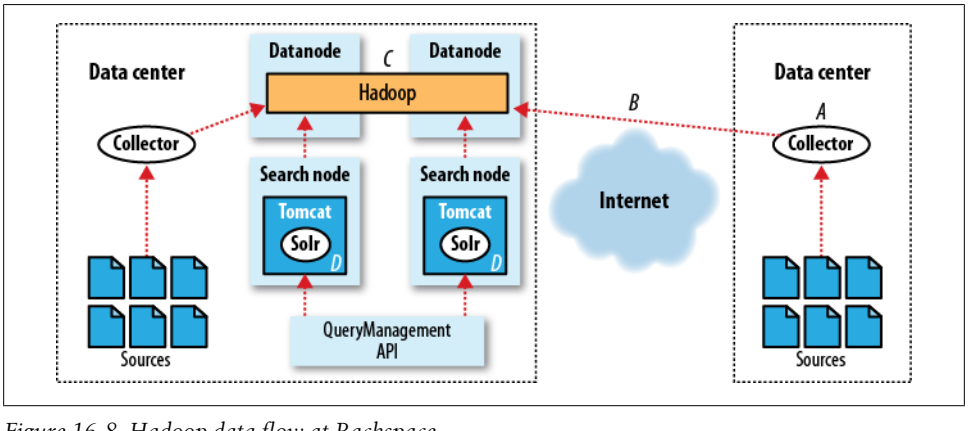


Figure 16-8. Hadoop data flow at Rackspace

Within a data center, syslog-ng is used to transfer logs from a *source* machine to a load-balanced set of *collector* machines. On the collectors, each type of log is aggregated into a single stream and lightly compressed with gzip (step A in Figure 16-8). From remote collectors, logs can be transferred through an SSH tunnel cross-data center to collectors that are local to the Hadoop cluster (step B).

Once the compressed log stream reaches a local collector, it can be written to Hadoop (step C). We currently use a simple Python script that buffers input data to disk and periodically pushes the data into the Hadoop cluster using the Hadoop command-line interface. The script copies the log buffers to input folders in Hadoop when they reach a multiple of the Hadoop block size or when enough time has passed.

This method of securely aggregating logs from different data centers was developed before SOCKet Secure (SOCKS) support was added to Hadoop via the `hadoop.rpc.socket.factory.class.default` parameter and `SocksSocketFactory` class. By using SOCKS support and the HDFS API directly from remote collectors, we could eliminate one disk write and a lot of complexity from the system. We plan to implement a replacement using these features in future development sprints.

Once the raw logs have been placed in Hadoop, they are ready for processing by our MapReduce jobs.

Log storage

Our Hadoop cluster currently contains 15 datanodes with commodity CPUs and three 500 GB disks each. We use a default replication factor of three for files that need to survive for our archive period of six months and a factor of two for anything else.

The Hadoop namenode uses hardware identical to the datanodes. To provide reasonably high availability, we use two secondary namenodes and a virtual IP that can easily be pointed at any of the three machines with snapshots of the HDFS. This means that in a failover situation, there is potential for us to lose up to 30 minutes of data, depending on the ages of the snapshots on the secondary namenodes. This is acceptable for our log processing application, but other Hadoop applications may require lossless failover by using shared storage for the namenode's image.

MapReduce for Logs

Processing

In distributed systems, the sad truth of unique identifiers is that they are rarely truly unique. All email messages have a (supposedly) unique identifier called a *message-id* that is generated by the host where they originated, but a bad client could easily send out duplicates. In addition, since the designers of Postfix could not trust the message-id to uniquely identify the message, they were forced to come up with a separate ID called a *queue-id*, which is guaranteed to be unique only for the lifetime of the message on a local machine.

Although the message-id tends to be the definitive identifier for a message, in Postfix logs, it is necessary to use queue-ids to find the message-id. Looking at the second line in Example 16-1 (which is formatted to better fit the page), you will see the hex string `1DBD21B48AE`, which is the queue-id of the message that the log line refers to. Because

information about a message (including its message-id) is output as separate lines when it is collected (potentially hours apart), it is necessary for our parsing code to keep state about messages.

Example 16-1. Postfix log lines

```
Nov 12 17:36:54 gate8.gate.sat.mlsrvr.com postfix/smtpd[2552]: connect from hostname
Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com postfix/qmgr[9489]: 1DBD21B48AE:
from=<mapreduce@rackspace.com>, size=5950, nrcpt=1 (queue active)
Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com postfix/smtpd[28085]: disconnect from
hostname
Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com postfix/smtpd[22593]: too many errors
after DATA from hostname
Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com postfix/smtpd[22593]: disconnect from
hostname
Nov 12 17:36:54 gate10.gate.sat.mlsrvr.com postfix/smtpd[10311]: connect from
hostname
Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com postfix/smtp[28107]: D42001B48B5:
to=<mapreduce@rackspace.com>, relay=hostname[ip], delay=0.32, delays=0.28/0/0/0.04,
dsn=2.0.0, status=sent (250 2.0.0 Ok: queued as 1DBD21B48AE)
Nov 12 17:36:54 gate20.gate.sat.mlsrvr.com postfix/smtpd[27168]: disconnect from
hostname
Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com postfix/qmgr[1209]: 645965A0224: removed
Nov 12 17:36:54 gate2.gate.sat.mlsrvr.com postfix/smtp[15928]: 732196384ED: to=<m
apreduce@rackspace.com>, relay=hostname[ip], conn_use=2, delay=0.69, delays=0.04/
0.44/0.04/0.17, dsn=2.0.0, status=sent (250 2.0.0 Ok: queued as 02E1544C005)
Nov 12 17:36:54 gate2.gate.sat.mlsrvr.com postfix/qmgr[13764]: 732196384ED: removed
Nov 12 17:36:54 gate1.gate.sat.mlsrvr.com postfix/smtpd[26394]: NOQUEUE: reject: RCP
T from hostname 554 5.7.1 <mapreduce@rackspace.com>: Client host rejected: The
sender's mail server is blocked; from=<mapreduce@rackspace.com> to=<mapred
uce@rackspace.com> proto=ESMTP helo=<mapreduce@rackspace.com>
```

From a MapReduce perspective, each line of the log is a single key-value pair. In phase 1, we need to map all lines with a single queue-id key together, and then reduce them to determine whether the log message values indicate that the queue-id is complete.

Similarly, once we have a completed the queue-id phase for a message, we need to group by the message-id in phase 2. We map each completed queue-id with its message-id as key and a list of its log lines as the value. In the reduce, we determine whether all of the queue-ids for the message-id indicate that the message left our system.

Together, the two phases of the mail log MapReduce job and their `InputFormat` and `OutputFormat` form a type of *staged event-driven architecture* (SEDA). In SEDA, an application is broken up into multiple "stages," which are separated by queues. In a Hadoop context, a queue could be either an input folder in HDFS that a MapReduce job consumes from or the implicit queue that MapReduce forms between the map and reduce steps.

In Figure 16-9, the arrows between stages represent the queues, with a dashed arrow being the implicit MapReduce queue. Each stage can send a key-value pair (SEDA calls them events or messages) to another stage via these queues.

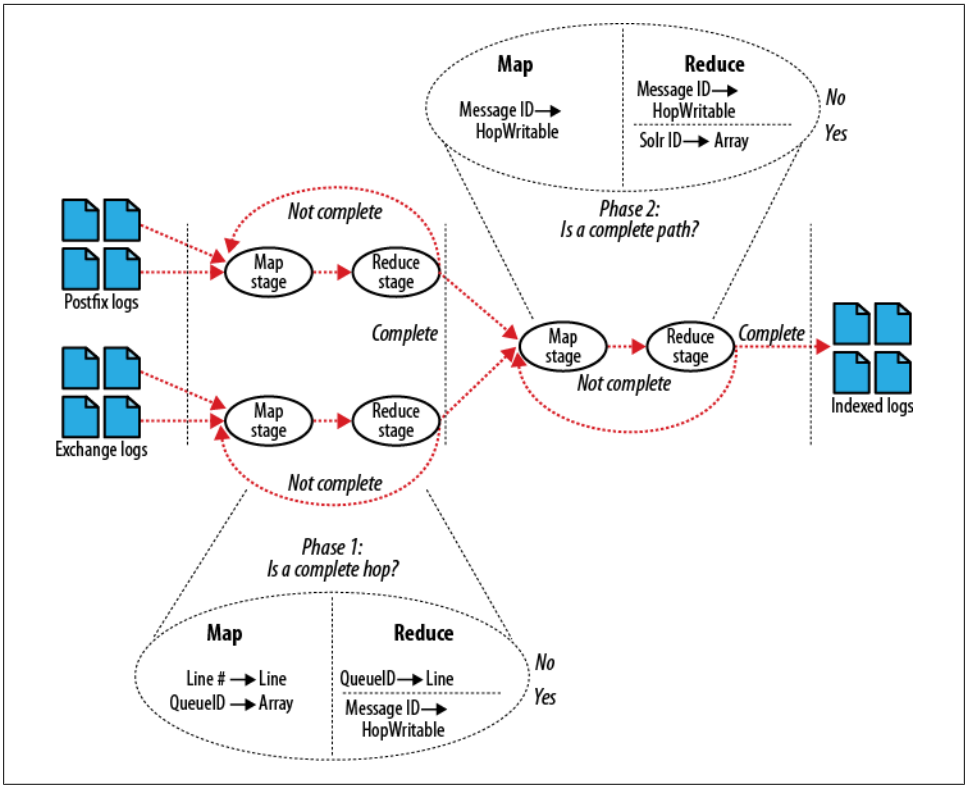


Figure 16-9. MapReduce chain

Phase 1: Map. During the first phase of our Mail log processing job, the inputs to the map stage are either a line number key and log message value or a queue-id key to an array of log-message values. The first type of input is generated when we process a raw logfile from the queue of input files, and the second type is an intermediate format that represents the state of a queue-id we have already attempted to process but that was requeued because it was incomplete.

In order to accomplish this dual input, we implemented a Hadoop `InputFormat` that delegates the work to an underlying `SequenceFileRecordReader` or `LineRecordReader`, depending on the file extension of the input `FileSplit`. The two input formats come from different input folders (queues) in HDFS.

Phase 1: Reduce. During this phase, the reduce stage determines whether the queue-id has enough lines to be considered completed. If the queue-id is completed, we output the message-id as key and a `HopWritable` object as value. Otherwise, the queue-id is set as the key, and the array of log lines is requeued to be Mapped with the next set of raw logs. This will continue until we complete the queue-id or until it times out.

The `HopWritable` object is a POJO that implements Hadoop's `Writable` interface. It completely describes a message from the viewpoint of a single server, including the sending address and IP, attempts to deliver the message to other servers, and typical message header information.

This split output is accomplished with an `OutputFormat` implementation that is somewhat symmetrical with our dual `InputFormat`. Our `MultiSequenceFileOutputFormat` was implemented before the Hadoop API added a `MultipleSequenceFileOutputFormat` in release0.17.0, but fulfills the same type of goal: we needed our reduce output pairs to go to different files, depending on characteristics of their keys.

Phase 2: Map. In the next stage of the mail log processing job, the input is a message-id key, with a `HopWritable` value from the previous phase. This stage does not contain any logic; instead, it simply combines the inputs from the first phase using the standard `SequenceFileInputFormat` and `IdentityMapper`.

Phase 2: Reduce. In the final reduce stage, we want to see whether all of the `HopWritables` we have collected for the message-id represent a complete message path through our system. A message path is essentially a directed graph (which is typically acyclic, but it may contain loops if servers are misconfigured). In this graph, a vertex is a server, which can be labeled with multiple queue-ids, and attempts to deliver the message from one server to another are edges. For this processing, we use the JGraphT graph library.

For output, we again use the `MultiSequenceFileOutputFormat`. If the reducer decides that all of the queue-ids for a message-id create a complete message path, then the message is serialized and queued for the `SolrOutputFormat`. Otherwise, the `HopWritables` for the message are queued for the phase 2: map stage to be reprocessed with the next batch of queue-ids.

The `SolrOutputFormat` contains an embedded Apache Solr instance—in the fashion that was originally recommended by the Solr wiki—to generate an index on local disk. Closing the `OutputFormat` then involves compressing the disk index to the final destination for the output file. This approach has a few advantages over using Solr's HTTP interface or using Lucene directly:

- We can enforce a Solr schema.
- Map and reduce remain idempotent.
- Indexing load is removed from the search nodes.

We currently use the default `HashPartitioner` class to decide which reduce task will receive particular keys, which means that the keys are semirandomly distributed. In a future iteration of the system, we'd like to implement a new `Partitioner` to split by the sending address instead (our most common search term). Once the indexes are split by sender, we can use the hash of the address to determine where to merge or query for an index, and our search API will need to communicate only with the relevant nodes.

Merging for near-term search

After a set of MapReduce phases have completed, a different set of machines are notified of the new indexes and can pull them for merging. These search nodes are running Apache Tomcat and Solr instances to host completed indexes, along with a service to pull and merge the indexes to local disk (step D in Figure 16-8).

Each compressed file from `SolrOutputFormat` is a complete Lucene index, and Lucene provides the `IndexWriter.addIndexes()` methods for quickly merging multiple indexes. Our `MergeAgent` service decompresses each new index into a Lucene `RAMDirectory` or `FSDirectory` (depending on size), merges them to local disk, and sends a `<commit/>` request to the Solr instance hosting the index to make the changed index visible to queries.

Sharding. The Query/Management API is a thin layer of PHP code that handles sharding the output indexes across all of the search nodes. We use a simple implementation of consistent hashing to decide which search nodes are responsible for each index file. Currently, indexes are sharded by their creation time and then by their hashed filename, but we plan to replace the filename hash with a sending address hash at some point in the future (see "Phase 2: reduce").

Because HDFS already handles replication of the Lucene indexes, there is no need to keep multiple copies available in Solr. Instead, in a failover situation, the search node is completely removed, and other nodes become responsible for merging the indexes.

Search results. With this system, we've achieved a 15-minute turnaround time from log generation to availability of a search result for our Customer Support team.

Our search API supports the full Lucene query syntax, so we commonly see complex queries like:

```
sender:"mapreduce@rackspace.com" -recipient:"hadoop@rackspace.com"
 recipient:"@rackspace.com" short-status:deferred timestamp:[1228140900 TO 2145916799]
```

Each result returned by a query is a complete serialized message path that indicates whether individual servers and recipients received the message. We currently display the path as a 2D graph (Figure 16-10) that the user can interact with to expand points of interest, but there is a lot of room for improvement in the visualization of this data.

Archiving for analysis

In addition to providing short-term search for Customer Support, we are also interested in performing analyses of our log data.

Every night, we run a series of MapReduce jobs with the day's indexes as input. We implemented a `SolrInputFormat` that can pull and decompress an index, and emit each document as a key-value pair. With this `InputFormat`, we can iterate over all message paths for a day and answer almost any question about our mail system, including:

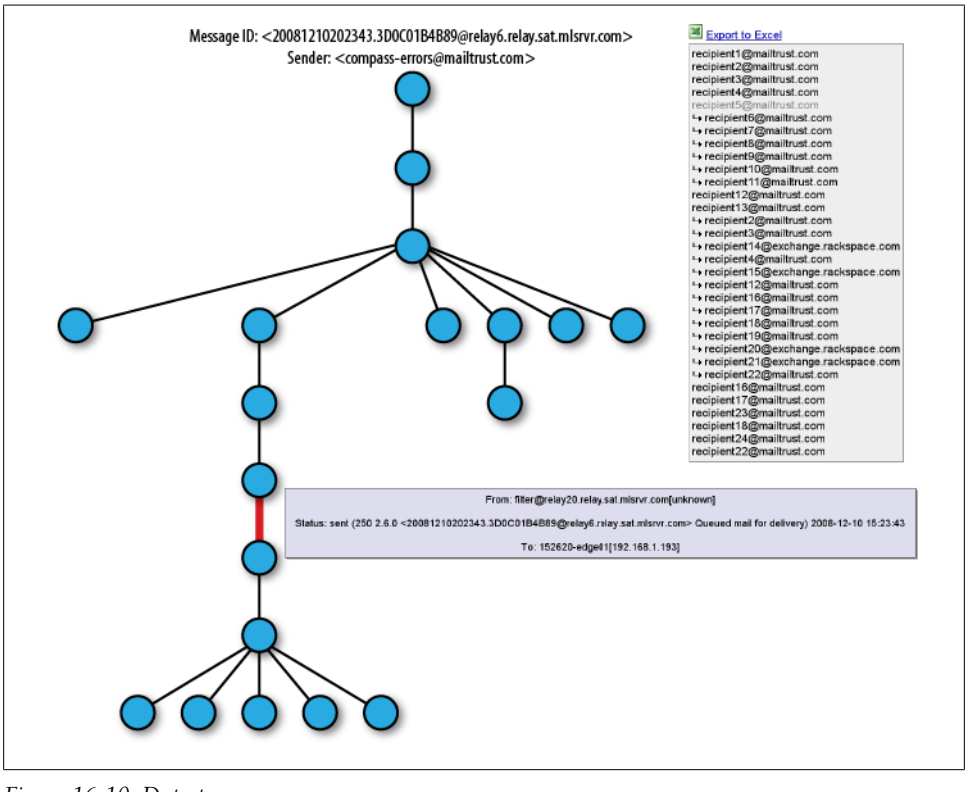


Figure 16-10. Data tree

- Per-domain data (viruses, spam, connections, recipients)
- Most effective spam rules
- Load generated by specific users
- Reasons for message bounces
- Geographical sources of connections
- Average latency between specific machines

Because we have months of compressed indexes archived in Hadoop, we are also able to retrospectively answer questions that our nightly log summaries leave out. For instance, we recently wanted to determine the top sending IP addresses per month, which we accomplished with a simple one-off MapReduce job.

—Stu Hood

Cascading

Cascading is an open source Java library and API that provides an abstraction layer for MapReduce. It allows developers to build complex, mission-critical data processing applications that run on Hadoop clusters.

The Cascading project began in the summer of 2007. Its first public release, version 0.1, launched in January 2008. Version 1.0 was released in January 2009. Binaries, source code, and add-on modules can be downloaded from the project website, *http://www.cascading.org/*.

Map and reduce operations offer powerful primitives. However, they tend to be at the wrong level of granularity for creating sophisticated, highly composable code that can be shared among different developers. Moreover, many developers find it difficult to "think" in terms of MapReduce when faced with real-world problems.

To address the first issue, Cascading substitutes the keys and values used in MapReduce with simple field names and a data tuple model, where a tuple is simply a list of values. For the second issue, Cascading departs from map and reduce operations directly by introducing higher-level abstractions as alternatives: `Function`s, `Filter`s, `Aggregator`s, and `Buffer`s.

Other alternatives began to emerge at about the same time as the project's initial public release, but Cascading was designed to complement them. Consider that most of these alternative frameworks impose pre- and post-conditions, or other expectations.

For example, in several other MapReduce tools, you must preformat, filter, or import your data into HDFS prior to running the application. That step of preparing the data must be performed outside of the programming abstraction. In contrast, Cascading provides the means to prepare and manage your data as integral parts of the programming abstraction.

This case study begins with an introduction to the main concepts of Cascading, then finishes with an overview of how ShareThis uses Cascading in its infrastructure.

See the Cascading User Guide on the project website for a more in-depth presentation of the Cascading processing model.

Fields, Tuples, and Pipes

The MapReduce model uses keys and values to link input data to the map function, the map function to the reduce function, and the reduce function to the output data.

But as we know, real-world Hadoop applications usually consist of more than one MapReduce job chained together. Consider the canonical word count example implemented in MapReduce. If you needed to sort the numeric counts in descending order, which is not an unlikely requirement, it would need to be done in a second MapReduce job.

So, in the abstract, keys and values not only bind map to reduce, but reduce to the next map, and then to the next reduce, and so on (Figure 16-11). That is, key-value pairs are sourced from input files and stream through chains of map and reduce operations, and finally rest in an output file. When you implement enough of these chained MapReduce applications, you start to see a well-defined set of key-value manipulations used over and over again to modify the key-value data stream.

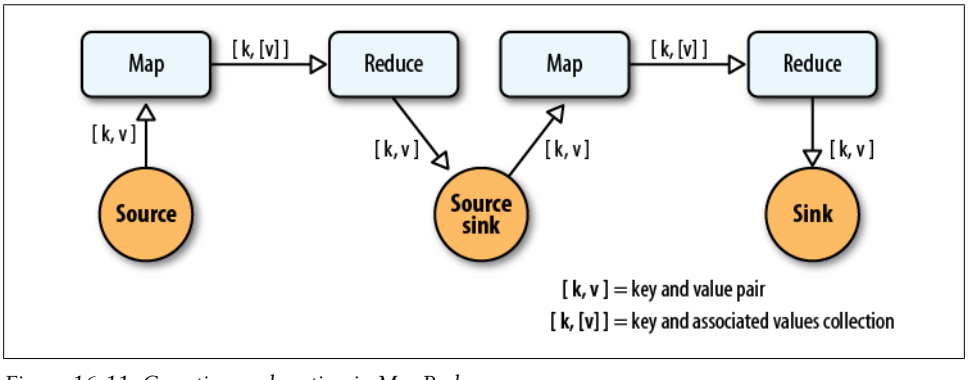


Figure 16-11. Counting and sorting in MapReduce

Cascading simplifies this by abstracting away keys and values and replacing them with tuples that have corresponding field names, similar in concept to tables and column names in a relational database. And during processing, streams of these fields and tuples are then manipulated as they pass through user-defined operations linked together by pipes (Figure 16-12).

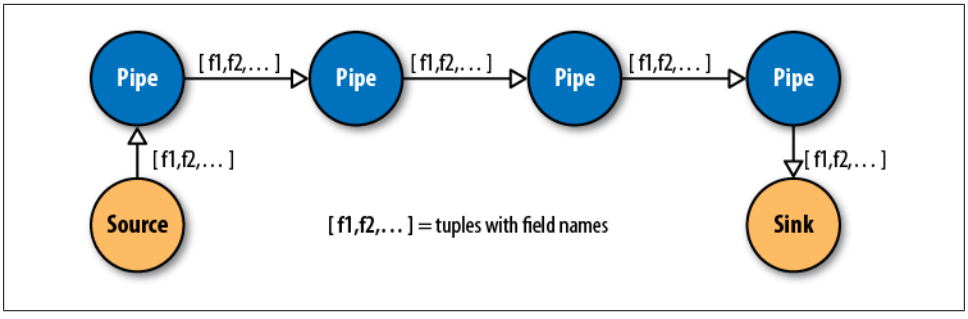


Figure 16-12. Pipes linked by fields and tuples

So, MapReduce keys and values are reduced to:

Fields
: Fields are a collection of either `String` names (such as "first_name"), numeric positions (such as 2, or –1, for the third and last position, respectively), or a combination of both, very much like column names. So fields are used to declare the names of values in a tuple and to select values by name from a tuple. The latter is like a SQL `select` call.

Tuple
: A tuple is simply an array of `java.lang.Comparable` objects. A tuple is very much like a database row or record.

And the map and reduce operations are abstracted behind one or more pipe instances (Figure 16-13):

`Each`
: The `Each` pipe processes a single input tuple at a time. It may apply either a `Function` or a `Filter` operation (described shortly) to the input tuple.

`GroupBy`
: The `GroupBy` pipe groups tuples on grouping fields. It behaves just like the SQL `group by` statement. It can also merge multiple input tuple streams into a single stream if they all share the same field names.

`CoGroup`
: The `CoGroup` pipe joins multiple tuple streams together by common field names, and it also groups the tuples by the common grouping fields. All standard join types (inner, outer, etc.) and custom joins can be used across two or more tuple streams.

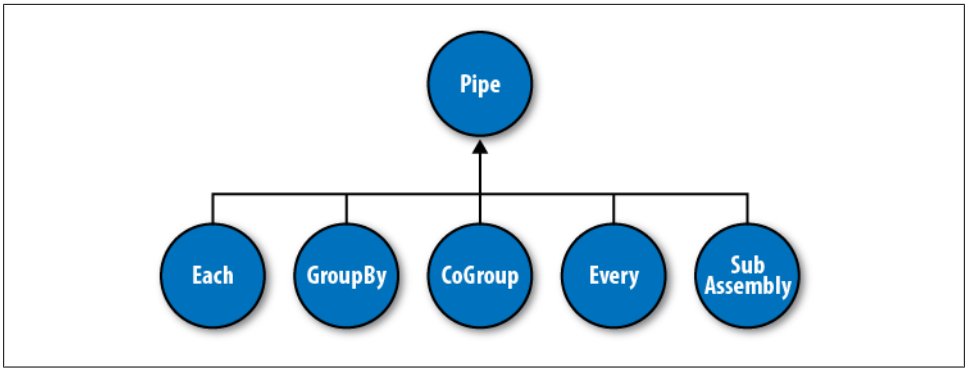


Figure 16-13. Pipe types

`Every`
: The `Every` pipe processes a single grouping of tuples at a time, where the group was grouped by a `GroupBy` or `CoGroup` pipe. The `Every` pipe may apply either an Aggregator or a Buffer operation to the grouping.

`SubAssembly`
: The `SubAssembly` pipe allows for nesting of assemblies inside a single pipe, which can, in turn, be nested in more complex assemblies.

All these pipes are chained together by the developer into "pipe assemblies" in which each assembly can have many input tuple streams (sources) and many output tuple streams (sinks). See Figure 16-14.

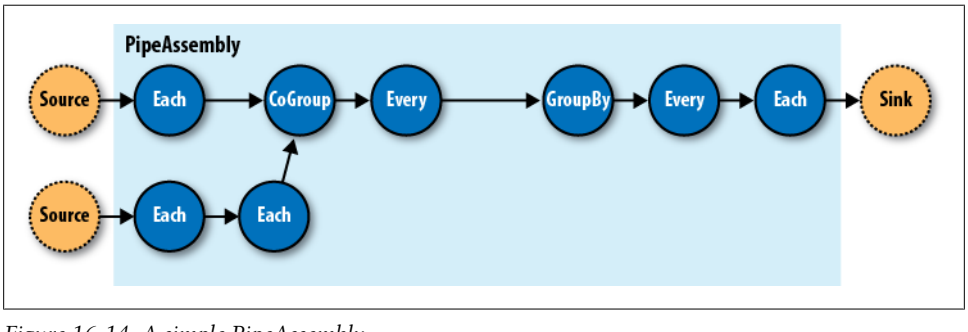


Figure 16-14. A simple PipeAssembly

On the surface, this might seem more complex than the traditional MapReduce model. And admittedly there are more concepts here than Map, Reduce, Key, and Value. But in practice, there are many more concepts that must all work in tandem to provide different behaviors.

For example, if a developer wanted to provide a "secondary sorting" of reducer values, he would need to implement a map, a reduce, a "composite" Key (two Keys nested in

a parent Key), a Value, a Partitioner, an "output value grouping" Comparator, and an "output key" Comparator, all of which would be coupled to one another in varying ways and, very likely, would not be reusable in subsequent applications.

In Cascading, this would be one line of code: `new GroupBy(<previous>, <grouping fields>, <secondary sorting fields>)`, where `previous` is the pipe that came before.

Operations

As mentioned earlier, Cascading departs from MapReduce by introducing alternative operations that are applied either to individual tuples or groups of tuples (Figure 16-15):

`Function`
: A `Function` operates on individual input tuples and may return zero or more output tuples for every one input. Functions are applied by the `Each` pipe.

`Filter`
: A `Filter` is a special kind of function that returns a Boolean value indicating whether the current input tuple should be removed from the tuple stream. A `function` could serve this purpose, but the `Filter` is optimized for this case, and many filters can be grouped by "logical" filters such as `And`, `Or`, `Xor`, and `Not`, rapidly creating more complex filtering operations.

`Aggregator`
: An `Aggregator` performs some operation against a group of tuples, where the grouped tuples are grouped by a common set of field values. For example, all tuples having the same "last-name" value. Common `Aggregator` implementations would be `Sum`, `Count`, `Average`, `Max`, and `Min`.

`Buffer`
: A `Buffer` is similar to the `Aggregator`, except it is optimized to act as a "sliding window" across all the tuples in a unique grouping. This is useful when the developer needs to efficiently insert missing values in an ordered set of tuples (such as a missing date or duration) or create a running average. Usually `Aggregator` is the operation of choice when working with groups of tuples, since many `Aggregators` can be chained together very efficiently, but sometimes a `Buffer` is the best tool for the job.

Operations are bound to pipes when the pipe assembly is created (Figure 16-16).

The `Each` and `Every` pipes provide a simple mechanism for selecting some or all values out of an input tuple before being passed to its child operation. And there is a simple mechanism for merging the operation results with the original input tuple to create the output tuple. Without going into great detail, this allows for each operation only to care about argument tuple values and fields, not the whole set of fields in the current input tuple. Subsequently, operations can be reusable across applications in the same way that Java methods can be reusable.

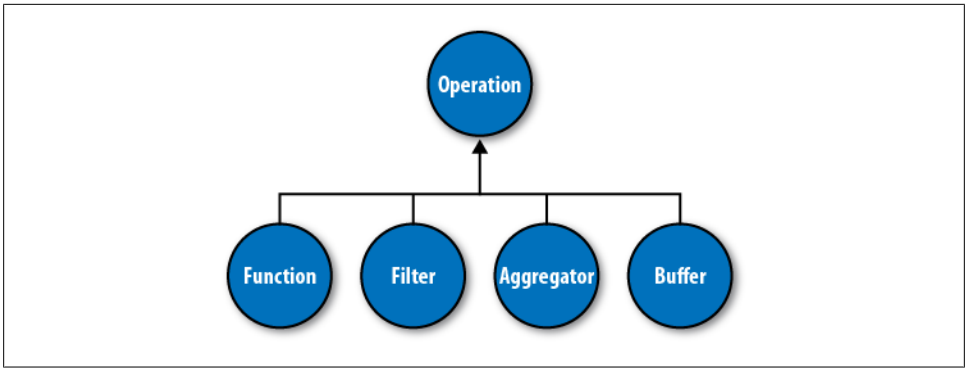


Figure 16-15. Operation types

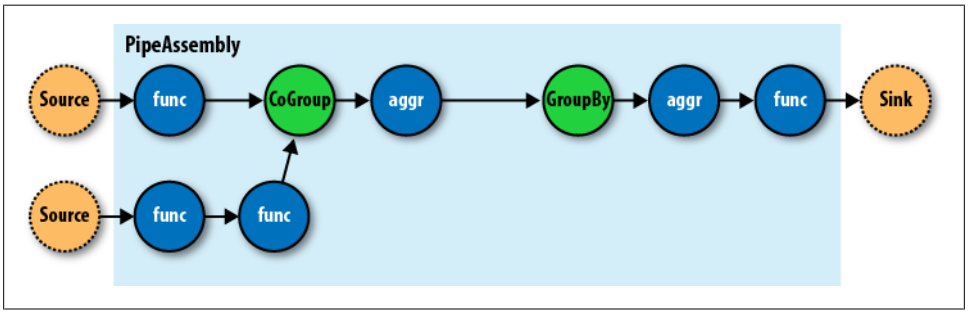


Figure 16-16. An assembly of operations

For example, in Java, a method declared as `concatenate(String first, String second)` is more abstract than `concatenate(Person person)`. In the second case, the `concatenate()` function must "know" about the `Person` object; in the first case, it is agnostic to where the data came from. Cascading operations exhibit this same quality.

Taps, Schemes, and Flows

In many of the previous diagrams, there are references to "sources" and "sinks." In Cascading, all data is read from or written to `Tap` instances, but is converted to and from tuple instances via `Scheme` objects:

`Tap`
: A `Tap` is responsible for the "how" and "where" parts of accessing data. For example, is the data on HDFS or the local filesystem? In Amazon S3 or over HTTP?

`Scheme`
: A `Scheme` is responsible for reading raw data and converting it to a tuple and/or writing a tuple out into raw data, where this "raw" data can be lines of text, Hadoop binary sequence files, or some proprietary format.

Note that `Taps` are not part of a pipe assembly, and so they are not a type of `Pipe`.

But they are connected with pipe assemblies when they are made cluster-executable. When a pipe assembly is connected with the necessary number of source and sink `Tap` instances, we get a `Flow`. A `Flow` is created when a pipe assembly is connected with its required number of source and sink `Taps`, and the `Taps` either emit or capture the field names the pipe assembly expects. That is, if a `Tap` emits a tuple with the field name "line" (by reading data from a file on HDFS), the head of the pipe assembly must be expecting a "line" value as well. Otherwise, the process that connects the pipe assembly with the `Taps` will immediately fail with an error.

So pipe assemblies are really data process definitions, and are not "executable" on their own. They must be connected to source and sink `Tap` instances before they can run on a cluster. This separation between `Taps` and pipe assemblies is part of what makes Cascading so powerful.

If you think of pipe assemblies like a Java class, then a `Flow` is like a Java Object instance (Figure 16-17). That is, the same pipe assembly can be "instantiated" many times into new `Flow`, in the same application, without fear of any interference between them. This allows pipe assemblies to be created and shared like standard Java libraries.

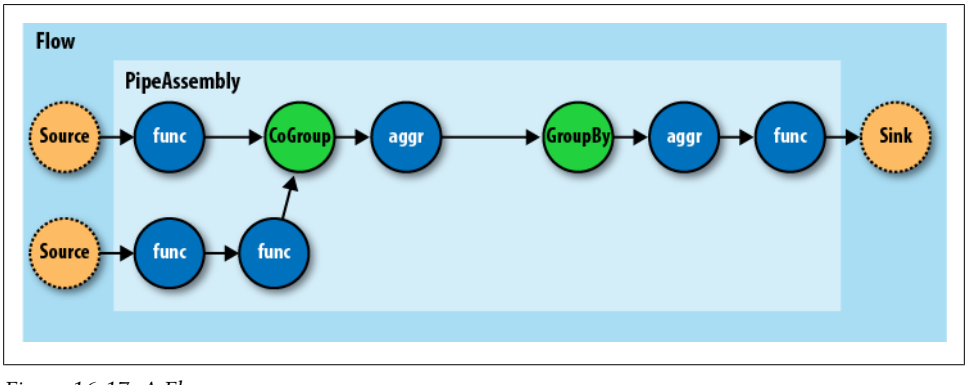


Figure 16-17. A Flow

Cascading in Practice

Now that we know what Cascading is and have a good idea of how it works, what does an application written in Cascading look like? See Example 16-2.

Example 16-2. Word count and sort

```
Scheme sourceScheme =
  new TextLine(new Fields("line")); ❶
Tap source =
  new Hfs(sourceScheme, inputPath); ❷

Scheme sinkScheme = new TextLine(); ❸
Tap sink =
  new Hfs(sinkScheme, outputPath, SinkMode.REPLACE); ❹
```

```
Pipe assembly = new Pipe("wordcount"); ❺

String regexString = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
Function regex = new RegexGenerator(new Fields("word"), regexString);
assembly =
  new Each(assembly, new Fields("line"), regex); ❻

assembly =
  new GroupBy(assembly, new Fields("word")); ❼

Aggregator count = new Count(new Fields("count"));
assembly = new Every(assembly, count); ❽

assembly =
  new GroupBy(assembly, new Fields("count"), new Fields("word")); ❾

FlowConnector flowConnector = new FlowConnector();
Flow flow =
  flowConnector.connect("word-count", source, sink, assembly); ❿

flow.complete();⓫
```

❶ We create a new `Scheme` that reads simple text files and emits a new `Tuple` for each line in a field named "line," as declared by the `Fields` instance.

❸ We create a new `Scheme` that writes simple text files and expects a `Tuple` with any number of fields/values. If more than one value, they will be tab-delimited in the output file.

❷ We create source and sink `Tap` instances that reference the input file and output ❹ directory, respectively. The sink `Tap` will overwrite any file that may already exist.

❺ We construct the head of our pipe assembly and name it "wordcount." This name is used to bind the source and sink `Taps` to the assembly. Multiple heads or tails would require unique names.

❻ We construct an `Each` pipe with a function that will parse the "line" field into a new `Tuple` for each word encountered.

❼ We construct a `GroupBy` pipe that will create a new `Tuple` grouping for each unique value in the field "word."

❽ We construct an `Every` pipe with an `Aggregator` that will count the number of `Tuple`s in every unique word group. The result is stored in a field named "count."

❾ We construct a `GroupBy` pipe that will create a new `Tuple` grouping for each unique value in the field "count" and secondary sort each value in the field "word." The result will be a list of "count" and "word" values with "count" sorted in increasing order.

❿ We connect the pipe assembly to its sources and sinks into a `Flow`, and then execute ⓫ the `Flow` on the cluster.

In the example, we count the words encountered in the input document, and we sort the counts in their natural order (ascending). And if some words have the same "count" value, these words are sorted in their natural order (alphabetical).

One obvious problem with this example is that some words might have uppercase letters—for example, "the" and "The" when the word comes at the beginning of a sentence. So we might decide to insert a new operation to force all the words to lowercase, but we realize that all future applications that need to parse words from documents should have the same behavior, so we decide to create a reusable pipe called `SubAssembly`, just like we would by creating a subroutine in a traditional application (see Example 16-3).

Example 16-3. Creating a SubAssembly

```
public class ParseWordsAssembly extends SubAssembly ❶
  {
  public ParseWordsAssembly(Pipe previous)
    {
    String regexString = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
    Function regex = new RegexGenerator(new Fields("word"), regexString);
    previous = new Each(previous, new Fields("line"), regex);

    String exprString = "word.toLowerCase()";
    Function expression =
      new ExpressionFunction(new Fields("word"), exprString, String.class); ❷
    previous = new Each(previous, new Fields("word"), expression);

    setTails(previous); ❸
    }
  }
```

❶ We subclass the `SubAssembly` class, which is itself a kind of `Pipe`.

❷ We create a Java expression function that will call `toLowerCase()` on the `String` value in the field named "word." We must also pass in the Java type the expression expects "word" to be, in this case, `String`. (*http://www.janino.net/* is used under the covers.)

❸ We must tell the `SubAssembly` superclass where the tail ends of our pipe subassembly are.

First, we create a `SubAssembly` pipe to hold our "parse words" pipe assembly. Because this is a Java class, it can be reused in any other application, as long as there is an incoming field named "word" (Example 16-4). Note that there are ways to make this function even more generic, but they are covered in the Cascading User Guide.

Example 16-4. Extending word count and sort with a SubAssembly

```
Scheme sourceScheme = new TextLine(new Fields("line"));
Tap source = new Hfs(sourceScheme, inputPath);

Scheme sinkScheme = new TextLine(new Fields("word", "count"));
Tap sink = new Hfs(sinkScheme, outputPath, SinkMode.REPLACE);
```

```
Pipe assembly = new Pipe("wordcount");

assembly =
  new ParseWordsAssembly(assembly); ❶

assembly = new GroupBy(assembly, new Fields("word"));

Aggregator count = new Count(new Fields("count"));
assembly = new Every(assembly, count);

assembly = new GroupBy(assembly, new Fields("count"), new Fields("word"));

FlowConnector flowConnector = new FlowConnector();
Flow flow = flowConnector.connect("word-count", source, sink, assembly);

flow.complete();
```

❶ We replace the `Each` from the previous example with our `ParseWordsAssembly` pipe.

Finally, we just substitute in our new `SubAssembly` right where the previous `Every` and word parser function was used in the previous example. This nesting can continue as deep as necessary.

Flexibility

Take a step back and see what this new model has given us or, better yet, what it has taken away.

You see, we no longer think in terms of MapReduce jobs, or `Mapper` and `Reducer` interface implementations and how to bind or link subsequent MapReduce jobs to the ones that precede them. During runtime, the Cascading "planner" figures out the optimal way to partition the pipe assembly into MapReduce jobs and manages the linkages between them (Figure 16-18).

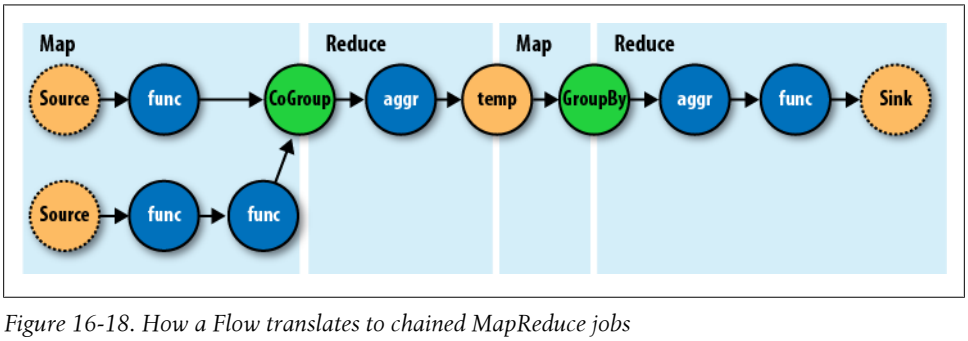


Figure 16-18. How a Flow translates to chained MapReduce jobs

Because of this, developers can build applications of arbitrary granularity. They can start with a small application that just filters a logfile, but then can iteratively build up more features into the application as needed.

Since Cascading is an API and not a syntax like strings of SQL, it is more flexible. First off, developers can create domain-specific languages (DSLs) using their favorite language, such as Groovy, JRuby, Jython, Scala, and others (see the project site for examples). Second, developers can extend various parts of Cascading, such as allowing custom Thrift or JSON objects to be read and written to and allowing them to be passed through the tuple stream.

Hadoop and Cascading at ShareThis

ShareThis is a sharing network that makes it simple to share any online content. With the click of a button on a web page or browser plug-in, ShareThis allows users to seamlessly access their contacts and networks from anywhere online and share the content through email, IM, Facebook, Digg, mobile SMS, etc., without ever leaving the current page. Publishers can deploy the ShareThis button to tap the service's universal sharing capabilities to drive traffic, stimulate viral activity, and track the sharing of online content. ShareThis also simplifies social media services by reducing clutter on web pages and providing instant distribution of content across social networks, affiliate groups, and communities.

As ShareThis users share pages and information through the online widgets, a continuous stream of events enter the ShareThis network. These events are first filtered and processed, and then handed to various backend systems, including AsterData, Hypertable, and Katta.

The volume of these events can be huge, too large to process with traditional systems. This data can also be very "dirty" thanks to "injection attacks" from rogue systems, browser bugs, or faulty widgets. For this reason, ShareThis chose to deploy Hadoop as the preprocessing and orchestration frontend to their backend systems. They also chose to use Amazon Web Services to host their servers, on the Elastic Computing Cloud (EC2), and provide long-term storage, on the Simple Storage Service (S3), with an eye toward leveraging Elastic MapReduce (EMR).

In this overview, we will focus on the "log processing pipeline" (Figure 16-19). This pipeline simply takes data stored in an S3 bucket, processes it (described shortly), and stores the results back into another bucket. Simple Queue Service (SQS) is used to coordinate the events that mark the start and completion of data processing runs. Downstream, other processes pull data that load AsterData, pull URL lists from Hypertable to source a web crawl, or pull crawled page data to create Lucene indexes for use by Katta. Note that Hadoop is central to the ShareThis architecture. It is used to coordinate the processing and movement of data between architectural components.

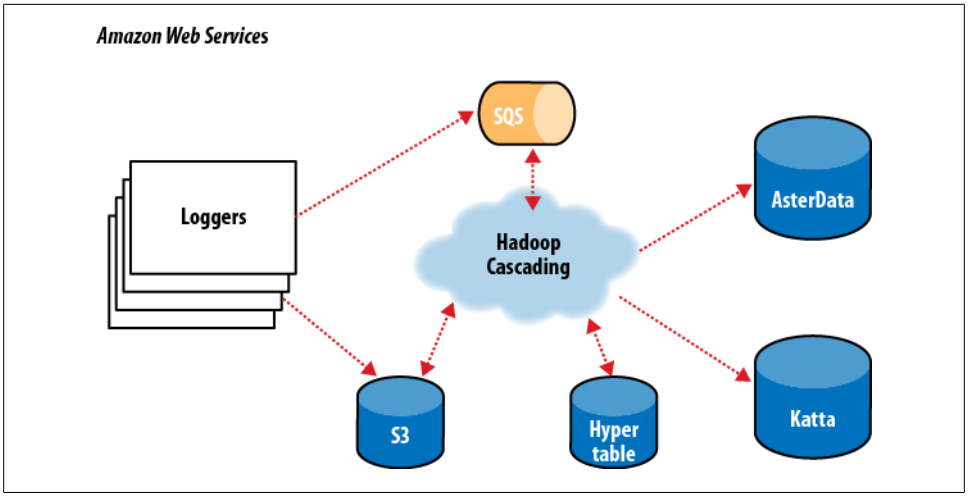


Figure 16-19. The ShareThis log processing pipeline

With Hadoop as the frontend, all the event logs can be parsed, filtered, cleaned, and organized by a set of rules before ever being loaded into the AsterData cluster or used by any other component. AsterData is a clustered data warehouse that can support large datasets and allow for complex ad hoc queries using a standard SQL syntax. ShareThis chose to clean and prepare the incoming datasets on the Hadoop cluster and then to load that data into the AsterData cluster for ad hoc analysis and reporting. Though possible with AsterData, it made a lot of sense to use Hadoop as the first stage in the processing pipeline to offset load on the main data warehouse.

Cascading was chosen as the primary data processing API to simplify the development process, codify how data is coordinated between architectural components, and provide the developer-facing interface to those components. This represents a departure from more "traditional" Hadoop use cases, which essentially just query stored data. Instead, Cascading and Hadoop together provide a better and simpler structure to the complete solution, end-to-end, and thus provide more value to the users.

For developers, Cascading made it easy to start with a simple unit test (by subclassing `cascading.ClusterTestCase`) that did simple text parsing and then to layer in more processing rules while keeping the application logically organized for maintenance. Cascading aided this organization in a couple of ways. First, standalone operations (`Functions`, `Filters`, etc.) could be written and tested independently. Second, the application was segmented into stages: one for parsing, one for rules, and a final stage for binning/collating the data, all via the `SubAssembly` base class described earlier.

The data coming from the ShareThis loggers looks a lot like Apache logs, with date/timestamps, share URLs, referrer URLs, and a bit of metadata. To use the data for analysis downstream, the URLs needed to be unpacked (parsing query-string data, domain names, etc.). So a top-level `SubAssembly` was created to encapsulate the parsing, and child subassemblies were nested inside to handle specific fields if they were sufficiently complex to parse.

The same was done for applying rules. As every `Tuple` passed through the rules `SubAssembly`, it was marked as "bad" if any of the rules were triggered. Along with the "bad" tag, a description of why the record was bad was added to the `Tuple` for later review.

Finally, a splitter `SubAssembly` was created to do two things. First, it allows for the tuple stream to split into two: one stream for "good" data and one for "bad" data. Second, the splitter binned the data into intervals, such as every hour. To do this, only two operations were necessary: the first to create the interval from the *timestamp* value already present in the stream, and the second to use the *interval* and *good/bad* metadata to create a directory path (for example, *05/good/*, where "05" is 5 a.m. and "good" means the tuple passed all the rules). This path would then be used by the Cascading `TemplateTap`, a special `Tap` that can dynamically output tuple streams to different locations based on values in the `Tuple`. In this case, the `TemplateTap` used the "path" value to create the final output path.

The developers also created a fourth `SubAssembly`—this one to apply Cascading Assertions during unit testing. These assertions double-checked that rules and parsing subassemblies did their job.

In the unit test in Example 16-5, we see the splitter isn't being tested, but it is added in another integration test not shown.

Example 16-5. Unit testing a Flow

```
public void testLogParsing() throws IOException
  {
  Hfs source = new Hfs(new TextLine(new Fields("line")), sampleData);
  Hfs sink =
    new Hfs(new TextLine(), outputPath + "/parser", SinkMode.REPLACE);

  Pipe pipe = new Pipe("parser");

  // split "line" on tabs
  pipe = new Each(pipe, new Fields("line"), new RegexSplitter("\t"));

  pipe = new LogParser(pipe);

  pipe = new LogRules(pipe);
```

```
// testing only assertions
pipe = new ParserAssertions(pipe);

Flow flow = new FlowConnector().connect(source, sink, pipe);

flow.complete(); // run the test flow

// verify there are 98 tuples, 2 fields, and matches the regex pattern
// for TextLine schemes the tuples are { "offset", "line" }
validateLength(flow, 98, 2, Pattern.compile("^[0-9]+(\\t[^\\t]*){19}$"));
}
```

For integration and deployment, many of the features built into Cascading allowed for easier integration with external systems and for greater process tolerance.

In production, all the subassemblies are joined and planned into a `Flow`, but instead of just source and sink `Taps`, trap `Taps` were planned in (Figure 16-20). Normally, when an operation throws an exception from a remote mapper or reducer task, the `Flow` will fail and kill all its managed MapReduce jobs. When a `Flow` has traps, any exceptions are caught and the data causing the exception is saved to the `Tap` associated with the current trap. Then the next `Tuple` is processed without stopping the `Flow`. Sometimes you want your `Flow`s to fail on errors, but in this case, the ShareThis developers knew they could go back and look at the "failed" data and update their unit tests while the production system kept running. Losing a few hours of processing time was worse than losing a couple of bad records.

Using Cascading's event listeners, Amazon SQS could be integrated. When a `Flow` finishes, a message is sent to notify other systems that there is data ready to be picked up from Amazon S3. On failure, a different message is sent, alerting other processes.

The remaining downstream processes pick up where the log processing pipeline leaves off on different independent clusters. The log processing pipeline today runs once a day, so there is no need to keep a 100-node cluster sitting around for the 23 hours it has nothing to do. So it is decommissioned and recommissioned 24 hours later.

In the future, it would be trivial to increase this interval on smaller clusters to every 6 hours, or 1 hour, as the business demands. Independently, other clusters are booting and shutting down at different intervals based on the needs of the business unit responsible for that component. For example, the web crawler component (using Bixo, a Cascading-based web-crawler toolkit developed by EMI and ShareThis) may run continuously on a small cluster with a companion Hypertable cluster. This on-demand model works very well with Hadoop, where each cluster can be tuned for the kind of workload it is expected to handle.

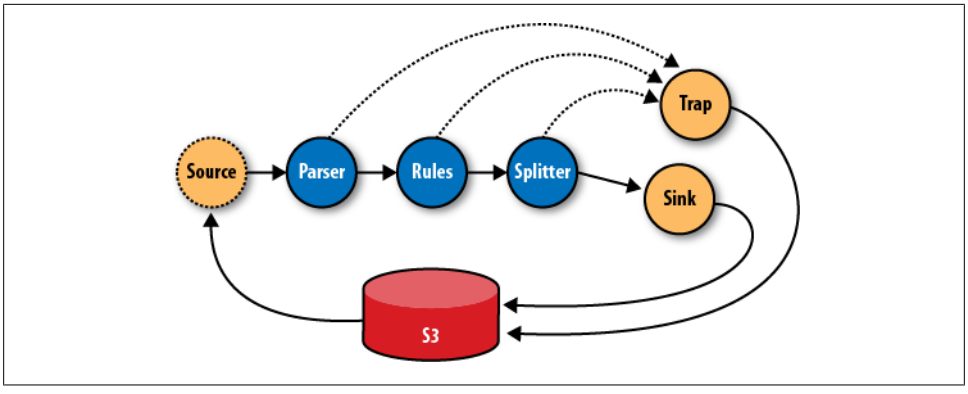


Figure 16-20. The ShareThis log processing Flow

Summary

Hadoop is a very powerful platform for processing and coordinating the movement of data across various architectural components. Its only drawback is that the primary computing model is MapReduce.

Cascading aims to help developers build powerful applications quickly and simply, through a well-reasoned API, without needing to think in MapReduce, and while leaving the heavy lifting of data distribution, replication, distributed process management, and liveness to Hadoop.

Read more about Cascading, join the online community, and download sample applications by visiting the project website.

—Chris K. Wensel

TeraByte Sort on Apache Hadoop

This article is reproduced from http://sortbenchmark.org/YahooHadoop.pdf, which was written in May 2008. Jim Gray and his successors define a family of benchmarks to find the fastest sort programs every year. TeraByte Sort and other sort benchmarks are listed with winners over the years at http://sortbenchmark.org/. In April 2009, Arun Murthy and I won the minute sort (where the aim is to sort as much data as possible in under one minute) by sorting 500 GB in 59 seconds on 1,406 Hadoop nodes. We also sorted a terabyte in 62 seconds on the same cluster. The cluster we used in 2009 was similar to the hardware listed in the following article, except that the network was much better, with only 2-to-1 oversubscription between racks instead of 5-to-1 in the previous year.

Additionally, we used LZO compression on the intermediate data between the nodes. We also sorted a petabyte (10^{15} bytes) in 975 minutes on 3,658 nodes, for an average rate of 1.03 TB/minute. See http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html for more details about the 2009 results.

Apache Hadoop is an open source software framework that dramatically simplifies writing distributed data-intensive applications. It provides a distributed filesystem, which is modeled after the Google File System,[1] and a MapReduce[2] implementation that manages distributed computation. Since the primary primitive of MapReduce is a distributed sort, most of the custom code is glue to get the desired behavior.

I wrote three Hadoop applications to run the terabyte sort:

1. `TeraGen` is a MapReduce program to generate the data.
2. `TeraSort` samples the input data and uses MapReduce to sort the data into a total order.
3. `TeraValidate` is a MapReduce program that validates the output is sorted.

The total is around 1,000 lines of Java code, which will be checked in to the Hadoop example directory.

`TeraGen` generates output data that is byte-for-byte equivalent to the C version, including the newlines and specific keys. It divides the desired number of rows by the desired number of tasks and assigns ranges of rows to each map. The map jumps the random number generator to the correct value for the first row and generates the following rows. For the final run, I configured `TeraGen` to use 1,800 tasks to generate a total of 10 billion rows in HDFS, with a block size of 512 MB.

`TeraSort` is a standard MapReduce sort, except for a custom partitioner that uses a sorted list of *N*–1 sampled keys that define the key range for each reduce. In particular, all keys such that *sample*[*i*–1] <= *key* < *sample*[*i*] are sent to reduce *i*. This guarantees that the output of reduce *i* are all less than the output of reduce *i*+1. To speed up the partitioning, the partitioner builds a two-level trie that quickly indexes into the list of sample keys based on the first two bytes of the key. `TeraSort` generates the sample keys by sampling the input before the job is submitted and writing the list of keys into HDFS.

I wrote an input and output format, which are used by all three applications, that read and write the text files in the right format. The output of the reduce has replication set to 1, instead of the default 3, because the contest does not require the output data be replicated on to multiple nodes. I configured the job with 1,800 maps and 1,800 reduces and `io.sort.mb`, `io.sort.factor`, `fs.inmemory.size.mb`, and a task heap size sufficient that transient data was never spilled to disk other at the end of the map. The sampler

1. S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System." In *19th Symposium on Operating Systems Principles* (October 2003), Lake George, NY: ACM.
2. J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In *Sixth Symposium on Operating System Design and Implementation* (December 2004), San Francisco, CA.

used 100,000 keys to determine the reduce boundaries, although as can be seen in Figure 16-21, the distribution between reduces was hardly perfect and would benefit from more samples. You can see the distribution of running tasks over the job run in Figure 16-22.

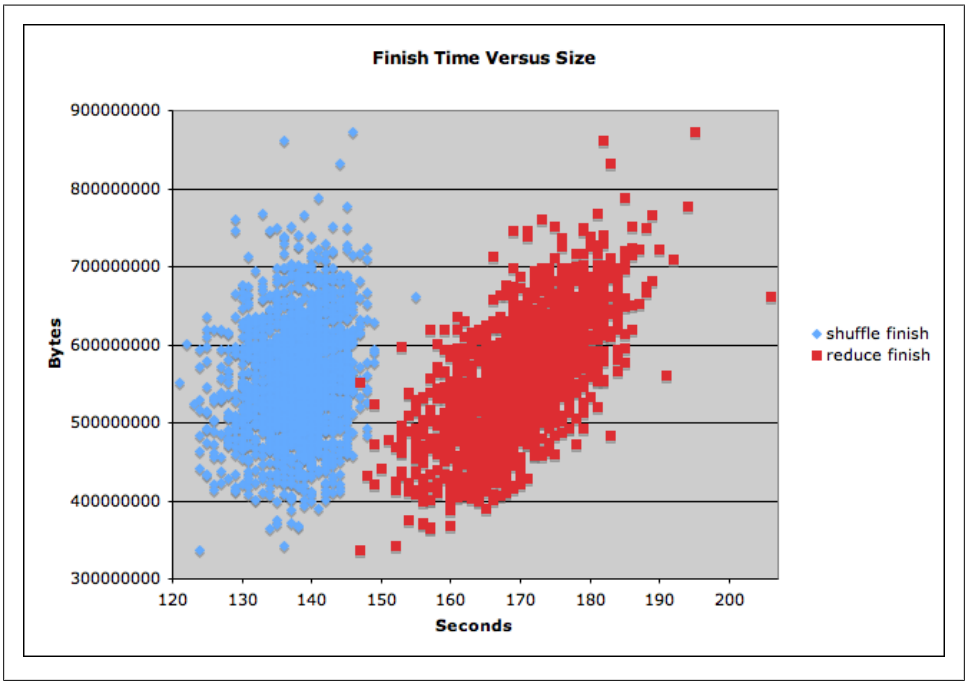


Figure 16-21. Plot of reduce output size versus finish time

`TeraValidate` ensures that the output is globally sorted. It creates one map per file in the output directory, and each map ensures that each key is less than or equal to the previous one. The map also generates records with the first and last keys of the file, and the reduce ensures that the first key of file *i* is greater than the last key of file *i*–1. Any problems are reported as output of the reduce with the keys that are out of order.

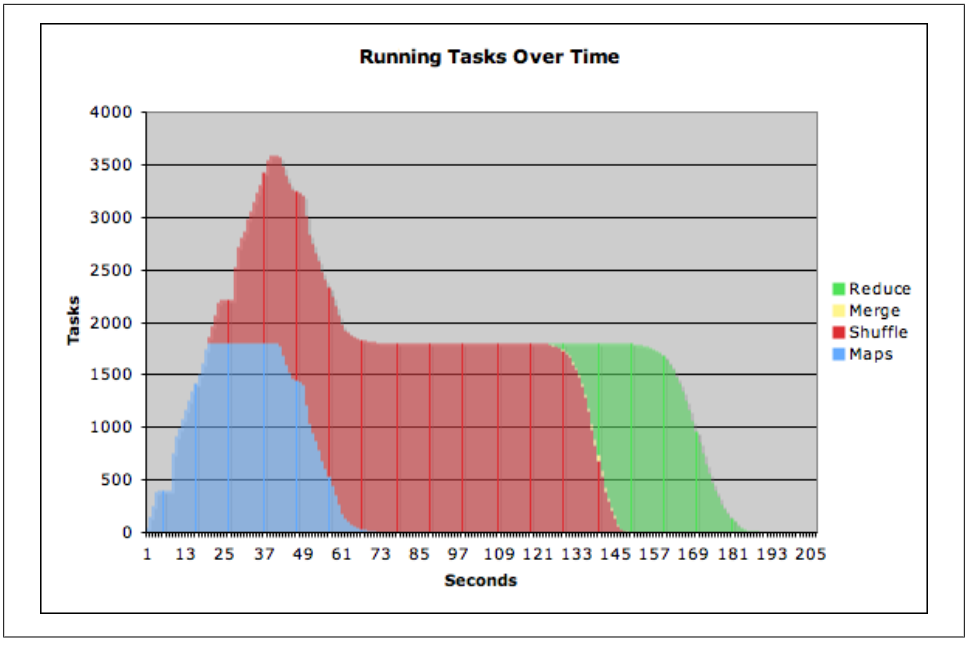


Figure 16-22. Number of tasks in each phase across time

The cluster I ran on was:

- 910 nodes
- 2 quad core Xeons at 2.0 GHz per node
- 4 SATA disks per node
- 8 G RAM per node
- 1 gigabit Ethernet on each node
- 40 nodes per rack
- 8 gigabit Ethernet uplinks from each rack to the core
- Red Hat Enterprise Linux Server release 5.1 (kernel 2.6.18)
- Sun Java JDK 1.6.0_05-b13

The sort completed in *209 seconds (3.48 minutes)*. I ran Hadoop trunk (pre-0.18.0) with patches for HADOOP-3443 and HADOOP-3446, which were required to remove intermediate writes to disk. Although I had the 910 nodes mostly to myself, the network core was shared with another active 2,000-node cluster, so the times varied a lot depending on the other activity.

—Owen O'Malley, Yahoo!

Using Pig and Wukong to Explore Billion-edge Network Graphs

Networks at massive scale are fascinating. The number of things they model are extremely general: if you have a collection of things (that we'll call nodes), they are related (edges), and if the nodes and edges tell a story (node/edge metadata), you have a network graph.

I started the Infochimps project, a site to find, share, or sell any dataset in the world. At Infochimps, we have a whole bag of tricks ready to apply to any interesting network graph that comes into the collection. We chiefly use Pig (described in Chapter 11) and Wukong, a toolkit we've developed for Hadoop streaming in the Ruby programming language. They let us write simple scripts like the ones that follow—almost all of which fit on a single printed page—to process terabyte-scale graphs. Here are a few datasets that come up in a search for "network" on infochimps.org[3]:

- A social network, such as Twitter or Facebook. We somewhat impersonally model people as nodes, and relationships (`@mrflip` is friends with `@tom_e_white`) or actions (`@infochimps` mentioned `@hadoop`) as edges. The number of messages a user has sent and the bag of words from all those messages are each important pieces of node metadata.
- A linked document collection such as Wikipedia or the entire Web.[4] Each page is a node (carrying its title, view count, and categories as node metadata). Each hyperlink is an edge, and the frequency at which people click from one page to the next is edge metadata.
- The connections of neurons (nodes) and synapses (edges) in the *C. elegans* roundworm.[5]

3. See *http://infochimps.org/search?query=network*
4. *http://www.datawrangling.com/wikipedia-page-traffic-statistics-dataset*
5. *http://www.wormatlas.org/neuronalwiring.html*

- A highway map, with exits as nodes and highway segments as edges. The Open Street Map project's dataset has global coverage of place names (node metadata), street number ranges (edge metadata), and more.[6]
- Or the many esoteric graphs that fall out when you take an interesting system and shake it just right. Stream through a few million Twitter messages, and emit an edge for every pair of nonkeyboard characters occurring within the same message. Simply by observing "often, when humans use 最, they also use 近," you can re-create a map of human languages (see Figure 16-23).

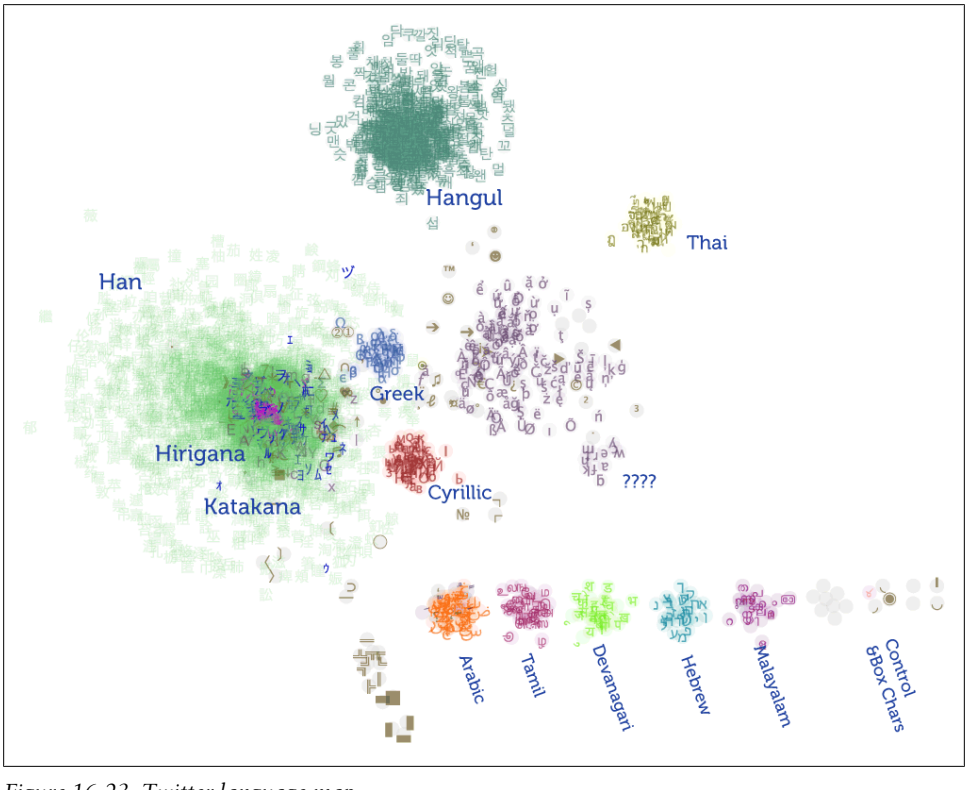


Figure 16-23. Twitter language map

6. *http://www.openstreetmap.org/*

What's amazing about these organic network graphs is that given enough data, a collection of powerful tools are able to *generically* use this network structure to expose insight. For example, we've used variants of the same algorithm[7] to do each of:

- Rank the most important pages in the Wikipedia linked-document collection. Google uses a vastly more refined version of this approach to identify top search hits.
- Identify celebrities and experts in the Twitter social graph. Users who have many more followers than their "trstrank" would imply are often spammers.
- Predict a school's impact on student education, using millions of anonymized exam scores gathered over five years.

Measuring Community

The most interesting network in the Infochimps collection is a massive crawl of the Twitter social graph. With more than 90 million nodes and 2 billion edges, it is a marvelous instrument for understanding what people talk about and how they relate to each other. Here is an exploration, using the subgraph of "People who talk about Infochimps or Hadoop,"[8] of three ways to characterize a user's community:

- Who are the people they converse with (the @reply graph)?
- Do the people they engage with reciprocate that attention (symmetric links)?
- Among the user's community, how many engage with each other (clustering coefficient)?

Everybody's Talkin' at Me: The Twitter Reply Graph

Twitter lets you reply to another user's message and thus engage in conversation. Since it's an expressly public activity, a reply is a strong *social token*: it shows interest in what the other is saying and demonstrates that interest is worth rebroadcasting.

The first step in our processing is done in Wukong, a Ruby language library for Hadoop. It lets us write small, agile programs capable of handling multiterabyte data streams. Here is a snippet from the class that represents a twitter message (or *tweet*):[9]

```
class Tweet < Struct.new(:tweet_id, :screen_name, :created_at,
                         :reply_tweet_id, :reply_screen_name, :text)
  def initialize(raw_tweet)
```

7. All are steady-state network flow problems. A flowing crowd of websurfers wandering the linked-document collection will visit the most interesting pages the most often. The transfer of social capital implied by social network interactions highlights the most central actors within each community. The year-to-year progress of students to higher or lower test scores implies what each school's effect on a generic class would be.
8. Chosen without apology, in keeping with the ego-centered ethos of social networks.
9. You can find full working source code on this book's website.

```
    # ... gory details of parsing raw tweet omitted
  end

  # Tweet is a reply if there's something in the reply_tweet_id slot
  def is_reply?
    not reply_tweet_id.blank?
  true
end
```

Twitter's Stream API lets anyone easily pull gigabytes of messages.[10] They arrive in a raw JSON format:

```
{"text":"Just finished the final draft for Hadoop: the Definitive Guide!",
 "screen_name":"tom_e_white","reply_screen_name":null,"id":3239897342,
 "reply_tweet_id":null,...}
{"text":"@tom_e_white Can't wait to get a copy!",
 "screen_name":"mrflip","reply_screen_name":"tom_e_white","id":3239873453,
 "reply_tweet_id":3239897342,...}
{"text":"@josephkelly great job on the #InfoChimps API.
  Remind me to tell you about the time a baboon broke into our house.",
 "screen_name":"wattsteve","reply_screen_name":"josephkelly","id":16434069252,...}
{"text":"@mza Re: http://j.mp/atbroxmr Check out @James_Rubino's
  http://bit.ly/clusterfork ? Lots of good hadoop refs there too",
 "screen_name":"mrflip","reply_screen_name":"@mza","id":7809927173,...}
{"text":"@tlipcon divide lots of data into little parts. Magic software gnomes
  fix up the parts, elves then assemble those into whole things #hadoop",
 "screen_name":"nealrichter","reply_screen_name":"tlipcon","id":4491069515,...}
```

The `reply_screen_name` and `reply_tweet_id` let you follow the conversation (as you can see, they're otherwise `null`). Let's find each reply and emit the respective user IDs as an edge:[11]

```
class ReplyGraphMapper < LineStreamer
  def process(raw_tweet)
    tweet = Tweet.new(raw_tweet)
    if tweet.is_reply?
      emit [tweet.screen_name, tweet.reply_screen_name]
    end
  end
end
```

The Mapper derives from `LineStreamer`, a class that feeds each line as a single record to its `process` method. We only have to define that `process` method; Wukong and Hadoop take care of the rest. In this case, we use the raw JSON record to create a tweet object. Where user A replies to user B, emit the edge as A and B separated by a tab. The raw output will look like this:

```
% reply_graph_mapper --run raw_tweets.json a_replies_b.tsv
mrflip          tom_e_white
```

10. Refer to the Twitter developer site, or use a tool like Hayes Davis' Flamingo.

11. In practice, we of course use numeric IDs and not screen names, but it's easier to follow along with screen names. In order to keep the graph-theory discussion general, I'm going to play loose with some details and leave out various janitorial details of loading and running.

```
wattsteve       josephkelly
mrflip          mza
nealrichter     tlipcon
```

You should read this as "a replies b" and interpret it as a directed "out" edge: `@wattsteve` conveys social capital to `@josephkelly`.

Edge pairs versus adjacency list

That is the *edge pairs* representation of a network. It's simple, and it gives an equal jumping-off point for in- or out-edges, but there's some duplication of data. You can tell the same story from the node's point of view (and save some disk space) by rolling up on the source node. We call this the *adjacency list*, and it can be generated in Pig by a simple `GROUP BY`. Load the file:

```
a_replies_b = LOAD 'a_replies_b.tsv' AS (src:chararray, dest:chararray);
```

Then find all edges out from each node by grouping on source:

```
replies_out = GROUP a_replies_b BY src;
DUMP replies_out
```

```
(cutting,{(tom_e_white)})
(josephkelly,{(wattsteve)})
(mikeolson,{(LusciousPear),(kevinweil),(LusciousPear),(tlipcon)})
(mndoci,{(mrflip),(peteskomoroch),(LusciousPear),(mrflip)})
(mrflip,{(LusciousPear),(mndoci),(mndoci),(esammer),(ogrisel),(esammer),(wattsteve)})
(peteskomoroch,{(CMastication),(esammer),(DataJunkie),(mndoci),(nealrichter),...
(tlipcon,{(LusciousPear),(LusciousPear),(nealrichter),(mrflip),(kevinweil)})
(tom_e_white,{(mrflip),(lenbust)})
```

Degree

A simple, useful measure of influence is the number of replies a user receives. In graph terms, this is the *degree* (specifically the *in-degree*, since this is a directed graph).

Pig's nested `FOREACH` syntax lets us count the distinct incoming repliers (neighbor nodes) and the total incoming replies in one pass:[12]

```
a_replies_b = LOAD 'a_replies_b.tsv' AS (src:chararray, dest:chararray);
replies_in  = GROUP a_replies_b BY dest; -- group on dest to get in-links
replies_in_degree = FOREACH replies_in {
  nbrs = DISTINCT a_replies_b.src;
  GENERATE group, COUNT(nbrs), COUNT(a_replies_b);
};
DUMP replies_in_degree
```

```
(cutting,1L,1L)
(josephkelly,1L,1L)
```

12. Due to the small size of the edge pair records and a pesky Hadoop implementation detail, the Mapper may spill data to disk early. If the jobtracker dashboard shows "spilled records" greatly exceeding "map output records," try bumping up the `io.sort.record.percent`:

```
PIG_OPTS="-Dio.sort.record.percent=0.25 -Dio.sort.mb=350" pig my_file.pig
```

```
(mikeolson,3L,4L)
(mndoci,3L,4L)
(mrflip,5L,9L)
(peteskomoroch,9L,18L)
(tlipcon,4L,8L)
(tom_e_white,2L,2L)
```

In this sample, `@peteskomoroch` has nine neighbors and 18 incoming replies, far more than most. This large variation in degree is typical for social networks. Most users see a small number of replies, but a few celebrities—such as `@THE_REAL_SHAQ` (basketball star Shaquille O'Neill) or `@sockington` (a fictional cat)—receive millions. By contrast, almost every intersection on a road map is four-way.[13] The skewed dataflow produced by this wild variation in degree has important ramifications for how you process such graphs—more later.

Symmetric Links

Although millions of people have given `@THE_REAL_SHAQ` a shout-out on twitter, he has understandably not reciprocated with millions of replies. As the graph shows, I frequently converse with `@mndoci`,[14] making ours a *symmetric link*. This accurately reflects the fact that I have more in common with `@mndoci` than with `@THE_REAL_SHAQ`.

One way to find symmetric links is to take the edges in `A Replied To B` that are also in `A Replied By B`. We can do that set intersection with an inner self-join:[15]

```
a_repl_to_b = LOAD 'a_replies_b.tsv' AS (user_a:chararray, user_b:chararray);
a_repl_by_b = LOAD 'a_replies_b.tsv' AS (user_b:chararray, user_a:chararray);
-- symmetric edges appear in both sets
a_symm_b_j  = JOIN a_repl_to_b BY (user_a, user_b),
                   a_repl_by_b BY (user_a, user_b);
...
```

However, this sends two full copies of the edge-pairs list to the reduce phase, doubling the memory required. We can do better by noticing that from a node's point of view, a symmetric link is equivalent to a paired edge: one out and one in. Make the graph undirected by putting the node with lowest sort order in the first slot—but preserve the direction as a piece of edge metadata:

```
a_replies_b = LOAD 'a_replies_b.tsv' AS (src:chararray, dest:chararray);
a_b_rels = FOREACH a_replies_b GENERATE
  ((src <= dest) ? src  : dest) AS user_a,
  ((src <= dest) ? dest : src)  AS user_b,
  ((src <= dest) ? 1 : 0)       AS a_re_b:int,
```

13. The largest outlier that comes to mind is the famous "Magic Roundabout" in Swindon, England, with degree 10. See *http://en.wikipedia.org/wiki/Magic_Roundabout_%28Swindon%29*.
14. Deepak Singh, open data advocate and bizdev manager of the Amazon AWS cloud.
15. Current versions of Pig get confused on self-joins, so just load the table with differently named relations as shown here.

```
  ((src <= dest) ? 0 : 1)       AS b_re_a:int;
DUMP a_b_rels
```

```
(mrflip,tom_e_white,1,0)
(josephkelly,wattsteve,0,1)
(mrflip,mza,1,0)
(nealrichter,tlipcon,0,1)
```

Now gather all edges for each node pair. A symmetric edge has at least one reply in each direction:

```
a_b_rels_g   = GROUP a_b_rels BY (user_a, user_b);
a_symm_b_all = FOREACH a_b_rels_g GENERATE
  group.user_a AS user_a,
  group.user_b AS user_b,
  (( (SUM(a_b_rels.a_re_b) > 0) AND
     (SUM(a_b_rels.b_re_a) > 0)     ) ? 1 : 0) AS is_symmetric:int;
DUMP a_symm_b_all
```

```
(mrflip,tom_e_white,1)
(mrflip,mza,0)
(josephkelly,wattsteve,0)
(nealrichter,tlipcon,1)
...
```

```
a_symm_b = FILTER a_symm_b_all BY (is_symmetric == 1);
STORE a_symm_b INTO 'a_symm_b.tsv';
```

Here's a portion of the output, showing that `@mrflip` and `@tom_e_white` have a symmetric link:

```
(mrflip,tom_e_white,1)
(nealrichter,tlipcon,1)
...
```

Community Extraction

So far, we've generated a node measure (in-degree) and an edge measure (symmetric link identification). Let's move out one step and look at a neighborhood measure: how many of a given person's friends are friends with each other? Along the way, we'll produce the edge set for a visualization like the one discussed earlier.

Get neighbors

Choose a seed node (here, `@hadoop`). First, round up the seed's neighbors:

```
a_replies_b = LOAD 'a_replies_b.tsv' AS (src:chararray, dest:chararray);
-- Extract edges that originate or terminate on the seed
n0_edges   = FILTER a_replies_b BY (src == 'hadoop') OR (dest == 'hadoop');
-- Choose the node in each pair that *isn't* our seed:
n1_nodes_all = FOREACH n0_edges GENERATE
  ((src == 'hadoop') ? dest : src) AS screen_name;
n1_nodes    = DISTINCT n1_nodes_all;
DUMP n1_nodes
```

Now intersect the set of neighbors with the set of starting nodes to find all edges originating in `n1_nodes`:

```
n1_edges_out_j = JOIN a_replies_b BY src,
                      n1_nodes    BY screen_name USING 'replicated';
n1_edges_out   = FOREACH n1_edges_out_j GENERATE src, dest;
```

Our copy of the graph (with more than 1 billion edges) is far too large to fit in memory. On the other hand, the neighbor count for a single user rarely exceeds a couple of million, which fits easily in memory. Including `USING 'replicated'` in the `JOIN` command instructs Pig to do a map-side join (also called a *fragment replicate join*). Pig holds the `n1_nodes` relation in memory as a lookup table and streams the full edge list past. Whenever the join condition is met—`src` is in the `n1_nodes` lookup table—it produces output. No reduce step means an enormous speedup!

To leave only edges where both source and destination are neighbors of the seed node, repeat the join:

```
n1_edges_j = JOIN n1_edges_out BY dest,
                  n1_nodes     BY screen_name USING 'replicated';
n1_edges   = FOREACH n1_edges_j GENERATE src, dest;
DUMP n1_edges
```

```
(mrflip,tom_e_white)
(mrflip,mza)
(wattsteve,josephkelly)
(nealrichter,tlipcon)
(bradfordcross,lusciouspear)
(mrflip,jeromatron)
(mndoci,mrflip)
(nealrichter,datajunkie)
```

Community metrics and the 1 million × 1 million problem

With `@hadoop`, `@cloudera`, and `@infochimps` as seeds, I applied similar scripts to 2 billion messages to create Figure 16-24 (this image is also hosted on this book's website).

As you can see, the big data community is very interconnected. The link neighborhood of a celebrity such as `@THE_REAL_SHAQ` is far more sparse. We can characterize this using the *clustering coefficient*: the ratio of actual `n1_edges` to the maximum number of possible `n1_edges`. It ranges from zero (no neighbor links to any other neighbor) to one (every neighbor links to every other neighbor). A moderately high clustering coefficient indicates a cohesive community. A low clustering coefficient could indicate widely dispersed interest (as it does with `@THE_REAL_SHAQ`), or it could indicate the kind of inorganic community that a spam account would engender.

Local properties at global scale

We've calculated community metrics at the scale of a node, an edge, and a neighborhood. How about the whole globe? There's not enough space here to cover it, but you can simultaneously determine the clustering coefficient for every node by generating

every "triangle" in the graph. For each user, comparing the number of triangles they belong to with their degree leads to the clustering coefficient.

Be careful, though! Remember the wide variation in node degree discussed earlier? Recklessly extending the previous method will lead to an explosion of data—pop star `@britneyspears` (5.2 million followers, 420,000 following as of July 2010) or `@WholeFoods` (1.7 million followers, 600,000 following) will each generate trillions of entries. What's worse, because large communities have a sparse clustering coefficient, almost all of these will be thrown away! There is a very elegant way to do this on the full graph,[16] but always keep in mind what the real world says about the problem. If you're willing to assert that `@britneyspears` isn't *really* friends with 420,000 people, you can keep only the strong links. Weight each edge (by number of replies, whether it's symmetric, and so on), and set limits on the number of links from any node. This sharply reduces the intermediate data size, yet still does a reasonable job of estimating cohesiveness.

—Philip (flip) Kromer, Infochimps

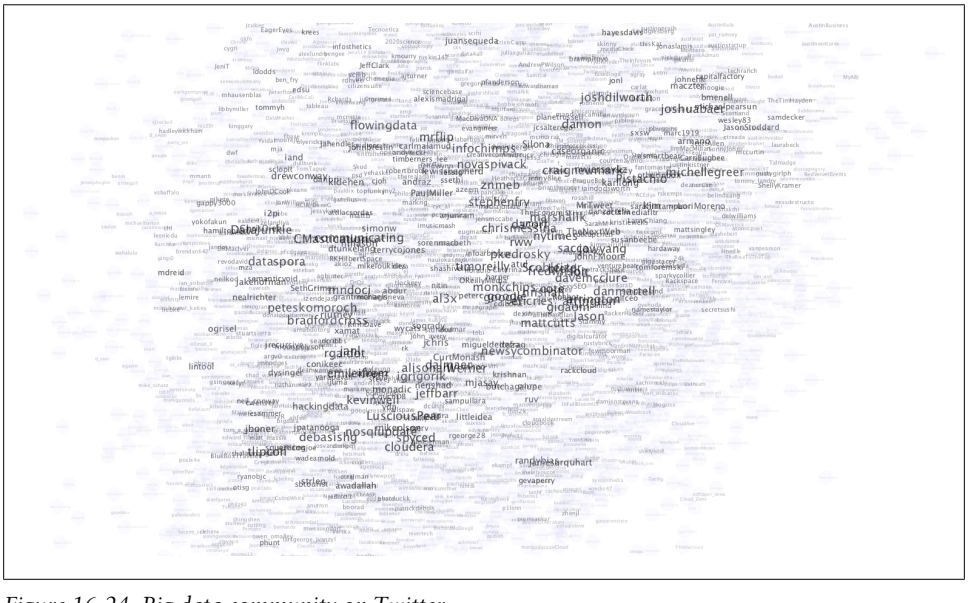


Figure 16-24. Big data community on Twitter

16. See *http://www.slideshare.net/ydn/3-xxl-graphalgohadoopsummit2010*. Sergei Vassilvitskii (`@vsergei`) and Jake Hofman (`@jakehofman`) of Yahoo! Research solve several graph problems by very intelligently throwing away most of the graph.