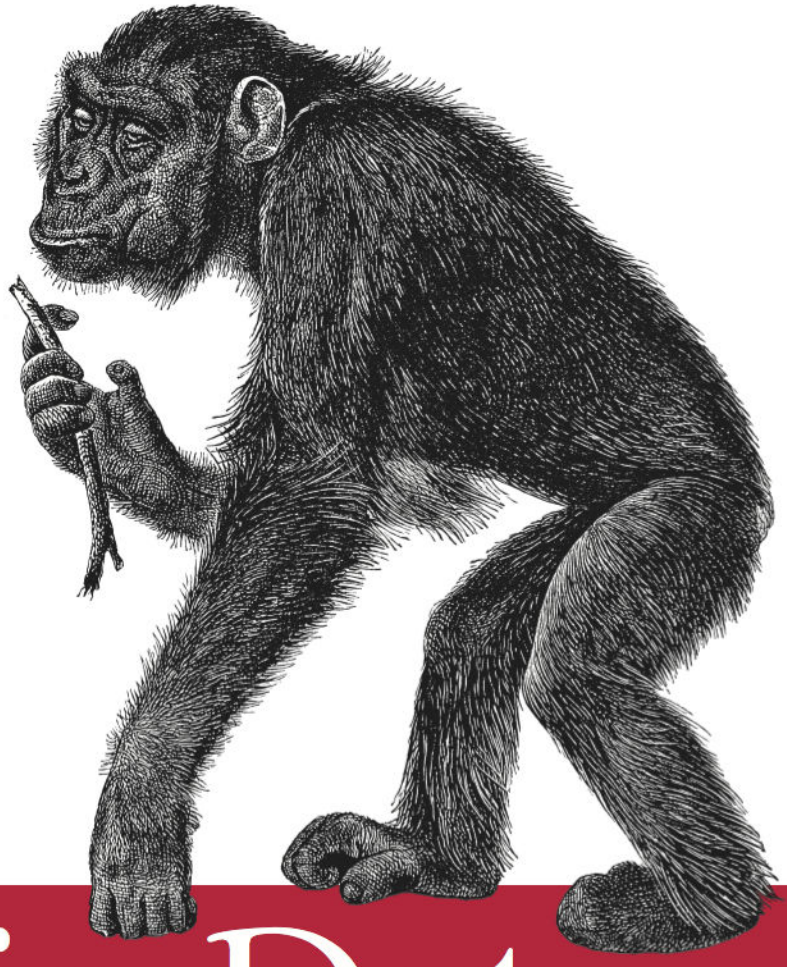


Big Data for Chimps

O'REILLY®

Philip Kromer

Big Data for Chimps

Philip (flip) Kromer

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Big Data for Chimps

by Philip (flip) Kromer

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*.

Editor: Amy Jollymore and Meghan Blanchette

: First Edition

Revision History for the First Edition:

2014-01-25: Working Draft

See *http://oreilly.com/catalog/errata.csp?isbn=* for release details.

[e]

Table of Contents

Preface

Mission Statement

Big Data for Chimps will:

1. Explain a practical, actionable view of big data, centered on tested best practices as well as give readers street fighting smarts with Hadoop
2. Readers will also come away with a useful, conceptual idea of big data; big data in its simplest form is a small cluster of well-tagged information that sits upon a central pivot, and can be manipulated through various shifts and rotations with the purpose of delivering insights ("Insight is data in context"). Key to understanding big data is scalability: infinite amounts of data can rest upon infinite pivot points (Flip - is that accurate or would you say there's just one central pivot - like a Rubic's cube?)
3. Finally, the book will contain examples with real data and real problems that will bring the concepts and applications for business to life.

About

What this book covers

Big Data for Chimps shows you how to solve important hard problems using simple, fun, elegant tools.

Geographic analysis is an important hard problem. To understand a disease outbreak in Europe, you need to see the data from Zurich in the context of Paris, Milan, Frankfurt and Munich; but to understand the situation in Munich requires context from Zurich, Prague and Vienna; and so on. How do you understand the part when you can't hold the whole world in your hand?

Finding patterns in massive event streams is an important hard problem. Most of the time, there aren't earthquakes — but the patterns that will let you predict one in advance lie within the data from those quiet periods. How do you compare the trillions of subsequences in billions of events, each to each other, to find the very few that matter? Once you have those patterns, how do you react to them in real-time?

We've chosen case studies anyone can understand that generalize to problems like those and the problems you're looking to solve. Our goal is to equip you with:

- How to think at scale — equipping you with a deep understanding of how to break a problem into efficient data transformations, and of how data must flow through the cluster to effect those transformations.
- Detailed example programs applying Hadoop to interesting problems in context
- Advice and best practices for efficient software development

All of the examples use real data, and describe patterns found in many problem domains:

- Statistical Summaries
- Identify patterns and groups in the data
- Searching, filtering and herding records in bulk
- Advanced queries against spatial or time-series data sets.

The emphasis on simplicity and fun should make this book especially appealing to beginners, but this is not an approach you'll outgrow. We've found it's the most powerful and valuable approach for creative analytics. One of our maxims is "Robots are cheap, Humans are important": write readable, scalable code now and find out later whether you want a smaller cluster. The code you see is adapted from programs we write at Infochimps to solve enterprise-scale business problems, and these simple high-level transformations (most of the book) plus the occasional Java extension (chapter XXX) meet our needs.

Many of the chapters have exercises included. If you're a beginning user, I highly recommend you work out at least one exercise from each chapter. Deep learning will come less from having the book in front of you as you *read* it than from having the book next to you while you **write** code inspired by it. There are sample solutions and result datasets on the book's website.

Feel free to hop around among chapters; the application chapters don't have large dependencies on earlier chapters.

Who This Book Is For

We'd like for you to be familiar with at least one programming language, but it doesn't have to be Ruby. Familiarity with SQL will help a bit, but isn't essential.

Most importantly, you should have an actual project in mind that requires a big data toolkit to solve — a problem that requires scaling out across multiple machines. If you don't already have a project in mind but really want to learn about the big data toolkit, take a quick browse through the exercises. At least a few of them should have you jumping up and down with excitement to learn this stuff.

Who This Book Is Not For

This is not "Hadoop the Definitive Guide" (that's been written, and well); this is more like "Hadoop: a Highly Opinionated Guide". The only coverage of how to use the bare Hadoop API is to say "In most cases, don't". We recommend storing your data in one of several highly space-inefficient formats and in many other ways encourage you to willingly trade a small performance hit for a large increase in programmer joy. The book has a relentless emphasis on writing **scalable** code, but no content on writing **performant** code beyond the advice that the best path to a 2x speedup is to launch twice as many machines.

That is because for almost everyone, the cost of the cluster is far less than the opportunity cost of the data scientists using it. If you have not just big data but huge data — let's say somewhere north of 100 terabytes — then you will need to make different tradeoffs for jobs that you expect to run repeatedly in production.

The book does have some content on machine learning with Hadoop, on provisioning and deploying Hadoop, and on a few important settings. But it does not cover advanced algorithms, operations or tuning in any real depth.

Hello, Early Releasers

Hello and thanks, courageous and farsighted early released-to'er! We want to make sure the book delivers value to you now, and rewards your early confidence by becoming a book you're proud to own.

Chimpanzee and Elephant

Starting with Chapter 2, you'll meet the zealous members of the Chimpanzee and Elephant Computing Company. Elephants have prodgious memories and move large heavy volumes with ease. They'll give you a physical analogue for using relationships to assemble data into context, and help you understand what's easy and what's hard in moving around massive amounts of data. Chimpanzees are clever but can only think about one thing at a time. They'll show you how to write simple transformations with a single

concern and how to analyze a petabytes data with no more than megabytes of working space.

Together, they'll equip you with a physical metaphor for how to work with data at scale.

Hadoop

In Doug Cutting's words, Hadoop is the "kernel of the big-data operating system". It's the dominant batch-processing solution, has both commercial enterprise support and a huge open source community, runs on every platform and cloud, and there are no signs any of that will change in the near term.

The code in this book will run unmodified on your laptop computer and on an industrial-strength Hadoop cluster. (Of course you will need to use a reduced data set for the laptop). You do need a Hadoop installation of some sort — Appendix (TODO: ref) describes your options, including instructions for running hadoop on a multi-machine cluster in the public cloud — for a few dollars a day you can analyze terabyte-scale datasets.

A Note on Ruby and Wukong

We've chosen Ruby for two reasons. First, it's one of several high-level languages (along with Python, Scala, R and others) that have both excellent Hadoop frameworks and widespread support. More importantly, Ruby is a very readable language — the closest thing to practical pseudocode we know. The code samples provided should map cleanly to those high-level languages, and the approach we recommend is available in any language.

In particular, we've chosen the Ruby-language Wukong framework. We're the principal authors, but it's open-source and widely used. It's also the only framework I'm aware of that runs on both Hadoop and Storm+Trident.

Helpful Reading

- Hadoop the Definitive Guide by Tom White is a must-have. Don't try to absorb its whole — the most powerful parts of Hadoop are its simplest parts — but you'll refer to often it as your applications reach production.
- Hadoop Operations by Eric Sammer — hopefully you can hand this to someone else, but the person who runs your hadoop cluster will eventually need this guide to configuring and hardening a large production cluster.
- "Big Data: principles and best practices of scalable realtime data systems" by Nathan Marz
- Patterns of MapReduce

What This Book Does Not Cover

We are not currently planning to cover Hive. The Pig scripts will translate naturally for folks who are already familiar with it. There will be a brief section explaining why you might choose it over Pig, and why I chose it over Hive. If there's popular pressure I may add a "translation guide".

This book picks up where the internet leaves off — apart from cheatsheets at the end of the book, I'm not going to spend any real time on information well-covered by basic tutorials and core documentation. Other things we do not plan to include:

- Installing or maintaining Hadoop
- we will cover how to design HBase schema, but not how to use HBase as *database*
- Other map-reduce-like platforms (disco, spark, etc), or other frameworks (MrJob, Scalding, Cascading)
- At a few points we'll use Mahout, R, D3.js and Unix text utils (cut/wc/etc), but only as tools for an immediate purpose. I can't justify going deep into any of them; there are whole O'Reilly books on each.

Feedback

- The source code for the book — all the prose, images, the whole works — is on github at `http://github.com/infochimps-labs/big_data_for_chimps`.
- Contact us! If you have questions, comments or complaints, the issue tracker *http://github.com/infochimps-labs/big_data_for_chimps/issues* is the best forum for sharing those. If you'd like something more direct, please email *meghan@oreilly.com* (the ever-patient editor) and *flip@infochimps.com* (your eager author). Please include both of us.

OK! On to the book. Or, on to the introductory parts of the book and then the book.

How this book is being written

I plan to push chapters to the publicly-viewable *Hadoop for Chimps* git repo as they are written, and to post them periodically to the Infochimps blog after minor cleanup.

We really mean it about the git social-coding thing — please comment on the text, file issues and send pull requests. However! We might not use your feedback, no matter how dazzlingly cogent it is; and while we are soliciting comments from readers, we are not seeking content from collaborators.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472 (707) 829-0515 (international or local)

To comment or ask technial questions about this book, send email to *bookquestions@oreilly.com*

To reach the authors:

Flip Kromer is @mrflip on Twitter

For comments or questions on the material, file a github issue at *http://github.com/infochimps-labs/big_data_for_chimps/issues*

CHAPTER 1

Insight comes from Data in Context

TODO: put in an interlude that is JT & Nanette meeting. (Told as a flashforward.) TODO: Get O'Reilly to do a promo version of just the interludes — "Big Data for Babies" — baby chimp on cover?

Data is worthless. Actually, it's worse than worthless: it requires money and effort to collect, store, transport and organize. Nobody wants data.

What's valuable is *insight* — summaries, patterns and connections that lead to deeper understanding and better decisions. And insight comes from synthesizing data in context. We can predict air flight delays by placing commercial flight arrival times in context with hourly global weather data (as we do in Chapter (REF)). Take the mountain of events in a large website's log files, and regroup using context defined by the paths users take through the site, and you'll illuminate articles of similar interest (see Chapter (REF)). In Chapter (REF), we'll dismantle the text of every article on Wikipedia, then reassemble the words from each article about a physical place into context groups defined by the topic's location — and produce insights into human language not otherwise quantifiable.

Within each of those examples are two competing forces that move them out of the domain of traditional data analysis and into the topic of this book: "big data" analytics: due to the volume of data, it is far too large to comfortably analyze on a single machine; and due to the comprehensiveness of the data, simple analytic methods are able to extract patterns not visible in the small.

Big Data: Tools to Solve the Crisis of Comprehensive Data

Let's take an extremely basic analytic operation: counting. To count the votes for a legislative bill, or Parent-Teacher association head, or what type of pizza to order, we gather the relevant parties into the same room at a fixed time and take a census of opintions. The logistics here are straightforward.

It is impossible, however, to count votes for the President of the United States this way. No conference hall is big enough to hold 300 million people; if there were, no roads are wide enough to get people to that conference hall; and even still the processing rate would not greatly exceed the rate at which voters come of age or die.

Once the volume of data required for synthesis exceeds some key limit of available computation — limited memory, limited network bandwith, limited time to prepare a relevant answer, or such — you're forced to fundamentally rework how you synthesize insight from data.

We conduct a presidential election by sending people to local polling places, distributed so that the participants to not need to travel far and numerous enough that the logistics of voting remain straightforward. At the end of day the vote totals from each polling place are summed to prepare the final result. This new approach doesn't completely discard the straightforward method (gathering people to the same physical location) that worked so well in the small. Instead, it calls in another local method (summing a table of numbers) and orchestrates the two so that the volume of people and data never exceeds what can be efficiently processed.

So our first definition of Big Data is a response to a crisis: "A collection of practical data analysis tools and processes that continue to scale even as the volume of data for justified synthesis exceeds some limit of available computation".

In Chapter 6 (REF) we'll map out the riotous diversity of tools in the Big Data ecosystem, Hadoop is the ubiquitous choice for processing batches of data at high Hadoop is the tool to use when you want to understand how patterns in data from your manufacturing devices corresponds to defective merchandise returned months later, or how patterns in patients' postoperative medical records correspond to the likelihood they'll be re-admitted with complications.

One solution to the big data crisis is high-performance supercomputing (HPC): push back the limits of computation with brute force. We could conduct our election by gathering supporters of one candidate on a set of cornfields in Iowa, supporters of the other on cornfields in Iowa, and using satellite imaging to tally the result. HPC solutions are exceptionally expensive, require the kind of technology seen only when military and industrial get complex, and though the traditional "all data is local" methods continue to work, they lose their essential straightforward flavor. A supercomputer is not one giant connected room, it's a series of largish rooms connected by very wide multidimensional hallways; HPC programmers have to constantly think about the motion of data among caches, processors, and backing store.

Hadoop's approach is effectively the opposite. Instead of full control over all aspects of computation and the illusion of data locality, Hadoop revokes almost all control over the motion of data and supports only one type of program, one that fits the "map / reduce" paradigm. Imagine a publisher that banned all literary forms except the haiku:

```
data flutters by
    elephants make sturdy piles
  context yields insight
```

Our Map/Reduce haiku illustrates Hadoop's template:

1. The Mapper portion of your script processes records, attaching a label to each.
2. Hadoop assembles those records into context groups according to their label.
3. The Reducer portion of your script processes those context groups and writes them to a data store or external system.

While it would be unworkable to have every novel, critical essay, or sonnet be composed of haikus, map/reduce is surprisingly more powerful. From this single primitive, we can construct the familiar relational operations (such as GROUPs and ROLLUPs) of traditional databases, many machine-learning algorithms, matrix and graph transformations and the rest of the advanced data analytics toolkit.

We will demonstrate map/reduce using Wukong, a thin layer atop Hadoop using the Ruby programming language. It's the most easily-readable way for us to demonstrate the patterns of data analysis, and you will be able to lift its content into the programming language of your choice [1]. It's also a powerful tool you won't grow out of — we perform code mixture is roughly 30% Wukong, 60% Pig (see the next paragraph) and 10% using Java to extend Pig.

The high-level Pig programming language has you describe the kind of full-table transformations familiar to database programmers (selecting filtered data, groups and aggregations, joining records from multiple tables). Pig carries out those transformations using efficient map/reduce scripts in Hadoop, based on optimized algorithms you'd otherwise have to reimplement or do without. To hit the sweet spot of "common things are simple, complex things remain possible", you can extend Pig with User-Defined Functions (UDFs), covered in chapter (REF).

Earlier, we defined insight as deeper understanding and better decisions. Hadoop's ability to process data of arbitrary scale, combined with our increasing ability to comprehensively instrument every aspect of an enterprise, represent a fundamental improvement in how we expose patterns and the range of human endeavors available for pattern mining.

But a funny thing happens as an organization's Hadoop investigations start to pay off: they realize they don't just want a deeper understanding of the patterns, they want to act on those patterns and make prompt decisions. The factory owner will want to stop

1. In the spirit of this book's open-source license, if an eager reader submits a "translation" of the example programs into the programming language of their choice we would love to fold it into in the example code repository and acknowledge the contribution in future printings.

the manufacturing line when signals predict later defects; the hospital will want to have a social worker follow up with patients unlikely to fill their postoperative medications. Just in time, a remarkable new capability has entered the core Big Data toolset: Streaming Analytics.

Streaming Analytics gets you *fast relevant insight* to go with Hadoop's *deep global insight*. Storm+Trident (the clear frontrunner toolkit) can process data with low latency and exceptional throughput (we've benchmarked it at half a million events per second); it can perform complex processing in Java, Ruby and more; it can hit remote APIs or databases with high concurrency.

This triad — Batch Analytics, Stream Analytics, and Scalable Datastores — are the three legs of the Big Data toolset. Together they let you analyze data at terabytes and petabytes, data at milliseconds, and data from ponderously many sources.

Big Data: Tools to Capitalize on the Opportunity of Comprehensive Data

That's the tools side of the Big Data ecosystem. What about the algorithms?

One common pattern for working with Big Data is to (a) assemble comprehensive data about the system, identify the data's structure and connectivity; (b) apply generic methods that use only that structure and connectivity, not its meaning, to expose patterns in the data; (c) interpret those patterns back in the system's domain.

(TODO need more here)

Peter Norvig (Google's Director of Research) calls this the "Unreasonable Effectiveness of Data" ("On the Unreasonable effectiveness of data").

This proposition is sure to cause barroom brawls at scientific conferences for years to come, because it advocates another path to truth that *does not follow* the Scientific Method. Roughly speaking, the scientific method has you (a) use a simplified model of the universe to make falsifiable predictions; (b) test those predictions in controlled circumstances; (c) use established truths to bound any discrepancies [2]. Under this paradigm, data is non-comprehensive: scientific practice demands you carefully control experimental conditions, and the whole point of the model is to strip out all but the reductionistically necessary parameter. A large part of the analytic machinery acts to account for discrepancies from sampling (too little comprehensiveness) or discrepancies from "extraneous" effects (too much comprehensiveness). If those discrepancies are modest, the model is judged to be valid. This paradigm is regarded as the only

2. plus (d) a secret dose of our sense of the model's elegance

acceptably rigorous way to admit a simplified representation of the world into the canon of truth.

Simple Exploration

(TODO transplant intro to UFO sighting data here) (TODO introduce this in context of reindeer?)

Sad to say, but many of the sighting reports are likely to be bogus. To eliminate sightings that lack a detailed description, we can filter out records whose description Field is shorter than 80 characters:

```
TODO code
```

A key activity in a Big Data exploration is summarizing big datasets into a comprehensible smaller ones. Each sighting has a field giving the shape of the flying object: cigar, disk, etc. This script will tell us how many sightings there are for each craft type:

```
LOAD sightings
GROUP sightings BY craft type
FOREACH cf_sightings GENERATE COUNTSTAR(sightings)
STORE cf_counts INTO 'out/geo/ufo_sightings/craft_type_counts';
```

We can make a little travel guide for the sightings by amending each sighting with the Wikipedia article about its place. The JOIN operator matches records from different tables based on a common key:

```
TODO pseudocode
```

This yields the following output:

Of course this would make a much better travel guide if it held not just the one article about the general location but a set of prominent nearby places of interest. We'll show you how to do a nearby-ness query in the Geodata chapter (REF), and how to attach a notion of "prominence" in the event log chapter (REF).

Grouping and Sorting: Analyzing UFO Sightings with Pig

While those embarassingly parallel, Map-only jobs are useful, Hadoop also shines when it comes to filtering, grouping, and counting items in a dataset. We can apply these techniques to build a travel guide of UFO sightings across the continental US.

Whereas our last example used the wukong framework, this time around we'll use another Hadoop abstraction, called Pig. [3] Pig's claim to fame is that it gives you full Hadoop

3. *http://pig.apache.org*

power, using a syntax that lets you think in terms of data flow instead of pure Map and Reduce operations.

The example data included with the book includes a data set from the National UFO Reporting Center, containing more than 60,000 documented UFO sightings [4].

Now it's sad to say, but many of the sighting reports are likely to be bogus. To eliminate sightings that lack a detailed description, we can filter out records whose description Field is shorter than 80 characters:

```
TODO code
```

A key activity in a Big Data exploration is summarizing big datasets into a comprehensible smaller ones. Each sighting has a field giving the shape of the flying object: cigar, disk, etc. This script will tell us how many sightings there are for each craft type:

```
LOAD sightings
GROUP sightings BY craft type
FOREACH cf_sightings GENERATE COUNTSTAR(sightings)
STORE cf_counts INTO 'out/geo/ufo_sightings/craft_type_counts';
```

We can make a little travel guide for the sightings by amending each sighting with the Wikipedia article about its place. The JOIN operator matches records from different tables based on a common key:

```
TODO pseudocode
```

This yields the following output:

```
TODO output
```

This travel guide is a bit lame, but of course we can come up with all sorts of ways to improve it. For instance, a proper guide would hold not just the one article about the general location, but a set of prominent nearby places of interest. These notions crop up in many different problems, so later in the book we'll show you how to do a nearbyness query (in the Geodata chapter (REF)), and how to attach a notion of "prominence" (in the event log chapter (REF)), and much more.

4. For our purposes, although sixty thousand records are too small to justify Hadoop on their own, it's the perfect size to learn with.

Outline from outline (in Google Docs)

1. Intro to Hadoop
    - C&E translate Shakespeare: distributed computation
    - Simple Mechanics of Using Hadoop
    - embarrassingly parallel job — igpay atinlay
    - mappers at moderate detail

CHAPTER 2
Hadoop Basics

Hadoop is a large and complex beast. There's a lot to learn before one can even begin to use the system, much less become meaningfully adept at doing so. While Hadoop has stellar documentation, not everyone is comfortable diving right in to the menagerie of Mappers, Reducers, Shuffles, and so on. For that crowd, we've taken a different route, in the form of a story.

Chimpanzee and Elephant Start a Business

A few years back, two friends — JT, a gruff silverback chimpanzee, and Nanette, a meticulous matriarch elephant — decided to start a business. As you know, Chimpanzees love nothing more than sitting at keyboards processing and generating text. Elephants have a prodigious ability to store and recall information, and will carry huge amounts of cargo with great determination. This combination of skills impressed a local publishing company enough to earn their first contract, so Chimpanzee and Elephant Corporation (C&E Corp for short) was born.

The publishing firm's project was to translate the works of Shakespeare into every language known to man, so JT and Nanette devised the following scheme. Their crew set up a large number of cubicles, each with one elephant-sized desk and one or more chimp-sized desks, and a command center where JT and Nanette can coordinate the action.

As with any high-scale system, each member of the team has a single responsibility to perform. The task of each chimpanzee is simply to read a set of passages and type out the corresponding text in a new language. JT, their foreman, efficiently assigns passages to chimpanzees, deals with absentee workers and sick days, and reports progress back to the customer. The task of each librarian elephant is to maintain a neat set of scrolls, holding either a passage to translate or some passage's translated result. Nanette serves

as chief librarian. She keeps a card catalog listing, for every book, the location and essential characteristics of the various scrolls that maintain its contents.

When workers clock in for the day, they check with JT, who hands off the day's translation manual and the name of a passage to translate. Throughout the day the chimps radio progress reports in to JT; if their assigned passage is complete, JT will specify the next passage to translate.

If you were to walk by a cubicle mid-workday, you would see a highly-efficient interplay between chimpanzee and elephant, ensuring the expert translators rarely had a wasted moment. As soon as JT radios back what passage to translate next, the elephant hands it across. The chimpanzee types up the translation on a new scroll, hands it back to its librarian partner and radios for the next passage. The librarian runs the scroll through a fax machine to send it to two of its counterparts at other cubicles, producing the redundant copies Nanette's scheme requires.

The fact that each chimpanzee's work is independent of any other's — no interoffice memos, no meetings, no requests for documents from other departments — made this the perfect first contract for the Chimpanzee & Elephant, Inc. crew. JT and Nanette, however, were cooking up a new way to put their million-chimp army to work, one that could radically streamline the processes of any modern paperful office [1]. JT and Nanette would soon have the chance of a lifetime to try it out for a customer in the far north with a big, big problem.

Map-only Jobs: Process Records Individually

Having read that short allegory, you've just learned a lot about how Hadoop operates under the hood. We can now use it to walk you through some examples. This first example uses only the *Map* phase of MapReduce, to take advantage of what some people call an "embarrassingly parallel" problem.

We might not be as clever as JT's multilingual chimpanzees, but even we can translate text into a language we'll call *Igpay Atinlay* [2]. For the unfamiliar, here's how to translate standard English into Igpay Atinlay:

1. Some chimpanzee philosophers have put forth the fanciful conceit of a "paper-less" office, requiring impossibilities like a sea of electrons that do the work of a chimpanzee, and disks of magnetized iron that would serve as scrolls. These ideas are, of course, pure lunacy — right up there with the foolish proposal one could harness superheated water orfg controlled explosions of marsh gas to replace the motive power of an Elephant!

2. Sharp-eyed readers will note that this language is really called *Pig Latin*. That term has another name in the Hadoop universe, though, so we've chosen to call it Igpay Atinlay — Pig Latinizing the term "Pig Latin" — for this example.

- If the word begins with a consonant-sounding letter or letters, move them to the end of the word adding "ay": "happy" becomes "appy-hay", "chimp" becomes "imp-chay" and "yes" becomes "es-yay".
- In words that begin with a vowel, just append the syllable "way": "another" becomes "another-way", "elephant" becomes "elephant-way".

Igpay Atinlay translator, actual version is our first Hadoop job, a program that translates plain text files into Igpay Atinlay. It's written in Wukong, a simple library to rapidly develop big data analyses. Like the chimpanzees, it is single-concern: there's nothing in there about loading files, parallelism, network sockets or anything else. Yet you can run it over a text file from the commandline — or run it over petabytes on a cluster (should you for whatever reason have a petabyte of text crying out for pig-latinizing).

Igpay Atinlay translator, actual version.

```
CONSONANTS   = "bcdfghjklmnpqrstvwxz"
UPPERCASE_RE = /[A-Z]/
PIG_LATIN_RE = %r{
\b                      # word boundary
([#{CONSONANTS}]*)  # all initial consonants
([\w\']+)               # remaining wordlike characters
}xi

each_line do |line|
latinized = line.gsub(PIG_LATIN_RE) do
head, tail = [$1, $2]
head    = 'w' if head.blank?
tail.capitalize! if head =~ UPPERCASE_RE
"#{tail}-#{head.downcase}ay"
end
yield(latinized)
end
```

Igpay Atinlay translator, pseudocode.

```
for each line,
recognize each word in the line and change it as follows:
separate the head consonants (if any) from the tail of the word
if there were no initial consonants, use 'w' as the head
give the tail the same capitalization as the word
change the word to "{tail}-#{head}ay"
end
emit the latinized version of the line
end
```

Ruby helper

- The first few lines define "regular expressions" selecting the initial characters (if any) to move. Writing their names in ALL CAPS makes them be constants.
- Wukong calls the `each_line do ... end` block with each line; the `|line|` part puts it in the `line` variable.
- the `gsub` ("globally substitute") statement calls its `do ... end` block with each matched word, and replaces that word with the last line of the block.
- `yield(latinized)` hands off the `latinized` string for wukong to output

It's best to begin developing jobs locally on a subset of data. Run your Wukong script directly from your terminal's commandline:

```
wu-local examples/text/pig_latin.rb data/magi.txt -
```

The `-` at the end tells wukong to send its results to standard out (STDOUT) rather than a file — you can pipe its output into other unix commands or Wukong scripts. In this case, there is no consumer and so the output should appear on your terminal screen. The last line should read:

```
Everywhere-way ey-thay are-way isest-way. Ey-thay are-way e-thay agi-may.
```

That's what it looks like when a `cat` is feeding the program data; let's see how it works when an elephant sets the pace.

Are you running on a cluster?

If you've skimmed Hadoop's documentation already, you've probaby seen the terms *fully-distributed, pseudo-distributed,* and *local,* bandied about. Those describe different ways to setup your Hadoop cluster, and they're relevant to how you'll run the examples in this chapter.

In short: if you're running the examples on your laptop, during a long-haul flight, you're likely running in local mode. That means all of the computation work takes place on your machine, and all of your data sits on your local filesystem.

On the other hand, if you have access to a cluster, your jobs run in fully-distributed mode. All the work is farmed out to the cluster machines. In this case, your data will sit in the cluster's filesystem called HDFS.

Run the following commands to copy your data to HDFS:

```
hadoop fs -mkdir ./data
hadoop fs -put   wukong_example_data/text ./data/
```

These commands understand `./data/text` to be a path on the HDFS, not your local disk; the dot `.` is treated as your HDFS home directory (use it as you would ~ in Unix.). The `wu-put` command, which takes a list of local paths and copies them to the HDFS, treats its final argument as an HDFS path by default, and all the preceding paths as being local.

(Note: if you don't have access to a Hadoop cluster, Appendix 1 (REF) lists resources for acquiring one.)

Run the Job

First, let's test on the same tiny little file we used at the commandline.

```
wukong launch examples/text/pig_latin.rb ./data/text/magi.txt ./output/latinized_magi
```

CODE: something about what the reader can expect to see on screen

While the script outputs a bunch of happy robot-ese to your screen, open up the jobtracker in your browser window (see the sidebar REF). The job should appear on the jobtracker window within a few seconds — likely in more time than the whole job took to complete.

```
SIDEBAR:  The Jobtracker Console

When you are running on a distributed Hadoop cluster, the jobtracker offers a built-in console for

Further down the page, you will see sections for running, completed and failed jobs. The last part

Clicking on the job name will take you to a page summarizing that job. We will walk through the pa
```

```
    You should observe slow progress through the shuffle stage beginning part way through the Map phas

    The job is not completely finished when the last Reducer hits 100 percent -- there remains a Commi

    The main thing to watch for in the Reduce phase is rapid progress by most of your Reducers and pai

    Do not put too much faith in the "percent complete" numbers for the job as a whole and even for it

    Above  the job progress bar graphs is a hot mess of a table showing all sorts of metrics about you

    Above that section is a smaller table giving the count of pending, running, complete, killed and f

    Lastly, near the top of a page is a long URL that ends with "job.xml". Do not go there now; it is
...

    SIDEBAR: How a job is born, the thumbnail version

    Apart from one important detail, the mechanics of how a job is born should never become interestin

    When you run `wukong ...` or `pig ...` or otherwise launch a Hadoop job, your local program contac

    As you have gathered, each Hadoop worker runs a tasktracker daemon to coordinate the tasks run by

    The one important detail to learn in all this is that _task trackers do not run your job, they onl
```

You can compare its output to the earlier by running

```
hadoop fs -cat ./output/latinized_magi/\*
```

That command, like the Unix 'cat' command, dumps the contents of a file to standard out, so you can pipe it into any other command line utility. It produces the full contents of the file, which is what you would like for use within scripts but if your file is hundreds of MB large, as HDFS files typically are, dumping its entire contents to your terminal screen is ill appreciated. We typically, instead, use the Unix 'head' command to limit its output (in this case, to the first ten lines).

```
hadoop fs -cat ./output/latinized_magi/\* | head -n 10
```

Since you wouldn't want to read a whole 10GB file just to see whether the right number of closing braces come at the end, there is also a `hadoop fs -tail` command that dumps the terminal one kilobyte of a file.

Here's what the head and tail of your output should contain:

```
CODE screenshot of hadoop fs -cat ./output/latinized_magi/\* | head -n 10
CODE screenshot of hadoop fs -tail ./output/latinized_magi/\*
```

See Progress and Results

Till now, we've run small jobs so you could focus on learning. Hadoop is built for big jobs, though, and it's important to understand how work flows through the system.

So let's run it on a corpus large enough to show off the power of distributed computing. Shakespeare's combined works are too small — at (CODE find size) even the prolific bard's lifetime of work won't make Hadoop break a sweat. Luckily, we've had a good slice of humanity typing thoughts into wikipedia for several years, and the corpus containing every single wikipedia article is enough to warrant Hadoop's power (and tsoris [3]).

```
wukong launch examples/text/pig_latin.rb ./data/text/wikipedia/wp_articles ./output/latinized_wiki
```

CODE screenshot of output, and fix up filenames

Visit the jobtracker console (see sidebar REF). The first thing you'll notice is how much slower this runs! That gives us a chance to demonstrate how to monitor its progress. (If your cluster is so burly the job finishes in under a minute or so, quit bragging and supply enough duplicate copies of the input to grant you time.) In the center of the Job Tracker's view of your job you will find a table that lists status of map and reduce tasks. The number of tasks pending (waiting to be run), running, complete, killed (terminated purposefully not by error) and failed (terminated due to failure).

The most important numbers to note are the number of running tasks (there should be some unless your job is finished or the cluster is congested) and the number of failed tasks (for a healthy job on a healthy cluster, there should never be any). Don't worry about killed tasks; for reasons we'll explain later on, it's OK if a few appear late in a job. We will describe what to do when there are failing attempts later in the section on debugging Hadoop jobs (REF), but in this case, there shouldn't be any. Clicking on the number of running Map tasks will take you to a window that lists all running attempts (and similarly for the other categories). On the completed tasks listing, note how long each attempt took; for the Amazon M3.xlarge machines we used, each attempt took about x seconds (CODE: correct time and machine size). There is a lot of information here, so we will pick this back up in chapter (REF), but the most important indicator is that your attempts complete in a uniform and reasonable length of time. There could be good reasons why you might find task 00001 to still be running after five minutes while other attempts have been finishing in ten seconds, but if that's not what you thought would happen you should dig deeper [4].

You should get in the habit of sanity-checking the number of tasks and the input and output sizes at each job phase for the jobs you write. In this case, the job should ultimately require x Map tasks, no Reduce tasks and on our x machine cluster, it completed in x minutes. For this input, there should be one Map task per HDFS block, x GB of input with the typical one-eighth GB block size, means there should be 8x Map tasks. Sanity

3. trouble and suffering

4. A good reason is that task 00001's input file was compressed in a non-splittable format and is 40 times larger than the rest of the files. A bad reason is that task 00001 is trying to read from a failing-but-not-failed datanode, or has a corrupted record that is sending the XML parser into recursive hell. The good reasons you can always predict from the data itself; otherwise assume it's a bad reason

checking the figure will help you flag cases where you ran on all the data rather than the one little slice you intended or vice versa; to cases where the data is organized inefficiently; or to deeper reasons that will require you to flip ahead to chapter (REF).

Annoyingly, the Job view does not directly display the Mapper input data, only the cumulative quantity of data per source, which is not always an exact match. Still, the figure for HDFS bytes read should closely match the size given by 'Hadoop fs -du' (CODE: add paths to command).

You can also estimate how large the output should be, using the "Gift of the Magi" sample we ran earlier (one of the benefits of first running in local mode). That job had an input size of x bytes and an output size of y bytes, for an expansion factor of z, and there is no reason to think the expansion factor on the whole Wikipedia corpus should be much different. In fact, dividing the HDFS bytes written by the HDFS bytes read line shows an expansion factor of q.

We cannot stress enough how important it is to validate that your scripts are doing what you think they are. The whole problem of Big Data is that it is impossible to see your data in its totality. You can spot-check your data, and you should, but without independent validations like these you're vulnerable to a whole class of common defects. This habit — of validating your prediction of the job's execution — is not a crutch offered to the beginner, unsure of what will occur; it is a best practice, observed most diligently by the expert, and one every practitioner should adopt.

In the next chapter, you'll learn about map/reduce jobs — the full power of Hadoop's processing paradigm.. Let's start by joining JT and Nannette with their next client.

Chimpanzee and Elephant Save Christmas (pt 1)

It was holiday time at the North Pole, and letters from little boys and little girls all over the world flooded in as they always do. But one year several years ago, the world had grown just a bit too much. The elves just could not keep up with the scale of requests — Christmas was in danger! Luckily, their friends at the Elephant & Chimpanzee Corporation were available to help. Packing their typewriters and good winter coats, JT, Nanette and the crew headed to the Santaplex, the headquarters for toy manufacture at the North Pole. Here's what they found.

Letters Cannot be Stored with the Right Context for Toy-Making

As you know, each year children from every corner of the earth write to Santa to request toys, and Santa — knowing who's been naughty and who's been nice — strives to meet the wishes of every good little boy and girl who writes him. He employs a regular army of toymaker elves, each of whom specializes in certain kinds of toy: some elves make Action Figures and Dolls, others make Xylophones and Yo-Yos.

Under the elves' old system, as bags of mail arrived they were examined by an elven postal clerk and then hung from the branches of the Big Tree at the center of the Santaplex. Letters were organized on the tree according to the child's town, as the shipping department has a critical need to organize toys by their final delivery schedule. But the toymaker elves must know what toys to make as well, and so for each letter a postal clerk recorded its Big Tree coordinates in a ledger that was organized by type of toy.

So to retrieve a letter, a doll-making elf would look under "Doll" in the ledger to find the next letter's coordinates, then wait as teamster elves swung a big claw arm to retrieve it from the Big Tree. As JT readily observed, the mail couldn't be organized both by toy type and also by delivery location, and so this ledger system was a necessary evil. "The next request for Lego is as likely to be from Cucamonga as from Novosibirsk, and letters can't be pulled from the tree any faster than the crane arm can move!"

What's worse, the size of Santa's operation meant that the workbenches were very far from where letters came in. The hallways were clogged with frazzled elves running from Big Tree to workbench and back, spending as much effort requesting and retrieving letters as they did making toys. "Throughput, not Latency!" trumpeted Nanette. "For hauling heavy loads, you need a stately elephant parade, not a swarm of frazzled elves!"

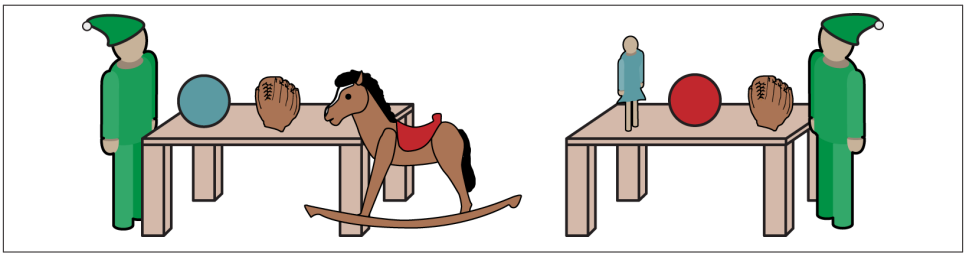


Figure 2-1. The elves' workbenches are meticulous and neat.

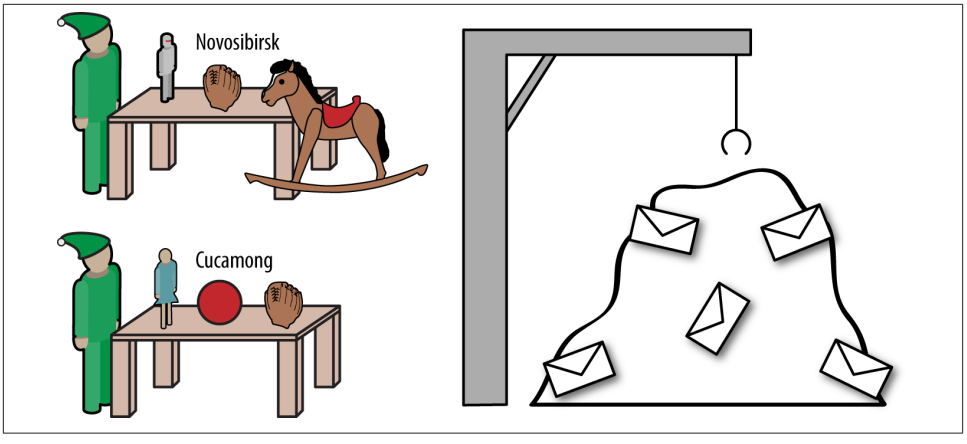


Figure 2-2. Little boys and girls' mail is less so.

Chimpanzees Process Letters into Labelled Toy Requests

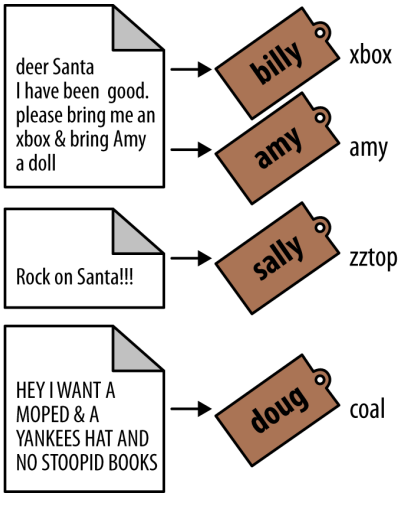
In marched Chimpanzee and Elephant, Inc, and set up a finite number of chimpanzees at a finite number of typewriters, each with an elephant desk-mate.

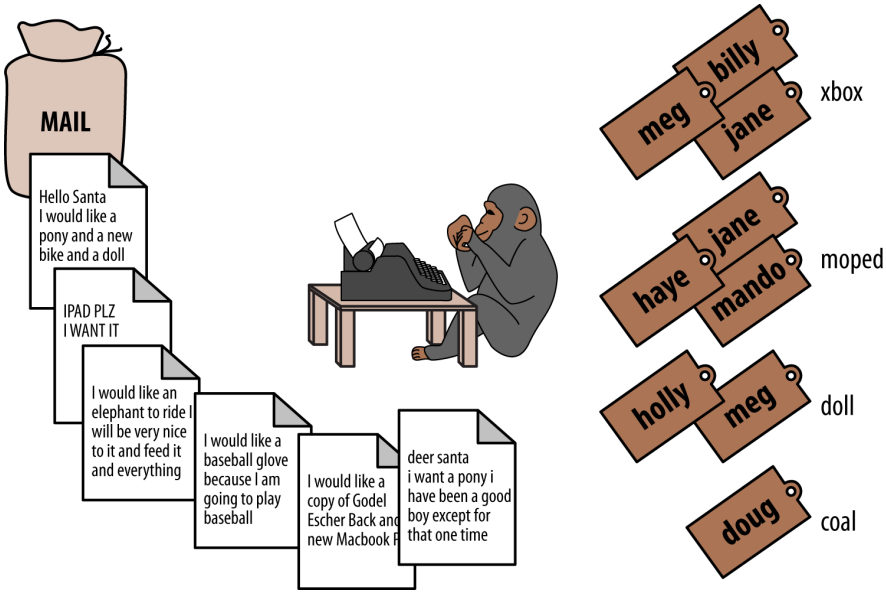
Postal clerks still stored each letter on the Big Tree (allowing the legacy shipping system to continue unchanged), but now also handed off bags holding copies of the mail. As she did with the translation passages, Nanette distributed these mailbags across the desks just as they arrived. The overhead of recording each letter in the much-hated ledger was no more, and the hallways were no longer clogged with elves racing to and fro.

The chimps' job was to take letters one after another from a mailbag, and fill out a toyform for each request. A toyform has a prominent label showing the type of toy, and a body with all the information you'd expect: Name, Nice/Naughty Status, Location, and so forth. You can see some examples here:

```
# Good kids, generates a toy for Julia and a toy for her brother

Deer SANTA                                          robot | type="optimus prime" recipient="Joe"
                                                    doll  | type="green hair"  recipient="Joe's sister"

I wood like a doll for me and
and an optimus prime robot for my
brother joe

I have been good this year

love julia

# Spam, no action

Greetings to you Mr Claus, I came to know
of you in my search for a  reliable and
reputable person to handle a very confidential
```

```
business transaction, which involves the
transfer of a huge sum of money...

# Frank is not only a jerk but a Yankees fan. He will get coal.

HEY SANTA I WANT A YANKEES HAT AND NOT             coal  | type="anthracite" recipient="Frank" rea
ANY DUMB BOOKS THIS YEAR

FRANK

---------------------------------------          # Spam, no action
```



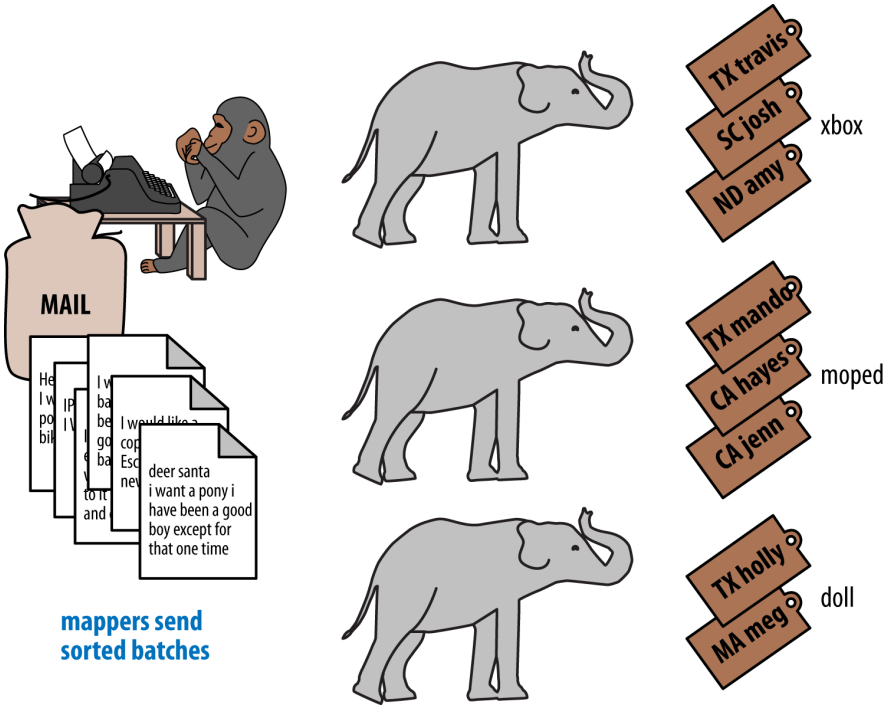


The first note, from a very good girl who is thoughtful for her brother, creates two toyforms: one for Joe's robot and one for Julia's doll. The second note is spam, so it creates no toyforms, while the third one yields a toyform directing Santa to put coal in his stocking.

Pygmy Elephants Carry Each Toyform to the Appropriate Workbench

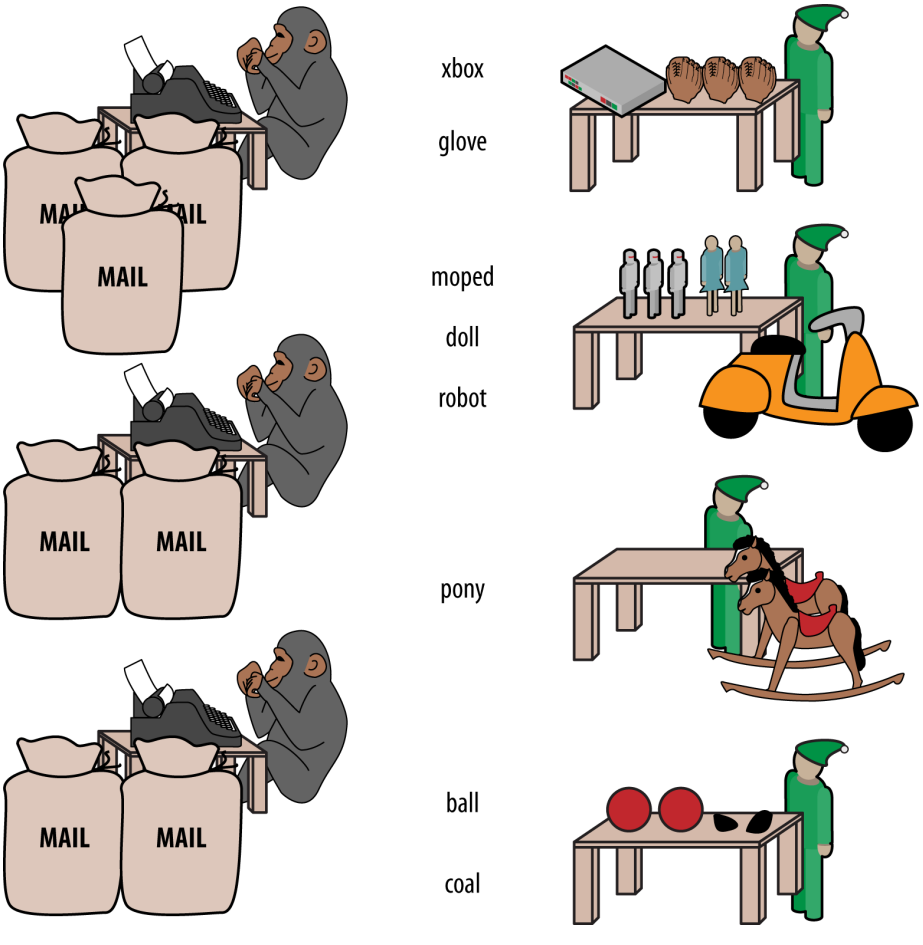
Here's the new wrinkle on top of the system used in the translation project. Next to every desk now stood a line of pygmy elephants, each dressed in a capes that listed the types of toy it would deliver. Each desk had a pygmy elephant for Archery Kits and Dolls, another one for Xylophones and Yo-Yos, and so forth — matching the different specialties of toymaker elves.

As the chimpanzees would work through a mail bag, they'd place each toyform into the basket on the back of the pygmy elephant that matched its type. At the completion of a bag, the current line of elephants would march off to the workbenches, and behind them a new line of elephants would trundle into place. What fun!



Finally, the pygmy elephants would march through the now-quiet hallways to the toy shop floor, each reporting to the workbench that matched its toy types. So the Archery Kit/Doll workbench had a line of pygmy elephants, one for every Chimpanzee&Elephant desk; similarly the Xylophone/Yo-Yo workbench, and all the rest.

Toymaker elves now began producing a steady stream of toys, no longer constrained by the overhead of walking the hallway and waiting for Big-Tree retrieval on every toy.



CHAPTER 3

Chimpanzee and Elephant Save Christmas

Outline

Tell about map/reduce, but don't yet worry about partition/group/secondary sort

03a: story

03b: * Example: group & aggregate on UFO visit stats - mapper and its output - reducer and its output * sidebar on "why hadoop vs databases"

- map/reduce haiku - admits to notion of group sort, but doesn't mention it in detail
- mappers at lightest detail or mappers at modest detail
- reducers at lightest detail
- ?? what mapper / reducer / hadoop do, phenomenologically

03c: * interlude part 2

03d: specializations of the reducer * code example: join: walk through code, makes clear the idea of sort + naming groups (explain how sorting is essential to how join works) * clarify concepts of partition, group, secondary sort * see effects of over/underspecifying partitions and group keys * Hadoop contract — casually formal description of the contract * how Hadoop manages mid-stream data: mapper spills, partitioners, reducers receive data and merge-sort it, reducers read records, reducers write output data and commit

In the previous chapter, you worked with the simple-as-possible Pig Latin script, which let you learn the mechanics of running Hadoop jobs, understand the essentials of the HDFS, and appreciate its scalability. It is an example of an "embarrassingly parallel" problem: each record could be processed individually, just as they were organized in the source files.

Hadoop's real power comes from the ability to process data in context, using what's known as the Map/Reduce paradigm. Every map/reduce job is a program with the same three phases. In the first phase, your program processes its input in any way you see fit, emitting labelled output records. In the second phase, Hadoop groups and sorts those records according to their labels. Finally, your program processes each group and Hadoop stores its output. That grouping-by-label part is where the magic lies: it ensures that no matter where the relevant records started, they arrive at the same place in a predictable manner, ready to be synthesized.

We'll open the chapter with a straightforward example map/reduce program: aggregating records from a dataset of Unidentified Flying Object sightings to find out when UFOs are most likely to appear.

Next, we'll outline how a map/reduce dataflow works — first with a physical analogy provided by our friends at Elephant and Chimpanzee inc, and then in moderate technical detail.

Summarizing UFO Sightings using Map/Reduce===

Santa Claus and his elves are busy year-round, but Santa's flying reindeer have few responsibilities outside the holiday season. As flying objects themselves, they spend a good part of their multi-month break is spent pursuing their favorite hobby: UFOlogy (the study of Unidentified Flying Objects and the search for extraterrestrial civilization). So you can imagine how excited they were to learn about the data set of more than 60,000 documented UFO sightings we worked with in the first chapter.

Sixty thousand sightings is much higher than a reindeer can count (only four hooves!), so JT and Nanette occasionally earn a little karmic bonus with Santa Claus by helping the reindeer analyzing UFO data. We can do our part by helping our reindeer friends understand when, during the day, UFOs are most likely to be sighted.

UFO Sighting Data Model

The data model for a UFO sighting has fields for date of sighting and of report; human-entered location; duration; shape of craft; and eye-witness description.

```
class SimpleUfoSighting
  include Wu::Model
  field :sighted_at,   Time
  field :reported_at,  Time
  field :shape,        Symbol
  field :city, String
  field :state, String
  field :country, String
  field :duration_str, String
  field :location_str, String
```

```
  field :description,  String
end
```

Group the UFO Sightings by Time Bucket

The first request from the reindeer team is to organize the sightings into groups by the shape of craft, and to record how many sightings there are for each shape.

Mapper

In the Chimpanzee&Elephant world, a chimp had the following role:

- reads and understand each letter
- creates a new intermediate item having a label (the type of toy) and information about the toy (the work order)
- hands it to the elephants for delivery to the elf responsible for making that toy type.

We're going to write a Hadoop "mapper" that performs a similar purpose:

- reads the raw data and parses it into a structured record
- creates a new intermediate item having a label (the shape of craft) and information about the sighting (the original record).
- hands it to Hadoop for delivery to the reducer responsible for that group

The program looks like this:

```
mapper(:count_ufo_shapes) do
  consumes UfoSighting, from: json
  #
  process do |ufo_sighting|      # for each record
    record = 1                   # create a dummy payload,
    label  = ufo_sighting.shape  # label with the shape,
    yield [label, record]        # and send it downstream for processing
  end
end
```

You can test the mapper on the commandline:

```
$ cat ./data/geo/ufo_sightings/ufo_sightings-sample.json   |
    ./examples/geo/ufo_sightings/count_ufo_shapes.rb --map |
    head -n25 | wu-lign
 disk      1972-06-16T05:00:00Z 1999-03-02T06:00:00Z    Provo (south of), UT           disk
 sphere    1999-03-02T06:00:00Z 1999-03-02T06:00:00Z    Dallas, TX                     sphere
 triangle  1997-07-03T05:00:00Z 1999-03-09T06:00:00Z    Bochum (Germany),              triangle
 light     1998-11-19T06:00:00Z 1998-11-19T06:00:00Z    Phoenix (west valley), AZ      light
 triangle  1999-02-27T06:00:00Z 1999-02-27T06:00:00Z    San Diego, CA                  triangle
 triangle  1997-09-15T05:00:00Z 1999-02-17T06:00:00Z    Wedgefield, SC                 triangle
...
```

The output is simply the partitioning label (UFO shape), followed by the attributes of the signing, separated by tabs. The framework uses the first field to group/sort by default; the rest is cargo.

Reducer

Just as the pygmy elephants transported work orders to elves' workbenches, Hadoop delivers each record to the *reducer*, the second stage of our job.

```
reducer(:count_sightings) do
  def process_group(label, group)
    count = 0
    group.each do |record|           # on each record,
      count += 1                     #   increment the count
      yield record                   #   re-output the record
    end                              #
    yield ['# count:', label, count] # at end of group, summarize
  end
end
```

The elf at each workbench saw a series of work orders, with the guarantee that a) work orders for each toy type are delivered together and in order; and b) this was the only workbench to receive work orders for that toy type.

Similarly, the reducer receives a series of records, grouped by label, with a guarantee that it is the unique processor for such records. All we have to do here is re-emit records as they come in, then add a line following each group with its count. We've put a # at the start of the summary lines, which lets you easily filter them.

Test the full mapper-sort-reducer stack from the commandline:

```
$ cat ./data/geo/ufo_sightings/ufo_sightings-sample.json      |
    ./examples/geo/ufo_sightings/count_ufo_shapes.rb --map    | sort |
    ./examples/geo/ufo_sightings/count_ufo_shapes.rb --reduce | wu-lign

1985-06-01T05:00:00Z        1999-01-14T06:00:00Z    North Tonawanda, NY     chevron         1
1999-01-20T06:00:00Z        1999-01-31T06:00:00Z    Olney, IL               chevron         10
1998-12-16T06:00:00Z        1998-12-16T06:00:00Z    Lubbock, TX             chevron         3
# count:    chevron 3
1999-01-16T06:00:00Z        1999-01-16T06:00:00Z    Deptford, NJ            cigar           2
# count:    cigar   1
1947-10-15T06:00:00Z        1999-02-25T06:00:00Z    Palmira,                circle          1
1999-01-10T06:00:00Z        1999-01-11T06:00:00Z    Tyson's Corner, VA      circle          1
...
```

SIDEBAR Hadoop vs Traditional Databases

Fundamentally, the storage engine at the heart of a traditional relational database does two things: it holds all the records, and it maintains a set of indexes for lookups and other operations. To retrieve a record, it must consult the appropriate index to find the

location of the record, then load it from the disk. This is very fast for record-by-record retrieval, but becomes cripplingly inefficient for general high-throughput access. If the records are stored by location and arrival time (as the mailbags were on the Big Tree), then there is no "locality of access" for records retrieved by, say, type of toy — records for Lego will be spread all across the disk. With traditional drives, the disk's read head has to physically swing back and forth in a frenzy across the disk, and though the newer flash drives have smaller retrieval latency it's still far too high for bulk operations.

What's more, traditional database applications lend themselves very well to low-latency operations (such as rendering a webpage showing the toys you requested), but very poorly to high-throughput operations (such as requesting every single doll order in sequence). Unless you invest specific expertise and effort, you have little ability to organize requests for efficient retrieval. You either suffer a variety of non-locality and congestion based inefficiencies, or wind up with an application that caters to the database more than to its users. You can to a certain extent use the laws of economics to bend the laws of physics — as the commercial success of Oracle and Netezza show — but the finiteness of time, space and memory present an insoluble scaling problem for traditional databases.

Hadoop solves the scaling problem by not solving the data organization problem. Rather than insist that the data be organized and indexed as it's written to disk, catering to every context that could be requested. Instead, it focuses purely on the throughput case. TODO explain disk is the new tape It takes X to seek but

The typical Hadoop operation streams large swaths of data The locality

TODO: finish this content

The Map-Reduce Haiku

As you recall, the bargain that Map/Reduce proposes is that you agree to only write programs that fit this Haiku:

```
data flutters by
    elephants make sturdy piles
  context yields insight
```

More prosaically,

1. **process and label** — turn each input record into any number of labelled records
2. **sorted context groups** — hadoop groups those records uniquely under each label, in a sorted order
3. **synthesize (process context groups)** — for each group, process its records in order; emit anything you want.

The trick lies in the *group/sort* step: assigning the same label to two records in the *label* step ensures that they will become local in the reduce step.

The machines in stage 1 (*label*) are out of context. They see each record exactly once, but with no promises as to order, and no promises as to which one sees which record. We've *moved the compute to the data*, allowing each process to work quietly on the data in its work space.

As each pile of output products starts to accumulate, we can begin to group them. Every group is assigned to its own reducer. When a pile reaches a convenient size, it is shipped to the appropriate reducer while the mapper keeps working. Once the map finishes, we organize those piles for its reducer to process, each in proper order.

If you notice, the only time data moves from one machine to another is when the intermediate piles of data get shipped. Instead of monkeys flinging poo, we now have a dignified elephant parade conducted in concert with the efforts of our diligent workers.

TODO: mappers in lightest detail TODO: reducers in lightest detail TODO: ??HDFS is lightest detail??[— writes are to local datanode; replication is a thing; we won't say more than this til a few chapters from now.

Elephant and Chimpanzee Save Christmas pt 2: A Critical Bottleneck Emerges===

After a day or two of the new toyform process, Mrs. Claus reported dismaying news. Even though productivity was much improved over the Big-Tree system, it wasn't going to be enough to hit the Christmas deadline.

The problem was plain to see. Repeatedly throughout the day, workbenches would run out of parts for the toys they were making. The dramatically-improved efficiency of order handling, and the large built-up backlog of orders, far outstripped what the toy parts warehouse could supply. Various workbenches were clogged with Jack-in-the-boxes awaiting springs, number blocks awaiting paint and the like. Tempers were running high, and the hallways became clogged again with overloaded parts carts careening off each other. JT and Nanette filled several whiteboards with proposed schemes, but none of them felt right.

To clear his mind, JT wandered over to the reindeer ready room, eager to join in the cutthroat games of poker Rudolph and his pals regularly ran. During a break in the action, JT found himself idly sorting out the deck of cards by number, as you do to check that it is a regular deck of 52. (With reindeer, you never know when an extra ace or three will inexplicably appear at the table). As he did so, something in his mind flashed back to the unfinished toys on the assembly floor: mounds of number blocks, stacks of Jack-in-the-boxes, rows of dolls. Sorting the cards by number had naturally organized them

into groups by kind as well: he saw all the numbers in blocks in a run, followed by all the jacks, then the queens and the kings and the aces.

"Sorting is equivalent to grouping!" he exclaimed to the reindeers' puzzlement. "Sorry, fellas, you'll have to deal me out," he said, as he ran off to find Nanette.

The next day, they made several changes to the toy-making workflow. First, they set up a delegation of elvish parts clerks at desks behind the letter-writing chimpanzees, directing the chimps to hand a carbon copy of each toy form to a parts clerk as well. On receipt of a toy form, each parts clerk would write out a set of tickets, one for each part in that toy, and note on the ticket the ID of its toyform. These tickets were then dispatched by pygmy elephant to the corresponding section of the parts warehouse to be retrieved from the shelves.

Now, here is the truly ingenious part that JT struck upon that night. Before, the chimpanzees placed their toy forms onto the back of each pygmy elephant in no particular order. JT replaced these baskets with standing file folders — the kind you might see on an organized person's desk. He directed the chimpanzees to insert each toy form into the file folder according to the alphabetical order of its ID. (Chimpanzees are exceedingly dextrous, so this did not appreciably impact their speed.) Meanwhile, at the parts warehouse Nanette directed a crew of elvish carpenters to add a clever set of movable set of frames to each of the part carts. She similarly prompted the parts pickers to put each cart's parts in the place properly preserving the alphabetical order of their toyform IDs.

//// Perhaps a smaller sizing for the image? Amy////

After a double shift that night by the parts department and the chimpanzees, the toymakers arrived in the morning to find, next to each workbench, the pygmy elephants with their toy forms and a set of carts from each warehouse section holding the parts they'd need. As work proceeded, a sense of joy and relief soon spread across the shop.

The elves were now producing a steady stream of toys as fast as their hammers could fly, with an economy of motion they'd never experienced. Since both the parts and the toy forms were in the same order by toyform ID, as the toymakers would pull the next toy form from the file they would always find the parts for it first at hand. Pull the toy form for a wooden toy train and you would find a train chassis next in the chassis cart, small wooden wheels next in the wheel cart, and magnetic bumpers next in the small parts cart. Pull the toy form for a rolling duck on a string, and you would find instead, a duck chassis, large wooden wheels and a length of string at the head of their respective carts.

Not only did work now proceed with an unbroken swing, but the previously cluttered workbenches were now clear — their only contents were the parts immediately required to assemble the next toy. This space efficiency let Santa pull in extra temporary workers from the elves' Rivendale branch, who were bored with fighting orcs and excited to help out.

Toys were soon coming off the line at a tremendous pace, far exceeding what the elves had ever been able to achieve. By the second day of the new system, Mrs. Claus excitedly reported the news everyone was hoping to hear: they were fully on track to hit the Christmas Eve deadline!

And that's the story of how Elephant and Chimpanzee saved Christmas.

TODO-qem: goal is to introduce group-sort notion first, but not overburden user with distinction of partition-group-secondarysort; then do join example; then discuss the technical part.

Close Encounters of the Reindeer Kind (part 2)

Since our reindeer friends want to spend their summer months visiting the locations of various UFO sighting, they would like more information to help plan their trip. The Geonames dataset (REF) provides more than seven million well-described points of interest, so we can extend each UFO sighting whose location matches a populated place name with its longitude, latitude, population and more.

Your authors have additionally run the free-text locations — "Merrimac, WI" or "Newark, NJ (south of Garden State Pkwy)" — through a geolocation service to (where possible) add structured geographic information: longitude, latitude and so forth.

Put UFO Sightings And Places In Context By Location Name

When you are writing a Map/Reduce job, the first critical question is how to group the records in context for the Reducer to synthesize. In this case, we want to match every UFO sighting against the corresponding Geonames record with the same city, state and country, so the Mapper labels each record with those three fields. This ensures records with the same location name all are received by a single Reducer in a single group, just as we saw with toys sent to the same workbench or visits "sent" to the same time bucket. The Reducer will also need to know which records are sightings and which records are places, so we have extended the label with an "A" for places and a "B" for sightings. (You will see in a moment why we chose those letters.) While we are at it, we will also eliminate Geonames records that are not populated places.

```
(TODO code for UFO sighting geolocator mapper)
```

```
        class UfoSighting
          include Wu::Model
          field :sighted_at,   Time
          field :reported_at,  Time
          field :shape,        Symbol
          field :city, String
          field :state, String
          field :country, String
          field :duration_str, String
          field :location_str, String
```

```
  #
  field :longitude,  Float
  field :latitude,   Float
  field :city,  String
  field :region,  String
  field :country,  String
  field :population,  Integer
  field :quadkey,    String
      #
  field :description,  String
end
```

Extend The UFO Sighting Records In Each Location Co-Group With Place Data

Building a toy involved selecting, first, the toy form, then each of the corresponding parts, so the elephants carrying toy forms stood at the head of the workbench next to all the parts carts. While the first part of the label (the partition key) defines how records are grouped, the remainder of the label (the sort key) describes how they are ordered within the group. Denoting places with an "A" and sightings with a "B" ensures our Reducer always first receives the place for a given location name followed by the sightings. For each group, the Reducer holds the place record in a temporary variable and appends the places fields to those of each sighting that follows. Iin the happy case where a group holds both place and sightings, the Reducer iterates over each sighting. There are many places that match no UFO sightings; these are discarded. There are some UFO sightings without reconcilable location data; we will hold onto those but leave the place fields blank. Even if these groups had been extremely large, this matching required no more memory overhead than the size of a place record.

Partition, Group and Secondary Sort

TODO: make this make sense

As you've seen, the way that Hadoop forms groups is actually by sorting the records. It's time now to clearly separate the three fundamental locality operations Hadoop performs for you:

- *partition*:
  — data in the same partition must go to the same machine
- *group*:
  — data in the same group must be in the same partition
- *sort*:

the Elves' system is meant to evoke the liabilities of database and worker-queue based systems:

- setup and teardown of workstation == using latency code for a throughput process
  - — running the same code in a tight loop makes life easy for the CPU cache in low level languages…
  - — and makes it easy for the interpreter in high-level languages, especially JIT
- swinging the mail claw out to retrieve next work order == latency of seek on disk
- chimpanzees are dextrous == processing sorted data in RAM is very fast
- elves pull work orders in sequence: The chimpanzees call this a "merge sort", and the elves' memory a "sort buffer"

Playing with Partitions: How Partition, Group and Sort affect a Job

TODO-qem — expand the prose with sample code and output as you think reasonable

TODO-qem — determine whether to hand-hold here, or add code examples

TODO: make this use the UFO data instead (pageview example won't be introduced until ch. 4 or 5.

Here's another version of the script to total wikipedia pageviews. We've modified the mapper to emit separate fields for the century, year, month, day and hour (you wouldn't normally do this; we're trying to prove a point). The reducer intends to aggregate the total pageviews across all pages by year and month: a count for December 2010, for January 2011, and so forth. We've also directed it to use twenty reducers, enough to illustrate a balanced distribution of reducer data.

Run the script on the subuniverse pageview data with `--partition_keys=3 --sort_keys=3` (TODO check params), and you'll see it use the first three keys (century/year/month) as both partition keys and sort keys. Each reducer's output will tend to have months spread across all the years in the sample, and the data will be fairly evenly distributed across all the reducers. In our runs, the `-00000` file held the months of (TODO insert observed months), while the `-00001` file held the months of (TODO insert observed months); all the files were close to (TODO size) MB large. (TODO consider updating to "1,2,3" syntax, perhaps with a gratuitous randomizing field as well. If not, make sure wukong errors on a partition_keys larger than the sort_keys). Running with `--partition_keys=3 --sort_keys=4` doesn't change anything: the `get_key` method in this particular reducer only pays attention to the century/year/month, so the ordering within the month is irrelevant.

Running it instead with `--partition_keys=2 --sort_keys=3` tells Hadoop to *partition* on the century/year, but do a secondary sort on the month as well. All records that

share a century and year now go to the same reducer, while the reducers still see months as continuous chunks. Now there are only six (or fewer) reducers that receive data — all of 2008 goes to one reducer, similarly 2009, 2010, and the rest of the years in the dataset. In our runs, we saw years X and Y (TODO adjust reducer count to let us prove the point, insert numbers) land on the same reducer. This uneven distribution of data across the reducers should cause the job to take slightly longer than the first run. To push that point even farther, running with `--partition_keys=1 --sort_keys=3` now partitions on the century — which all the records share. You'll now see 19 reducers finish promptly following the last mapper, and the job should take nearly twenty times as long as with `--partition_keys=3`.

Finally, try running it with `--partition_keys=4 --sort_keys=4`, causing records to be partitioned by century/year/month/day. Now the days in a month will be spread across all the reducers: for December 2010, we saw `-00000` receive X, Y and `-00001` receive X, Y, Z; out of 20 reducers, X of them received records from that month (TODO insert numbers). Since our reducer class is coded to aggregate by century/year/month, each of those reducers prepared its own meaningless total pageview count for December 2010, each of them a fraction of the true value. You must always ensure that all the data you'll combine in an aggregate lands on the same reducer.

Hadoop's Contract

We will state very precisely what Hadoop guarantees, so that you can both attach a rigorous understanding to the haiku-level discussion and see how *small* the contract is. This formal understanding of the contract is very useful for reasoning about how Hadoop jobs work and perform.

Hadoop imposes a few seemingly-strict constraints and provides a very few number of guarantees in return. As you're starting to see, that simplicity provides great power and is not as confining as it seems. You can gain direct control over things like partitioning, input splits and input/output formats. We'll touch on a very few of those, but for the most part this book concentrates on using Hadoop from the outside — (REF) *Hadoop: The Definitive Guide* covers this stuff (definitively).

The Mapper Guarantee

The contract Hadoop presents for a map task is simple, because there isn't much of one. Each mapper will get a continuous slice (or all) of some file, split at record boundaries, and in order within the file. You won't get lines from another input file, no matter how short any file is; you won't get partial records; and though you have no control over the processing order of chunks ("file splits"), within a file split all the records are in the same order as in the original file.

For a job with no reducer — a "mapper-only" job — you can then output anything you like; it is written straight to disk. For a Wukong job with a reducer, your output should be tab-delimited data, one record per line. You can designate the fields to use for the partition key, the sort key and the group key. (By default, the first field is used for all three.)

The typical job turns each input record into zero, one or many records in a predictable manner, but such decorum is not required by Hadoop. You can read in lines from Shakespeare and emit digits of *pi*; read in all input records, ignore them and emit nothing; or boot into an Atari 2600 emulator, publish the host and port and start playing Pac-Man. Less frivolously: you can accept URLs or filenames (local or HDFS) and emit their contents; accept a small number of simulation parameters and start a Monte Carlo simulation; or accept a database query, issue it against a datastore and emit each result.

The Group/Sort Guarantee

When Hadoop does the group/sort, it establishes the following guarantee for the data that arrives at the reducer:

- each labelled record belongs to exactly one sorted group;
- each group is processed by exactly one reducer;
- groups are sorted lexically by the chosen group key;
- and records are further sorted lexically by the chosen sort key.

It's very important that you understand what that unlocks, so I'm going to redundantly spell it out a few different ways:

- Each mapper-output record goes to exactly one reducer, solely determined by its key.
- If several records have the same key, they will all go to the same reducer.
- From the reducer's perspective, if it sees any element of a group it will see all elements of the group.

You should typically think in terms of groups and not about the whole reduce set: imagine each partition is sent to its own reducer. It's important to know, however, that each reducer typically sees multiple partitions. (Since it's more efficient to process large batches, a certain number of reducer processes are started on each machine. This is in contrast to the mappers, who run one task per input split.) Unless you take special measures, the partitions are distributed arbitrarily among the reducers [1]. They are fed to the reducer in order by key.

1. Using a "consistent hash"; see (REF) the chapter on Sampling

Similar to a mapper-only task, your reducer can output anything you like, in any format you like. It's typical to output structured records of the same or different shape, but you're free engage in any of the shenanigans listed above.

The Map Phase Processes Records Individually

TODO-qem: Are there parts of this that dive into the weeds, and if so we could move them to 06a-Hadoop Internals TODO-qem: does anything here get tangled with the "hadoop contract" section

The Map phase receives 0, 1 or many records individually, with no guarantees from Hadoop about their numbering, order or allocation. (FOOTNOTE: In special cases, you may know that your input bears additional guarantees — for example, the MERGE/JOIN described in Chapter (REF) requires its inputs to be in total sorted order. It is on you, however, to enforce and leverage those special properties.) Hadoop does guarantee that every record arrives in whole to exactly one Map task and that the job will only succeed if every record is processed without error.

The Mapper receives those records sequentially — it must fully process one before it receives the next — and can emit 0, 1 or many inputs of any shape or size. The chimpanzees working on the SantaCorp project received letters but dispatched toy forms. Julia's thoughtful note produced two toy forms, one for her doll and one for Joe's robot, while the spam letter produced no toy forms. Hadoop's *distcp* utility, used to copy data from cluster to cluster, takes this to a useful extreme: Each Mapper's input is a remote file to fetch. Its action is to write that file's contents directly to the HDFS as a Datanode client and its output is a summary of what it transferred.

The right way to bring in data from an external resource is by creating a custom loader or input format (see the chapter on Advanced Pig (REF)), which decouples loading data from processing data and allows Hadoop to intelligently manage tasks. The poor-man's version of a custom loader, useful for one-offs, is to prepare a small number of file names, URLs, database queries or other external handles as input and emit the corresponding contents.

Please be aware, however, that it is only appropriate to access external resources from within a Hadoop job in exceptionally rare cases. Hadoop processes data in batches, which means failure of a single record results in the retry of the entire batch. It also means that when the remote resource is unavailable or responding sluggishly, Hadoop will spend several minutes and unacceptably many retries before abandoning the effort. Lastly, Hadoop is designed to drive every system resource at its disposal to its performance limit. (FOOTNOTE: We will drive this point home in the chapter on Event Log Processing (REF), where we will stress test a web server to its performance limit by replaying its request logs at full speed.)

While a haiku with only its first line is no longer a haiku, a Hadoop job with only a Mapper is a perfectly acceptable Hadoop job, as you saw in the Pig Latin translation example. In such cases, each Map Task's output is written directly to the HDFS, one file per Map Task, as you've seen. Such jobs are only suitable, however, for so-called "embarrassingly parallel problems" — where each record can be processed on its own with no additional context.

The Map stage in a Map/Reduce job has a few extra details. It is responsible for labeling the processed records for assembly into context groups. Hadoop files each record into the equivalent of the pigmy elephants' file folders: an in-memory buffer holding each record in sorted order. There are two additional wrinkles, however, beyond what the pigmy elephants provide. First, the Combiner feature lets you optimize certain special cases by preprocessing partial context groups on the Map side; we will describe these more in a later chapter (REF). Second, if the sort buffer reaches or exceeds a total count or size threshold, its contents are "spilled" to disk and subsequently merge/sorted to produce the Mapper's proper output.

How Hadoop Manages Midstream Data

The first part of this chapter (REF) described the basics of what Hadoop supplies to a Reducer: each record is sent to exactly one reducer; all records with a given label are sent to the same Reducer; and all records for a label are delivered in a continuous ordered group. Let's understand the remarkably economical motion of data Hadoop uses to accomplish this.

Mappers Spill Data In Sorted Chunks ===

As your Map task produces each labeled record, Hadoop inserts it into a memory buffer according to its order. Like the dextrous chimpanzee, the current performance of CPU and memory means this initial ordering imposes negligible overhead compared to the rate that data can be read and processed. When the Map task concludes or that memory buffer fills, its contents are flushed as a stream to disk. The typical Map task operates on a single HDFS block and produces an output size not much larger. A well-configured Hadoop cluster sets the sort buffer size accordingly (FOOTNOTE: The chapter on Hadoop Tuning For The Brave And Foolish (REF) shows you how); that most common case produces only a single spill.

If there are multiple spills, Hadoop performs the additional action of merge/sorting the chunks into a single spill. (FOOTNOTE: This can be somewhat expensive, so in Chapter (REF), we will show you how to avoid unnecessary spills.) Whereas the pygmy elephants each belonged to a distinct workbench, a Hadoop Mapper produces only that one unified spill. That's ok — it is easy enough for Hadoop to direct the records as each is sent to its Reducer.

As you know, each record is sent to exactly one Reducer. The label for each record actually consists of two important parts: the partition key that determines which Reducer the record belongs to, and the sort key, which groups and orders those records within the Reducer's input stream. You will notice that, in the programs we have written, we only had to supply the record's natural label and never had to designate a specific Reducer; Hadoop handles this for you by applying a partitioner to the key.

Partitioners Assign Each Record To A Reducer By Label

The default partitioner, which we find meets almost all our needs, is called the "RandomPartitioner." (FOOTNOTE: In the next chapter (REF), you will meet another partitioner, when you learn how to do a total sort.) It aims to distribute records uniformly across the Reducers by giving each key the same chance to land on any given Reducer. It is not really random in the sense of nondeterministic; running the same job with the same configuration will distribute records the same way. Rather, it achieves a uniform distribution of keys by generating a cryptographic digest — a number produced from the key with the property that any change to that key would instead produce an arbitrarily distinct number. Since the numbers thus produced have high and uniform distribution, the digest MODULO the number of Reducers reliably balances the Reducer's keys, no matter their raw shape and size. (FOOTNOTE: If you will recall, x MODULO y gives the remainder after dividing x and y. You can picture it as a clock with y hours on it: 15 MODULO 12 is 3; 4 MODULO 12 is 4; 12 MODULO 12 is 0).

NOTE The default partitioner aims to provide a balanced distribution of *keys* — which does not at all guarantee a uniform distribution of *records* ! If 40-percent of your friends have the last name Chimpanzee and 40-percent have the last name Elephant, running a Map/Reduce job on your address book, partitioned by last name, will send all the Chimpanzees to some Reducer and all the Elephants to some Reducer (and if you are unlucky, possibly even the same one). Those unlucky Reducers will struggle to process 80-percent of the data while the remaining Reducers race through their unfairly-small share of what is left. This situation is far more common and far more difficult to avoid than you might think, so large parts of this book's intermediate chapters are, in effect, tricks to avoid that situation.

(TODO: Move merge/sort description here??)

Reducers Receive Sorted Chunks From Mappers

Partway through your job's execution, you will notice its Reducers spring to life. Before each Map task concludes, it streams its final merged spill over the network to the appropriate Reducers (FOOTNOTE: Note that this communication is direct; it does not use the HDFS). Just as above, the Reducers file each record into a sort buffer, spills that buffer to disk as it fills and begins merge/sorting them once a threshold of spills is reached.

Whereas the numerous Map tasks typically skate by with a single spill to disk, you are best off running a number of Reducers, the same as or smaller than the available slots. This generally leads to a much larger amount of data per Reducer and, thus, multiple spills.

Reducers Read Records With A Final Merge/Sort Pass

The Reducers do not need to merge all records to a single unified spill. The elves at each workbench pull directly from the limited number of parts carts as they work' similarly, once the number of mergeable spills is small enough, the Reducer begins processing records from those spills directly, each time choosing the next in sorted order.

Your program's Reducer receives the records from each group in sorted order, outputting records as it goes. Your reducer can output as few or as many records as you like at any time: on the start or end of its run, on any record, or on the start or end of a group. It is not uncommon for a job to produce output the same size as or larger than its input — "Reducer" is a fairly poor choice of names. Those output records can also be of any size, shape or format; they do not have to resemble the input records, and they do not even have to be amenable to further Map/Reduce processing.

Reducers Write Output Data and Commit

As your Reducers emit records, they are streamed directly to the job output, typically the HDFS or S3. Since this occurs in parallel with reading and processing the data, the primary spill to the Datanode typically carries minimal added overhead.

TODO a bit more about the fact that data **is** written to disk TODO: mention commit phase TODO: check that we have here or in chapter 2 talked about the highest-level detail of how data is written to disk

You may wish to send your job's output not to the HDFS or S3 but to a scalable database or other external data store. (We'll show an example of this in the chapter on HBase (REF)) While your job is in development, though, it is typically best to write its output directly to the HDFS (perhaps at replication factor 1), then transfer it to the external target in a separate stage. The HDFS is generally the most efficient output target and the least likely to struggle under load. This checkpointing also encourages the best practice of sanity-checking your output and asking questions.

Outro

You've just seen how records move through a map/reduce workflow, along with aggregation of records and matching records betweent datasets — patterns that will recur in many explorations. Next, JT and Nanette will make a new friend, and we'll see another model for Hadoop analytics based on those patterns.

CHAPTER 4

Structural Operations

Olga, the Remarkable Calculating Pig

JT and Nanette were enjoying the rising success of C&E Corp. The translation and SantaCorp projects were in full production, and they'd just closed two more deals that closely resembled the SantaCorp gig.

Still, it was quite a thrill when the manager for Olga the Remarkable Calculating Pig reached out to *them*, saying Olga had a proposition to discuss. Imagine! The star that played nightly to sell-out crowds at Carnegie Hall, whose exploits of numeracy filled the journals and whose exploits of romance filled the tabloids, working with JT and Nanette! "Why don't you kids come see the show — we'll leave tickets for you at the gate — and you can meet with Olga after she gets off."

That night they watched, spellbound, as Olga performed monstrous feats of calculation and recall. In one act, she tallied the end-of-year accounting reports for three major retailers while riding a unicycle; in another, she listed the box-office numbers for actors whose names were drawn from a hat. Needless to say, the crowd roared for more, JT and Nanette along with them. For the grand finale, a dozen audience members wrote down their favorite baseball players — most well-known, but of course some wise guy wrote down Alamazoo Jennings, Snooks Dowd or Vinegar Bend Mizell to be intentionally obscure [1]. Olga not only recited the complete career stats for every one, but the population of their hometown; every teammate they held in common; and the construction date of every stadium they played in.

"I tell you, that's some pig", Nanette said to JT as they waited outside the dressing rooms. "Terrific," JT agreed. A voice behind them said "Radiant and Humble, too, they tell me."

1. Yes, these are names of real major league baseball players.

They turned to find Olga, now dressed in street clothes. "Why don't you join me for a drink? We can talk then."

Nanette and Olga Have an Idea

Once settled in at a bar down the street, JT broke the ice. "Olga, your show was amazing. When you rattled off Duluth, Minnesota's daily low and high temperatures from 1973 to 1987, chills ran down my spine. But I can't quite figure out what kind of help C&E Corp can provide for you?" Nanette chimed in, "Actually, I think I have an idea — but I'd like to hear your thoughts first, Olga."

As olga explained, "I first heard about you from my publisher — my friend Charlotte and I wrote a book about web crawlers, and thanks to your work we're selling as many copies overseas as we are domestically. But it wasn't until I visited the translation floor that I really appreciated the scale of content you guys were moving. And that's what I'm looking for — high scale.

"You might know that besides my stage act I consult on the side for companies who need a calculating animal savant. I love that just as much as being on stage, but the fact is that what I can do for my clients just seems so *limited*. I've got insurance companies who want to better judge tornado risk so they can help people protect their homes; but to do this right means using the full historical weather data. I have to explain to them that I'm just one pig — I'd melt down if I tried to work with that much information.

"Goldbug automakers engages me to make inventory reports based on daily factory output and dealership sales, and I can literally do this in my sleep. But they're collecting thousands of times that much data each second. For instance, they gather status reports from every automated step in their factory. If I could help Goldbug compare the manufacturing data of the cars as they're built to the maintenance records of those cars after sale, we'd be able to find patterns in the factory that match warranty claims down the road. Predicting these manufacturing defects early would enable my client to improve quality, profit and customer satisfaction.

"I wish I could say I invited you for this drink because I knew the solution, but all I have is a problem I'd like to fix. I know your typewriter army helps companies process massive amounts of documents, so you're used to working with the amount of information I'm talking about. Is the situation hopeless, or can you help me find a way to apply my skills at a thousand times the scale I work at now?"

Nanette smiled. "It's not hopeless at all, and to tell you the truth your proposal sounds like the other end of a problem I've been struggling with.

"We've now had several successful client deliveries, and recently JT's made some breakthroughs in what our document handling system can do — it involves having the chimpanzees at one set of typewriters send letters to another set of chimpanzees at a different

set of typewriters. One thing we're learning is that even though the actions that the chimpanzees take are different for every client, there are certain themes in how the chimpanzees structure their communication that recur across clients.

"Now JT here" (at this, JT rolled his eyes for effect, as he knew what was coming) "spent all his time growing up at a typewriter, and so he thinks about information flow as a set of documents. Designing a new scheme for chimpanzees to send inter-office memos is like pie for him. But where JT thinks about working with words on a page, I think about managing books and libraries. And the other thing we're learning is that our clients think like me. They want to be able to tell us the big picture, not fiddly little rules about what should happen to each document. Tell me how you describe the players-and-stadiums trick you did in the grand finale.

"Well, I picture in my head the teams every player was on for each year they played, and at the same time a listing of each team's stadium by year. Then I just think *match the players*\ seasons to the teams\' seasons using the team and year', and the result pops into my head.

Nanette nodded and looked over at JT. "I see what you're getting at now," he replied. "In my head I'm thinking about the process of matching individual players and stadiums — when I explain it you're going to think it sounds more complicated but I don't know, to me it seems simpler. I imagine that I could ask each player to write down on a yellow post-it note the team-years they played on, and ask each stadium manager to write down on blue post-it notes the team-years it served. Then I put those notes in piles — whenever there's a pile with yellow post-it notes, I can read off the blue post-it notes it matched.

Nanette leaned in. "So here's the thing. Elephants and Pigs have amazing memories, but not Chimpanzees — JT can barely keep track of what day of the week it is. JT's scheme never requires him to remember anything more than the size of the largest pile — in fact, he can get by with just remembering what's on the yellow post-it notes. But

"Well," Nanette said with a grin, "Pack a suitcase with a very warm jacket. We're going to take a trip up north — way north."

Pig Helps Hadoop work with Tables, not Records

Pig is an open-source, high-level language that enables you to create efficient Map/Reduce jobs using clear, maintainable scripts. Its interface is similar to SQL, which makes it a great choice for folks with significant experience there. (It's not identical, though, and things that are efficient in SQL may not be so in Pig; we will try to highlight those traps.)

Let's dive in with an example using the UFO dataset to estimate whether aliens tend to visit in some months over others:

```
PARALLEL 1; (USE 1 REDUCER) (DISABLE COMBINERS)
LOAD UFO table
EXTRACT MONTH FROM EACH LINE
GROUP ON MONTHS
COUNT WITHIN GROUPS
STORE INTO OUTPUT FILE
```

In a Wukong script or traditional Hadoop job, the focus is on the record and you're best off thinking in terms of message passing or grouping. In Pig, the focus is much more on the structure and you should think in terms of relational and set operations. In the example above, each line described an operation on the full dataset; we declared what change to make and Pig, as you'll see, executes those changes by dynamically assembling and running a set of Map/Reduce jobs.

Here's what you might write in Wukong to answer the same question:

```
DEFINE MODEL FOR INPUT RECORDS
MAPPER EXTRACTS MONTHS, EMITS MONTH AS KEY WITH NO VALUE
COUNTING REDUCER INCREMENTS ON EACH ENTRY IN GROUP AND EMITS TOTAL IN FINALIZED METHOD
```

Did you notice, by the way, that in both cases, the output was sorted? that is no coincidence — as you saw in Chapter (TODO: REF), Hadoop sorted the results in order to group them.

To run the Pig job, go into the 'EXAMPLES/UFO' directory and run

```
pig monthly_visit_counts.pig /data_UFO_sightings.tsv /dataresults monthly_visit_counts-pig.tsv
```

To run the Wukong job, go into the (TODO: REF) directory and run

```
wu-run monthly_visit_counts.rb --reducers_count=1 /data_UFO_sightings.tsv /dataresults monthly_vis
```

If you consult the output, you'll see (TODO: INSERT CONCLUSIONS).

If you consult the Job Tracker Console, you should see a single Map/Reduce for each with effectively similar statistics; the dataflow Pig instructed Hadoop to run is essentially similar to the Wukong script you ran. What Pig ran was, in all respects, a Hadoop job. It calls on some of Hadoop's advanced features to help it operate but nothing you could not access through the standard Java API.

Wikipedia Visitor Counts

Let's put Pig to a sterner test. Here's the script above, modified to run on the much-larger Wikipedia dataset and to assemble counts by hour, not month:

```
LOAD SOURCE FILE
PARALLEL 3
TURN RECORD INTO HOUR PART OF TIMESTAMP AND COUNT
GROUP BY HOUR
SUM THE COUNTS BY HOUR
ORDER THE RESULTS BY HOUR
STORE INTO FILE
```

(TODO: If you do an order and then group, is Pig smart enough to not add an extra REDUCE stage?)

Run the script just as you did above:

```
(TODO: command to run the script)
```

Up until now, we have described Pig as authoring the same Map/Reduce job you would. In fact, Pig has automatically introduced the same optimizations an advanced practitioner would have introduced, but with no effort on your part. If you compare the Job Tracker Console output for this Pig job with the earlier ones, you'll see that, although x bytes were read by the Mapper, only y bytes were output. Pig instructed Hadoop to use a Combiner. In the naive Wukong job, every Mapper output record was sent across the network to the Reducer but in Hadoop, as you will recall from (TODO: REF), the Mapper output files have already been partitioned and sorted. Hadoop offers you the opportunity to do pre-Aggregation on those groups. Rather than send every record for, say, August 8, 2008 8 pm, the Combiner outputs the hour and sum of visits emitted by the Mapper.

```
SIDEBAR:  You can write Combiners in Wukong, too. (TODO: Insert example with Combiners)
```

You'll notice that, in the second script, we introduced the additional operation of instructing Pig to explicitly sort the output by minute. We did not do that in the first example because its data was so small that we had instructed Hadoop to use a single Reducer. As you will recall from (TODO: REF), Hadoop uses a Sort to prepare the Reducer groups, so its output was naturally ordered. If there are multiple Reducers, however, that would not be enough to give you a Result file you can treat as ordered. By default, Hadoop assigns partitions to Reducers using the 'RandomPartitioner', designed to give each Reducer a uniform chance of claiming any given partition. This defends against the problem of one Reducer becoming overwhelmed with an unfair share of records but means the keys are distributed willy-nilly across machines. Although each Reducer's output is sorted, you will see records from 2008 at the top of each result file and records from 2012 at the bottom of each result file.

What we want instead is a total sort, the earliest records in the first numbered file in order, the following records in the next file in order, and so on until the last numbered file. Pig's 'ORDER' Operator does just that. In fact, it does better than that. If you look at the Job Tracker Console, you will see Pig actually ran three Map/Reduce jobs. As you would expect, the first job is the one that did the grouping and summing and the last job is the one that sorted the output records. In the last job, all the earliest records were sent to Reducer 0, the middle range of records were sent to Reducer 1 and the latest records were sent to Reducer 2.

Hadoop, however, has no intrinsic way to make that mapping happen. Even if it figured out, say, that the earliest buckets were in 2008 and the latest buckets were in 2012, if we fed it a dataset with skyrocketing traffic in 2013, we would end up sending an over-

whelming portion of results to that Reducer. In the second job, Pig sampled the set of output keys, brought them to the same Reducer, and figured out the set of partition breakpoints to distribute records fairly.

In general, Pig offers many more optimizations beyond these and we will talk more about them in the chapter on Advanced Pig (TODO: REF). In our experience, the only times Pig will author a significantly less-performant dataflow than would an expert comes when Pig is overly aggressive about introducing an optimization. The chief example you'll hit is that often, the intermediate stage in the total sort to calculate partitions has a larger time penalty than doing a bad job of partitioning would; you can disable that by (TODO: Describe how).

Group and Flatten

The fundamental Map/Reduce operation is to group a set of records and operate on that group. In fact, it's a one-liner in Pig:

```
BINS = Group WP_pageviews by (date, hour)
DESCRIBE BINS
(TODO: SHOW OUTPUT)
```

The result is always a tuple whose first field is named "Group" — holding the individual group keys in order. The next field has the full input record with all its keys, even the group key. Here's a Wukong script that illustrates what is going on:

```
(TODO: Wukong script)
```

You can group more than one dataset at the same time. In weather data, there is one table listing the location and other essentials of each weather station and a set of tables listing, for each hour, the weather at each station. Here's one way to combine them into a new table, giving the explicit latitude and longitude of every observation:

```
G1=GROUP WSTNS BY (ID1, ID2), WOBS BY (ID1, ID2);
G2=FLATTEN G1…
G3=FOR EACH G2 …
```

This is equivalent to the following Wukong job:

```
(TODO: Wukong job)
```

(TODO: replace with an example where you would use a pure code group).

Join Practicalities

The output of the Join job has one line for each discrete combination of A and B. As you will notice in our Wukong version of the Join, the job receives all the A records for a given key in order, strictly followed by all the B records for that key in order. We have to accumulate all the A records in memory so we know what rows to emit for each B record. All the A records have to be held in memory at the same time, while all the B

records simply flutter by; this means that if you have two datasets of wildly different sizes or distribution, it is worth ensuring the Reducer receives the smaller group first. In Wukong, you do this by giving it an earlier-occurring field group label; in Pig, always put the table with the largest number of records per key last in the statement.

Ready Reckoner: How fast should your Pig fly?

TODO: describe for each Pig command what jobs should result.

More

There are a few more Operators we will use later in the book: Cube, which produces aggregations at multiple levels within a Group; Rank, which is sugar on top of Order to produce a number, total-ordered set of records; Split, to separate a dataset into multiple pieces; and Union, to produce a new dataset to have all the records from its input datasets.

That's really about it. Pig is an extremely sparse language. By having very few Operators and very uniform syntax (FOOTNOTE: Something SQL users but non-enthusiasts like your authors appreciate), the language makes it easy for the robots to optimize the dataflow and for humans to predict and reason about its performance.

We won't spend any more time introducing Pig, the language, as its usage will be fairly clear in context as you meet it later in the book. The online Pig manual at (TODO: REF) is quite good and for a deeper exploration, consult (TODO: Add name of best Pig book here).

Pig Gotchas

That one error where you use the dot or the colon when you should use the other. Where to look to see that Pig is telling you have either nulls, bad fields, numbers larger than your type will hold or a misaligned schema.

Fundamental Data Operations

Pig's operators can be grouped into several families.

- Simple processing operations (FOREACH, ASSERT, STREAM, UNION) modify the contents of records individually. These become Mapper-Only jobs with exactly the same count of output records as input records.
- Simple filtering operations (FILTER, SAMPLE, LIMIT, SPLIT) accept or reject each record individually; these also produce Mapper-Only jobs with the same or fewer number of records and each record has the same contents and schema as its input.

(SPLIT is effectively several FILTERs run simultaneously, so its total output record count is the sum of what each of its filters would produce)

- Ungrouping operations (FOREACH..FLATTEN) turn records that have bags of tuples into records with each such tuple from the bags in combination. It is most commonly seen after a grouping operation (and thus occurs within the Reduce) but just produces a Mapper-Only job when used on its own. Records with empty bags will disappear in the output while, if you FLATTEN a record having one bag field with three tuples and another bag field with four tuples, you will get 12 output records, one for each of the bags in combination.
- Grouping operations (JOIN, GROUP, COGROUP, CUBE, DISTINCT, CROSS) place records into context with each other. They make no modifications to their input records. You will often find them followed by a FOREACH that is able to take advantage of the group context. These jobs require a Map and Reduce phase. The GROUP and COGROUP yield themselves one output record per distinct GROUP value. A JOIN is simply an optimized GROUP/FLATTEN/FOREACH sequence, so its output size follows the same logic as FLATTEN.
- Sorting operations (ORDER BY, RANK) perform a total sort on their input; every record in file 00000 is in sorted order and comes before all records in 00001 and so forth for the number of output files. These require two jobs: first, a light Mapper-Only pass to understand the distribution of sort keys, next a Map/Reduce job to perform the sort.
- Serialization operations (LOAD, STORE) load and store data into file systems or datastores.
- Directives (DESCRIBE, ILLUSTRATE, REGISTER, and others) to Pig itself. These do not modify the data, they modify Pig's execution: outputting debug information, registering external UDFs, and so forth.

That's it. That's everything you can do with Pig — and everything you need to do with data. Each of those operations leads to a predictable set of map and reduce steps, so it's very straightforward to reason about your job's performance. Pig is very clever about chaining and optimizing these steps. For example, a GROUP followed by a FOREACH and a FILTER will only require one map phase and one reduce phase. In that case, the FOREACH and FILTER will be done in the reduce step — and in the right circumstances, pig will "push" part of the `FOREACH` and `FILTER` *before* the `JOIN`, potentially eliminating a great deal of processing.

In the remainder of this chapter, we'll illustrate the essentials for each family of operations, demonstrating them in actual use. In the following chapter (TODO ref), we'll learn how to implement the corresponding patterns in a plain map-reduce approach — and therefore how to reason about their performance. Finally, the chapter on Advanced

Pig (TODO ref) will cover some deeper-level topics, such as a few important optimized variants of the JOIN statement and how to endow Pig with new functions and loaders.

We will not explore every nook and cranny of its syntax, only illustrate its patterns of use. We've omitted operations whose need hasn't arisen naturally in the explorations later, along with fancy but rarely-used options or expressions [2]

LOAD..AS gives the location and schema of your source data

Pig scripts need data to process, and so your pig scripts will begin with a LOAD statement and have one or many STORE statements throughout. Here's a script to find all wikipedia articles that contain the words *Hadoop*:

```
articles = LOAD './data/wp/articles.tsv' AS (page_id: long, namespace: int, wikipedia_id: chararra
hadoop_articles = FILTER articles BY text matches '.*Hadoop.*';
STORE hadoop_articles INTO './data/tmp/hadoop_articles.tsv';
```

Simple Types

As you can see, the `LOAD` statement not only tells pig where to find the data, it also describes the table's schema. Pig understands ten kinds of simple type. Six of them are numbers: signed machine integers, as `int` (32-bit) or `long` (64-bit); signed floating-point numbers, as `float` (32-bit) or `double` (64-bit); arbitrary-length integers as `biginteger`; and arbitrary-precision real numbers, as `bigdecimal`. If you're supplying a literal value for a long, you should append a capital *L* to the quantity: `12345L`; if you're supplying a literal float, use an *F*: `123.45F`.

The `chararray` type loads text as UTF-8 encoded strings (the only kind of string you should ever traffic in). String literals are contained in single quotes — `'hello, world'`. Regular expressions are supplied as string literals, as in the example above: `'.*[Hh]adoop.*'`. The `bytearray` type does no interpretation of its contents whatsoever, but be careful — the most common interchange formats (`tsv`, `xml` and `json`) cannot faithfully round-trip data that is truly freeform.

Lastly, there are two special-purpose simple types. Time values are described with `datetime`, and should be serialised in the the ISO-8601 format: `1970-01-01T00:00:00.000+00:00`. Boolean values are described with `boolean`, and should bear the values `true` or `false`.

2. For example, it's legal in Pig to load data without a schema — but you shouldn't, and so we're not going to tell you how.

Complex Type 1: Tuples are fixed-length sequences of typed fields

Pig also has three complex types, representing collections of fields. A `tuple` is a fixed-length sequence of fields, each of which has its own schema. They're ubiquitous in the results of the various structural operations you're about to learn. We usually don't serialize tuples, but so far `LOAD` is the only operation we've taught you, so for pretend's sake here's how you'd load a listing of major-league ballpark locations:

```
-- The address and geocoordinates are stored as tuples. Don't do that, though.
ballpark_locations = LOAD 'ballpark_locations' AS (
    park_id:chararray, park_name:chararray,
    address:tuple(full_street:chararray, city:chararray, state:chararray, zip:chararray),
    geocoordinates:tuple(lng:float, lat:float)
);
ballparks_in_texas = FILTER ballpark_locations BY (address.state == 'TX');
STORE ballparks_in_texas INTO '/tmp/ballparks_in_texas.tsv'
```

Pig displays tuples using parentheses: it would dump a line from the input file as `BOS07,Fenway Park,(4 Yawkey Way,Boston,MA,02215),(-71.097378,42.3465909)'`. As shown above, you address single values with `in a tuple using `tuple_name.subfield_name` — `address.state` will have the schema `state:chararray`. You can also project fields in a tuple into a new tuple by writing `tuple_name.(subfield_a, subfield_b, ...)` — `address.(zip, city, state)` will have schema `address_zip_city_state:tuple(zip:chararray, city:chararray, state:chararray)`. (Pig helpfully generated a readable name for the tuple).

Tuples can contain values of any type, even bags and other tuples, but that's nothing to be proud of. You'll notice we follow almost every structural operation with a `FOREACH` to simplify its schema as soon as possible, and so should you — it doesn't cost anything and it makes your code readable.

Complex Type 2: Bags hold zero one or many tuples

A `bag` is an arbitrary-length collection of tuples, all of which are expected to have the same schema. Just like with tuples, they're ubiquitous yet rarely serialized tuples; but again for pretend's sake we can load a dataset listing for each team the year and park id of the ballparks it played in:

```
team_park_seasons = LOAD 'team_parks' AS (
    team_id:chararray,
    park_years: bag{tuple(year:int, park_id:chararray)}
    );
```

You address values within a bag again using `bag_name.(subfield_a, subfield_b)`, but this time the result is a bag with the given projected tuples — you'll see examples of this shortly when we discuss `FLATTEN` and the various group operations. Note that the

only type a bag holds is tuple, even if there's only one field — a bag of just park ids would have schema `bag{tuple(park_id:chararray)}`.

Complex Type 3: Maps hold collections of key-value pairs for lookup

Pig offers a `map` datatype to represent a collection of key-value pairs. The only context we've seen them used is for loading JSON data. A tweet from the twitter firehose has a sub-hash holding info about the user; the following snippet loads raw JSON data, immediately fixes the schema, and then describes the new schema to you:

```
REGISTER piggybank.jar
raw_tweets = LOAD '/tmp/tweets.json' USING org.apache.pig.piggybank.storage.JsonLoader(
             'created_at:chararray, text:chararray, user:map[]');
tweets = FOREACH raw_tweets GENERATE
        created_at,
        text,
        user#'id' AS user_id:long,
        user#'name' AS user_name:chararray,
        user#'screen_name' AS user_screen_name:chararray;
DESCRIBE tweets;
```

A `map` schema is described using square brackets: `map[value_schema]`. You can leave the value schema blank if you supply one later (as in the example that follows). The keys of a map are *always* of type chararray; the values can be any simple type. Pig renders a map as `[key#value,key#value,...]`: my twitter user record as a hash would look like `[name#Philip Kromer,id#1554031,screen_name#mrflip]`.

Apart from loading complex data, the `map` type is surprisingly useless. You might think it would be useful to carry around a lookup-table in a map field — a mapping from ids to names, say — and then index into it using the value of some other field, but a) you cannot do so and b) it isn't useful. The only thing you can do with a `map` field is dereference by a constant string, as we did above (`user#'id'`). Carrying around such a lookup table would be kind of silly, anyway, as you'd be duplicating it on every row. What you most likely want is either an off-the-cuff UDF or to use Pig's "replicated" `JOIN` operation; both are described in the chapter on Advanced Pig (TODO ref).

Since the map type is mostly useless, we'll seize the teachable moment and use this space to illustrate the other way schema are constructed: using a `FOREACH`. As always when given a complex schema, we took the first available opportunity to simplify it. The `FOREACH` in the snippet above dereferences the elements of the user `map` and supplies a schema for each new field with the `AS <schema>` clauses. The `DESCRIBE` directive that follows causes Pig to dump the schema to console: in this case, you should see `tweets: {created_at: chararray,text: chararray,user_id: long,user_name: charar ray,user_screen_name: chararray}`.

(TODO ensure these topics are covered later: combining input splits in Pig; loading different data formats)

FOREACH: modify the contents of records individually

We can now properly introduce you to the first interesting Pig command. A `FOREACH` makes simple transformations to each record.

For example, baseball fans use a few rough-but-useful player statistics to compare players' offensive performance: batting average, slugging average, and offensive percentage. This script calculates just those statistics, along with the player's name, id and number of games played.

```
player_seasons = LOAD `player_seasons` AS (...);
qual_player_seasons = FILTER player_years BY plapp > what it should be;
player_season_stats = FOREACH qual_player_seasons GENERATE
    player_id, name, games,
    hits/ab AS batting_avg,
    whatever AS slugging_avg,
    whatever AS offensive_avg,
    whatever+whatever AS ops
    ;
STORE player_season_stats INTO '/tmp/baseball/player_season_stats';
```

This example digests the players table; selects only players who have more than a qualified number of plate appearances; and generates the stats we're interested in (If you're not a baseball fan, just take our word that "these four fields are particularly interesting")

A `FOREACH` won't cause a new Hadoop job stage: it's chained onto the end of the preceding operation (and when it's on its own, like this one, there's just a single a mapper-only job). A FOREACH always produces exactly the same count of output records as input records.

Within the GENERATE portion of a FOREACH, you can apply arithmetic expressions (as shown); project fields (rearrange and eliminate fields); apply the FLATTEN operator (see below); and apply Pig functions to fields. Let's look at Pig's functions.

Pig Functions act on fields

Pig offers a sparse but essential set of built-in functions. The Pig cheatsheet (TODO ref) at the end of the book gives a full list, but here are the highlights:

- **Math functions** for all the things you'd expect to see on a good calculator: `LOG`/`LOG10`/`EXP`, `RANDOM`, `ROUND`/`FLOOR`/`CEIL`, `ABS`, trigonometric functions, and so forth.
- **String comparison**:

  — `matches` tests a value against a regular expression:

— Compare strings directly using `==`. `EqualsIgnoreCase` does a case-insensitive match, while `STARTSWITH`/`ENDSWITH` test whether one string is a prefix or suffix of the other.

— `SIZE` returns the number of characters in a `chararray`, and the number of bytes in a `bytearray`. Be reminded that characters often occupy more than one byte: the string *Motörhead* has nine characters, but because of its umlaut-ed *ö* occupies ten bytes. You can use `SIZE` on other types, too; but as mentioned, use `COUNT_STAR` and not `SIZE` to find the number of elements in a bag.

— `INDEXOF` finds the character position of a substring within a `chararray` // `LAST_INDEX_OF`

- **Transform strings**:

  — `CONCAT` concatenates all its inputs into a new string

  — `LOWER` converts a string to lowercase characters; `UPPER` to all uppercase // `LCFIRST`, `UCFIRST`

  — `TRIM` strips leading and trailing whitespace // `LTRIM`, `RTRIM`

  — `REPLACE(string, 'regexp', 'replacement')` substitutes the replacement string wherever the given regular expression matches, as implemented by `java.string.replaceAll`. If there are no matches, the input string is passed through unchanged.

  — `REGEX_EXTRACT(string, regexp, index)` applies the given regular expression and returns the contents of the indicated matched group. If the regular expression does not match, it returns NULL. The `REGEX_EXTRACT_ALL` function is similar, but returns a tuple of the matched groups.

  — `STRSPLIT` splits a string at each match of the given regular expression

  — `SUBSTRING` selects a portion of a string based on position

- **Datetime Functions**, such as `CurrentTime`, `ToUnixTime`, `SecondsBetween` (duration between two given datetimes)

- **Aggregate functions** that act on bags:

  — `AVG`, `MAX`, `MIN`, `SUM`

  — `COUNT_STAR` reports the number of elements in a bag, including nulls; `COUNT` reports the number of non-null elements. `IsEmpty` tests that a bag has elements. Don't use the quite-similar-sounding `SIZE` function on bags: it's much less efficient.

  — `SUBTRACT(bag_a, bag_b)` returns a new bag with all the tuples that are in the first but not in the second, and `DIFF(bag_a, bag_b)` returns a new bag with all

tuples that are in either but not in both. These are rarely used, as the bags must be of modest size — in general us an inner JOIN as described below.

— `TOP(num, column_index, bag)` selects the top `num` of elements from each tuple in the given bag, as ordered by `column_index`. This uses a clever algorithm that doesn't require an expensive total sort of the data — you'll learn about it in the Statistics chapter (TODO ref)

- **Conversion Functions** to perform higher-level type casting: `TOTUPLE`, `TOBAG`, `TOMAP`

FILTER: eliminate records using given criteria

The `FILTER` operation select a subset of records. This example selects all wikipedia articles that contain the word *Hadoop*:

```
articles = LOAD './data/wp/articles.tsv' AS (page_id: long, namespace: int, wikipedia_id: chararra
hadoop_articles = FILTER articles BY text matches '.*Hadoop.*';
STORE hadoop_articles INTO './data/tmp/hadoop_articles.tsv';
```

Filter as early as possible — and in all other ways reduce the number of records you're working with. (This may sound obvious, but in the next chapter (TODO ref) we'll highlight many non-obvious expressions of this rule).

It's common to want to extract a *uniform* sample — one where every record has an equivalent chance of being selected. Pig's `SAMPLE` operation does so by generating a random number to select records. This brings an annoying side effect: the output of your job is different on every run. A better way to extract a uniform sample is the "consistent hash digest" — we'll describe it, and much more about sampling, in the Statistics chapter (TODO ref).

LIMIT selects only a few records

The `LIMIT` operator selects only a given number of records. In general, you have no guarantees about which records it will select. Changing the number of mappers or reducers, small changes in the data, and so forth can change which records are selected. However, using the `ORDER` operator before a `LIMIT` *does* guarantee you will get the top `k` records — not only that, it applies a clever optimization (reservoir sampling, see TODO ref) that sharply limits the amount of data sent to the reducers. If you truly don't care which records to select, just use one input file (`some_data/part-00000`, not all of `some_data`).

Pig matches records in datasets using JOIN

For the examples in this chapter and often throughout the book, we will use the Retrosheet.org compendium of baseball data. We will briefly describe tables as we use them, but for a full explanation of its structure see the "Overview of Datasets" appendix (TODO: REF).

The core operation you will use to put records from one table into context with data from another table is the JOIN. A common application of the JOIN is to reunite data that has been normalized — that is to say, where the database tables are organized to eliminate any redundancy. For example, each Retrosheet game log lists the ballpark in which it was played but, of course, it does not repeat the full information about that park within every record. Later in the book, (TODO: REF) we will want to label each game with its geo-coordinates so we can augment each with official weather data measurements.

To join the game_logs table with the parks table, extracting the game time and park geocoordinates, run the following Pig command:

```
gls_with_parks_j = JOIN
    parks     BY (park_id),
    game_logs BY (park_id);
explain gls_with_parks_j;
gls_with_parks = FOREACH gls_with_parks_j GENERATE
  (game_id, gamelogs.park_id, game_time, park_lng, statium_lat);
explain gls_with_parks;
(TODO output of explain command)
```

The output schema of the new `gls_with_parks` table has all the fields from the `parks` table first (because it's first in the join statement), stapled to all the fields from the `game_logs` table. We only want some of the fields, so immediately following the JOIN is a FOREACH to extract what we're interested in. Note there are now two *park_id* columns, one from each dataset, so in the subsequent FOREACH, we need to dereference the column name with the table from which it came. (TODO: check that Pig does push the projection of fields up above the JOIN). If you run the script, *examples/geo/baseball_weather/geolocate_games.pig* you will see that its output has example as many records as there are *game_logs* because there is exactly one entry in the *parks* table for each park.

In the general case, though, a JOIN can be many to many. Suppose we wanted to build a table listing all the home ballparks for every player over their career. The *player_seasons* table has a row for each year and team over their career. If a player changed teams mid year, there will be two rows for that player. The *park_years* table, meanwhile, has rows by season for every team and year it was used as a home stadium. Some ballparks have served as home for multiple teams within a season and in other cases (construction or special circumstances), teams have had multiple home ballparks within a season.

The Pig script (TODO: write script) includes the following JOIN:

```
JOIN
 player_park_years=JOIN
  parks(year,team_ID),
  players(year,team_ID);
explain_player_park_year;
```

First notice that the JOIN expression has multiple columns in this case separated by commas; you can actually enter complex expressions here — almost all (but not all) the things you do within a FOREACH. If you examine the output file (TODO: name of output file), you will notice it has appreciably more lines than the input *player* file. For example (TODO: find an example of a player with multiple teams having multiple parks), in year x player x played for the x and the y and y played in stadiums p and q. The one line in the *players* table has turned into three lines in the *players_parks_years* table.

The examples we have given so far are joining on hard IDs within closely-related datasets, so every row was guaranteed to have a match. It is frequently the case, however, you will join tables having records in one or both tables that will fail to find a match. The *parks_info* datasets from Retrosheet only lists the city name of each ballpark, not its location. In this case we found a separate human-curated list of ballpark geolocations, but geolocating records — that is, using a human-readable location name such as "Austin, Texas" to find its nominal geocoordinates (-97.7,30.2) — is a common task; it is also far more difficult than it has any right to be, but a useful first step is match the location names directly against a gazette of populated place names such as the open source Geonames dataset.

Run the script (TODO: name of script) that includes the following JOIN:

```
park_places = JOIN
  parks BY (location) LEFT OUTER,
  places BY (concatenate(city, ", ", state);
DESCRIBE park_places;
```

In this example, there will be some parks that have no direct match to location names and, of course, there will be many, many places that do not match a park. The first two JOINs we did were "inner" JOINs — the output contains only rows that found a match. In this case, we want to keep all the parks, even if no places matched but we do not want to keep any places that lack a park. Since all rows from the left (first most dataset) will be retained, this is called a "left outer" JOIN. If, instead, we were trying to annotate all places with such parks as could be matched — producing exactly one output row per place — we would use a "right outer" JOIN instead. If we wanted to do the latter but (somewhat inefficiently) flag parks that failed to find a match, you would use a "full outer" JOIN. (Full JOINs are pretty rare.)

In a Pig JOIN it is important to order the tables by size — putting the smallest table first and the largest table last. (You'll learn why in the "Map/Reduce Patterns" (TODO: REF)

chapter.) So while a right join is not terribly common in traditional SQL, it's quite valuable in Pig. If you look back at the previous examples, you will see we took care to always put the smaller table first. For small tables or tables of similar size, it is not a big deal — but in some cases, it can have a huge impact, so get in the habit of always following this best practice.

NOTE
A Pig join is outwardly similar to the join portion of a SQL SELECT statement, but notice that al

Group Elements From Multiple Tables On A Common Attribute (COGROUP)

The fundamental structural operation in Map/Reduce is the COGROUP: assembling records from multiple tables into groups based on a common field; this is a one-liner in Pig, using, you guessed it, the COGROUP operation. This script returns, for every world map grid cell, all UFO sightings and all airport locations within that grid cell [3]:

```
sightings = LOAD('/data/gold/geo/ufo_sightings/us_ufo_sightings.tsv') AS (...);
airports     = LOAD('/data/gold/geo/airflights/us_airports.tsv') AS (...);
cell_sightings_airports = COGROUP
    sightings by quadkey(lng, lat),
    airports  by quadkey(lng, lat);
STORE cell_sightings_locations INTO '...';
```

In the equivalent Map/Reduce algorithm, you label each record by both the indicated key and a number based on its spot in the COGROUP statement (here, records from sightings would be labeled 0 and records from airports would be labeled 1). Have Hadoop then PARTITION and GROUP on the COGROUP key with a secondary sort on the table index. Here is how the previous Pig script would be done in Wukong:

```
mapper(partition_keys: 1, sort_keys: 2) do
  recordize_by_filename(/sightings/ => Wu::Geo::UfoSighting, /airport/ => Wu::Geo::Airport)
  TABLE_INDEXES = { Wu::Geo::UfoSighting => 0, Wu::Geo::Airport => 1 }
  def process(record)
    table_index = TABLE_INDEXES[record.class] or raise("Don't know how to handle records of type '
    yield( [Wu::Geo.quadkey(record.lng, record.lat), table_index, record.to_wire] )
  end
end

reducer do
  def recordize(quadkey, table_index, jsonized_record) ; ...; end
  def start(key, *)
    @group_key = key ;
    @groups = [ [], [] ]
  end
```

3. We've used the `quadkey` function to map geocoordinates into grid cells; you'll learn about in the Geodata Chapter (REF)

```
  def accumulate(quadkey, table_index, record)
    @groups[table_index.to_i] << record
  end
  def finalize
    yield(@group_key, *groups)
  end
end
```

The Mapper loads each record as an object (using the file name to recognize which class to use) and then emits the quadkey, the table index (0 for sightings, 1 for airports) and the original record's fields. Declaring partition keys 1, sort keys 2 insures all records with the same quadkey are grouped together on the same Reducer and all records with the same table index arrive together. The body of the Reducer makes temporary note of the GROUP key, then accumulates each record into an array based on its type.

The result of the COGROUP statement always has the GROUP key as the first field. Next comes the set of elements from the table named first in the COGROUP statement — in Pig, this is a bag of tuples, in Wukong, an array of objects. After that comes the set of elements from the next table in the GROUP BY statement and so on.

While a standalone COGROUP like this is occasionally interesting, it is also the basis for many other common patterns, as you'll see over the next chapters.

Complex FOREACH

Let's continue our example of finding the list of home ballparks for each player over their career.

```
parks = LOAD '.../parks.tsv' AS (...);
team_seasons = LOAD '.../team_seasons.tsv' AS (...)
park_seasons = JOIN parks BY park_id, team_seasons BY park_id;
park_seasons = FOREACH park_seasons GENERATE
    team_seasons.team_id, team_seasons.year, parks.park_id, parks.name AS park_name;

player_seasons = LOAD '.../player_seasons.tsv' AS (...);
player_seasons = FOREACH player_seasons GENERATE
    player_id, name AS player_name, year, team_id;
player_season_parks = JOIN
    parks           BY (year, team_id),
    player_seasons BY (year, team_id);
player_season_parks = FOREACH player_season_parks GENERATE player_id, player_name, parks::year AS

player_all_parks = GROUP player_season_parks BY (player_id);
describe player_all_parks;
Player_parks = FOREACH player_all_parks {
    player = FirstFromBag(players);
    home_parks = DISTINCT(parks.park_id);
    GENERATE group AS player_id,
        FLATTEN(player.name),
        MIN(players.year) AS beg_year, MAX(players.year) AS end_year,
```

```
        home_parks; -- TODO ensure this is still tuple-ized
}
```

Whoa! There are a few new tricks here. This alternative { curly braces form of FOREACH lets you describe its transformations in smaller pieces, rather than smushing everything into the single GENERATE clause. New identifiers within the curly braces (such as player) only have meaning within those braces, but they do inform the schema.

We would like our output to have one row per player, whose fields have these different flavors:

- Aggregated fields (beg_year, end_year) come from functions that turn a bag into a simple type (MIN, MAX).
- The player_id is pulled from the group field, whose value applies uniformly to the the whole group by definition. Note that it's also in each tuple of the bagged player_park_seasons, but then you'd have to turn many repeated values into the one you want…
- … which we have to do for uniform fields (like name) that are not part of the group key, but are the same for all elements of the bag. The awareness that those values are uniform comes from our understanding of the data — Pig doesn't know that the name will always be the same. The FirstFromBag (TODO fix name) function from the Datafu package grabs just first one of those values
- Inline bag fields (home_parks), which continue to have multiple values.

We've applied the DISTINCT operation so that each home park for a player appears only once. DISTINCT is one of a few operations that can act as a top-level table operation, and can also act on bags within a foreach — we'll pick this up again in the next chapter (TODO ref). For most people, the biggest barrier to mastery of Pig is to understand how the name and type of each field changes through restructuring operations, so let's walk through the schema evolution.

We JOIN`ed player seasons and team seasons on `(year, team_id). The resulting schema has those fields twice. To select the name, we use two colons (the disambiguate operator): players::year.

After the GROUP BY operation, the schema is group:int, player_season_parks:bag{tuple(player_id, player_name, year, team_id, park_id, park_name)}. The schema of the new group field matches that of the BY clause: since park_id has type chararray, so does the group field. (If we had supplied multiple fields to the BY clause, the group field would have been of type tuple). The second field, player_season_parks, is a bag of size-6 tuples. Be clear about what the names mean here: grouping on the player_season_parks *table* (whose schema has six fields) produced the player_parks table. The second field of the player_parks table is a tuple of

size six (the six fields in the corresponding table) named `player_season_parks` (the name of the corresponding table).

So within the `FOREACH`, the expression `player_season_parks.park_id` is *also* a bag of tuples (remember, bags only hold tuples!), now size-1 tuples holding only the park_id. That schema is preserved through the `DISTINCT` operation, so `home_parks` is also a bag of size-1 tuples.

Some late night under deadline, Pig will supply you with the absolutely baffling error message "scalar has more than one row in the output". You've gotten confused and used the tuple element operation (`players.year`) when you should have used the disambiguation operator (`players::year`). The dot is used to reference a tuple element, a common task following a `GROUP`. The double-colon is used to clarify which specific field is intended, common following a join of tables sharing a field name.

```
team_park_seasons = LOAD '/tmp/team_parks.tsv' AS (
    team_id:chararray,
    park_years: bag{tuple(year:int, park_id:chararray)},
    park_ids_lookup: map[chararray]
    );
team_parks = FOREACH team_park_seasons { distinct_park_ids = DISTINCT park_years.park_id; GENE
DUMP team_parks;
```

Ungrouping operations (FOREACH..FLATTEN) expand records

So far, we've seen using a group to aggregate records and (in the form of `JOIN') to match records between tables. Another frequent pattern is restructuring data (possibly performing aggregation at the same time). We used this several times in the first exploration (TODO ref): we regrouped wordbags (labelled with quadkey) for quadtiles containing composite wordbags; then regrouping on the words themselves to find their geographic distribution.

The baseball data is closer at hand, though, so l

```
team_player_years = GROUP player_years BY (team,year);
FOREACH team_player_years GENERATE
    FLATTEN(player_years.player_id), group.team, group.year, player_years.player_id;
```

In this case, since we grouped on two fields, `group` is a tuple; earlier, when we grouped on just the `player_id` field, `group` was just the simple value.

The contextify / reflatten pattern can be applied even within one table. This script will find the career list of teammates for each player — all other players with a team and year in common [4].

```
GROUP player_years BY (team,year);
FOREACH
    cross all players, flatten each playerA/playerB pair AS (player_a
FILTER coplayers BY (player_a != player_b);
GROUP by playerA
FOREACH {
    DISTINCT player B
}
```

Here's another

The result of the cross operation will include pairing each player with themselves, but since we don't consider a player to be their own teammate we must eliminate player pairs of the form `(Aaronha, Aaronha)`. We did this with a FILTER immediate before the second GROUP (the best practice of removing data before a restructure), but a defensible alternative would be to `SUBTRACT` playerA from the bag right after the `DISTINCT` operation.

Sorting (ORDER BY, RANK) places all records in total order

To put all records in a table in order, it's not sufficient to use the sorting that each reducer applies to its input. If you sorted names from a phonebook, file `part-00000` will have names that start with A, then B, up to Z; `part-00001` will also have names from A-Z; and so on. The collection has a *partial* order, but we want the *total order* that Pig's `ORDER BY` operation provides. In a total sort, each record in `part-00000` is in order and precedes every records in `part-00001`; records in `part-00001` are in order and precede every record in `part-00002`; and so forth. From our earlier example to prepare topline batting statistics for players, let's sort the players in descending order by the "OPS" stat (slugging average plus offensive percent, the simplest reasonable estimator of a player's offensive contribution).

```
player_seasons = LOAD `player_seasons` AS (...);
qual_player_seasons = FILTER player_years BY plapp > what it should be;
player_season_stats = FOREACH qual_player_seasons GENERATE
    player_id, name, games,
    hits/ab AS batting_avg,
    whatever AS slugging_avg,
    whatever AS offensive_pct
    ;
```

4. yes, this will have some false positives for players who were traded mid-year. A nice exercise would be to rewrite the above script using the game log data, now defining teammate to mean "all other players they took the field with over their career".

```
player_season_stats_ordered = ORDER player_season_stats BY (slugging_avg + offensive_pct) DESC;
STORE player_season_stats INTO '/tmp/baseball/player_season_stats';
```

This script will run *two* Hadoop jobs. One pass is a light mapper-only job to sample the sort key, necessary for Pig to balance the amount of data each reducer receives (we'll learn more about this in the next chapter (TODO ref). The next pass is the map/reduce job that actually sorts the data: output file `part-r-00000` has the earliest-ordered records, followed by `part-r-00001`, and so forth.

The custom partitioner of an `ORDER` statement subtly breaks the reducer contract: it may send records having the same key to different reducers. This will cause them to be in different output (`part-xxxxx`) files, so make sure anything using the sorted data doesn't assume keys uniquely correspond to files.

STORE operation serializes to disk

The STORE operation writes your data to the destination you specify (typically the HDFS).

```
articles = LOAD './data/wp/articles.tsv' AS (page_id: long, namespace: int, wikipedia_id: chararra
hadoop_articles = FILTER articles BY matches('.*[Hh]adoop.*');
STORE hadoop_articles INTO './data/tmp/hadoop_articles.tsv';
```

As with any Hadoop job, Pig creates a *directory* (not a file) at the path you specify; each task generates a file named with its task ID into that directory. In a slight difference from vanilla Hadoop, If the last stage is a reduce, the files are named like `part-r-00000` (`r` for reduce, followed by the task ID); if a map, they are named like `part-m-00000`.

Try removing the STORE line from the script above, and re-run the script. You'll see nothing happen! Pig is declarative: your statements inform Pig how it could produce certain tables, rather than command Pig to produce those tables in order.

The behavior of only evaluating on demand is an incredibly useful feature for development work. One of the best pieces of advice we can give you is to checkpoint all the time. Smart data scientists iteratively develop the first few transformations of a project, then save that result to disk; working with that saved checkpoint, develop the next few transformations, then save it to disk; and so forth. Here's a demonstration:

```
great_start = LOAD '...' AS (...);
-- ...
-- lots of stuff happens, leading up to
-- ...
important_milestone = JOIN [...];

-- reached an important milestone, so checkpoint to disk.
STORE important_milestone INTO './data/tmp/important_milestone';
    important_milestone = LOAD './data/tmp/important_milestone' AS (...schema...);
```

In development, once you've run the job past the `STORE important_milestone` line, you can comment it out to make pig skip all the preceding steps — since there's nothing tying the graph to an output operation, nothing will be computed on behalf of `important_milestone`, and so execution will start with the following `LOAD`. The gratuitous save and load does impose a minor cost, so in production, comment out both the `STORE` and its following `LOAD` to eliminate the checkpoint step.

These checkpoints bring two other benefits: an inspectable copy of your data at that checkpoint, and a description of its schema in the re-`LOAD` line. Many newcomers to Big Data processing resist the idea of checkpointing often. It takes a while to accept that a terabyte of data on disk is cheap — but the cluster time to generate that data is far less cheap, and the programmer time to create the job to create the data is most expensive of all. We won't include the checkpoint steps in the printed code snippets of the book, but we've left them in the example code.

Directives that aid development: DESCRIBE, ASSERT, EXPLAIN, LIMIT..DUMP, ILLUSTRATE

`DESCRIBE` shows the schema of a table

You've already seen the `DESCRIBE` directive, which writes a description of a table's schema to the console. It's invaluable, and even as your project goes to production you shouldn't be afraid to leave these statements in where reasonable.

`ASSERT` checks that your data is as you think it is

The `ASSERT` operation applies a test to each record as it goes by, and fails the job if the test is ever false. It doesn't create a new table, or any new map/reduce passes — it's slipstreamed into whatever operations precede it — but it does cause per-record work. The cost is worth it, and you should look for opportunities to add assertions wherever reasonable.

`DUMP` shows data on the console with great peril

The `DUMP` directive is actually equivalent to `STORE`, but (gulp) writes its output to your console. Very handy when you're messing with data at your console, but a trainwreck when you unwittingly feed it a gigabyte of data. So you should never use a `DUMP` statement except as in the following stanza: `dumpable = LIMIT table_to_dump 10; DUMP dumpable;`. (ATTN tech reviewers: should we even discuss `DUMP`? Is there a good alternative, given `ILLUSTRATE`s flakiness?)

`ILLUSTRATE` magically simulates your script's actions, except when it fails to work

The `ILLUSTRATE` directive is one of our best-loved, and most-hated, Pig operations. Even if you only want to see an example line or two of your output, using a `DUMP` or a `STORE` requires passing the full dataset through the processing pipeline. You might think, "OK, so just choose a few rows at random and run on that" — but if your job has steps that try to match two datasets using a `JOIN`, it's exceptionally unlikely that any matches will survive the limiting. (For example, the players in the first few rows of the baseball players table belonged to teams that are not in the first few rows from the baseball teams table.) `ILLUSTRATE` walks your execution graph to intelligently mock up records at each processing stage. If the sample rows would fail to join, Pig uses them to generate fake records that will find matches. It solves the problem of running on ad-hoc subsets, and that's why we love it.

However, not all parts of Pig's functionality work with ILLUSTRATE, meaning that it often fails to run. When is the `ILLUSTRATE` command is most valuable? When applied to less-widely-used operations and complex sequences of statements, of course. What parts of Pig are most likely to lack `ILLUSTRATE` support or trip it up? Well, less-widely-used operations and complex sequences of statements, of course. And when it fails, it does so with perversely opaque error messages, leaving you to wonder if there's a problem in your script or if `ILLUSTRATE` has left you short. If you, eager reader, are looking for a good place to return some open-source karma: consider making `ILLUSTRATE` into the tool it could be. Until somebody does, you should checkpoint often (described along with the `STORE` command above) and use the strategies for subuniverse sampling from the Statistics chapter (TODO ref).

Lastly, while we're on the subject of development tools that don't work perfectly in Pig: the Pig shell gets confused too easily to be useful. You're best off just running your script directly.

`EXPLAIN` shows Pig's execution graph

The `EXPLAIN` directive writes the "execution graph" of your job to the console. It's extremely verbose, showing *everything* pig will do to your data, down to the typecasting it applies to inputs as they are read. We mostly find it useful when trying to understand whether Pig has applied some of the optimizations you'll learn about in Tuning for the Wise and Lazy (TODO ref). (QUESTION for tech reviewers: move this section to advanced Pig and explain EXPLAIN?)

CHAPTER 5

Core Analytic Patterns

Now that you've met the fundamental analytic operations — in both their map/reduce and table-operation form — it's time to put them to work in an actual data exploration.

This chapter will equip you to think tactically, to think in terms of the changes you would like to make to the data. Each section introduces a repeatedly-useful data transformation pattern, demonstrated in Pig (and, where we'd like to reinforce the record-by-record action, in Wukong as well).

Geographic Flavor

There's no better example of data that is huge, unruly, organic, highly-dimensional and deeply connected than Wikipedia. Six million articles having XXX million associated properties and connected by XXX million links are viewed by XXX million people each year (TODO: add numbers). The full data — articles, properties, links and aggregated pageview statistics — is free for anyone to access it. (See the ??? for how.)

The Wikipedia community have attach the latitude and longitude to more than a million articles: not just populated places like Austin, TX, but landmarks like the University of Texas' Memorial Stadium, Snow's BBQ ("The Best Texas BBQ in the World") and the TACC (Texas Advanced Computer Center, home of the world's largest academic supercomputer). This lets us put not just each article, but the cloud of data around it, in geographical context.

What happens if we apply this context to not just the article, but those articles' words? Barbeque is popular all through Texas and the Southeastern US — is the term "Barbeque" overrepresented in articles from that region? We can brainstorm a few more terms with strong place affinity, like "beach" (the coasts) or "wine" (France, Napa Valley), and ones without, like "hat" or "couch". Hadoop, combined with the Wikipedia dataset, will let us *rigorously* identify words with this kind of geographic flavor, along with the locales they have affinity for.

At a high level, what we'll do is this:

- Divide the world into grid cells, and group all the words in wikipedia onto their article's grid cell
- Determine the overall frequency of each word in the wikipedia corpus
- Identify prominent (unusually frequent) words on each grid cell
- Identify words that are prominent on a large number of grid cells — these have strong geographic "flavor"

Match Wikipedia Article Text with Article Geolocation

Let's start by assembling the data we need. The wikipedia dataset has three different tables for each article: the metadata for each page (page id, title, size, last update time, and so); the full text of each article (a very large field); and the article geolocations. Below are snippets from the articles table and of the geolocations table:

Wikipedia article record for "Lexington, Texas".

```
Lexington,_Texas  Lexington is a town in Lee County, Texas, United States. ... Snow's BBQ, which T
```

Article coordinates.

```
Lexington,_Texas -97.01 30.41 023130130
```

Since we want to place the words in each article in geographic context, our first step is to reunite each article with its geolocation.

```
article_text = LOAD ('...');
article_geolocations = LOAD('...');
articles = JOIN article_geolocations BY page_id, article_text BY page_id;
articles = FOREACH articles GENERATE article_text::page_id, QUADKEY(lng, lat) as quadcell, text;
```

The quadkey field you see is a label for the grid cell containing an article's location; you'll learn all about them in the "Geographic Data" chapter, but for the moment just trust us that it's a clever way to divide up the world map for big data computation. Here's the result:

Wordbag with coordinates.

```
Lexington,_Texas 023130130 Lexington is a town in Lee County, Texas ...
```

Summarize Wikipedia Articles

Next, we will summarize each article by preparing its "word bag" — a simple count of the terms on its wikipedia page. We've written a Pig UDF (User-Defined Function) to do so:

```
REGISTER path/to/udf ...;
wordbags = FOREACH articles {
    wds = WORDBAG(text)
      AS tuple(tot_usages:long, num_terms:long, num_onces:long,
         wordbag:bag{tuple(article_term_usages,term)});
    GENERATE page_id, quadcell, wds.tot_usages, wds.num_terms, wds.num_onces, wds.wordbag;
};
term_article_freqs = FOREACH wordbags GENERATE
    page_id, quadcell, tot_usages, num_terms, FLATTEN(wordbag);
STORE term_article_freqs INTO '/data/work/geo_flavor/term_article_freqs;
```

Here's the output:

Wordbag for "Lexington, Texas".

```
Lexington,_Texas 023130130 TODO:tot_usages TODO:terms TODO:NUMONCES {(4,"texas")(2,"lexington"),(2
```

And the output after the flatten:

"term_cell_freqs" result.

```
Lexington,_Texas 023130130 tot_usages num_terms 4   "texas"
Lexington,_Texas 023130130 tot_usages num_terms 2   "lexington"
Lexington,_Texas 023130130 tot_usages num_terms 2   "best"
Lexington,_Texas 023130130 tot_usages num_terms 2   "bbq"
Lexington,_Texas 023130130 tot_usages num_terms 1   "barbecue"
```

Pattern: Atom-only Records

All of the fields in the table we've just produced are atomic — strings, numbers and such, rather than bags or tuples — what database wonks call "First Normal Form". There is a lot of denormalization (each article's quadcell and total term count are repeated for every term in the article), but the simplicity of each record's schema has a lot of advantages.

Think of this atom-only form as the neutral fighting stance for your tables. From here we're ready to put each record into context of other records with the same term, same geolocation, same frequency; we can even reassemble the wordbag by grouping on the page_id. The exploration will proceed from here by reassembling these records into various context groups, operating on those groups, and then expanding the result back into atom-only form.

Term Statistics by Grid Cell

```
taf_g = GROUP term_article_freqs BY quadcell, term;
cell_freqs = FOREACH taf_g GENERATE
      group.quadcell AS quadcell,
      group.term AS term,
      SUM(term_article_freqs.article_term_usages) AS cell_term_usages;
cf_g = GROUP cell_freqs BY quadcell;
term_cell_freqs = FOREACH cf_g GENERATE
```

```
    group AS quadcell,
    SUM(cell_term_usages) AS cell_usages
    FLATTEN(cell_term_usages, term);
```

"cell_freqs" result.

```
023130130 7 "bbq"
023130130 20 "texas"
```

"cf_g" result.

```
023130130 {(7,"bbq"),(20,"texas"),...}
```

"term_cell_freqs" result.

```
023130130 95 7 "bbq"
023130130 95 20 "texas"
```

Term Statistics

We will be defining the prominence of a term on a grid cell by comparing its local frequency to the overall frequency of the term. The occurrence frequency of the term "the" is XX parts per million (ppm), while that of "barbeque"'s is XX ppm. However, on the quadcell surrounding Lexington, Texas, "the" occurs at XX ppm and "barbeque" at XX ppm — a significantly elevated rate.

Let's now prepare those global statistics.

```
all_terms = GROUP term_article_freqs BY term;
term_info_1 = FOREACH all_terms GENERATE
    group AS term,
    COUNT_STAR(term_article_freqs) AS num_articles,
    SUM(article_term_usages) AS term_usages;
global_term_info_g = GROUP term_info BY ALL;
global_term_info = FOREACH global_term_info_g GENERATE
    COUNT_STAR(term_info) AS num_terms,
    SUM(term_usages) AS global_usages;
STORE global_term_info INTO '/data/work/geo_flavor/global_term_info';
```

(The actual code is somewhat different from what you see here — we'll explain below)

i. (TODO describe term_info)

GROUP/COGROUP To Restructure Tables

This next pattern is one of the more difficult to picture but also one of the most important to master. Once you can confidently recognize and apply this pattern, you can consider yourself a black belt in the martial art of Map/Reduce.

(TODO: describe this pattern)

Pattern: Extend Records with Uniquely Matching Records from Another Table

Using a join as we just did — to extend the records in one table with the fields from one matching record in another — is a very common pattern. Datasets are commonly stored as tables in *normalized* form — that is, having tables structured to minimize redundancy and dependency. The global hourly weather dataset has one table giving the metadata for every weather station: identifiers, geocoordinates, elevation, country and so on. The giant tables listing the hourly observations from each weather station are normalized to not repeat the station metadata on each line, only the weather station id. However, later in the book (REF) we'll do geographic analysis of the weather data — and one of the first tasks will be to denormalize the geocoordinates of each weather station with its observations, letting us group nearby observations.

Another reason to split data across tables is *vertical partitioning*: storing fields that are very large or seldom used in context within different tables. That's the case with the Wikipedia article tables — the geolocation information is only relevant for geodata analysis; the article text is both large and not always relevant.

Pattern: Summarizing Groups

Pretty much every data exploration you perform will involve summarizing datasets using statistical aggregations — counts, averages and so forth. You have already seen an example of this when we helped the reindeer count UFO visit frequency by month and later in the book, we will devote a whole chapter to statistical summaries and aggregation.

Pattern: Re-injecting global totals

We also extract two global statistics: the number of distinct terms, and the number of distinct usages. This brings up one of the more annoying things about Hadoop programming. The global_term_info result is two lousy values, needed to turn the global *counts* for each term into the global *frequency* for each term. But a pig script just orchestrates the top-level motion of data: there's no intrinsic way to bring the result of a step into the declaration of following steps. The proper recourse is to split the script into two parts, and run it within a workflow tool like Rake, Drake or Oozie. The workflow layer can fish those values out of the HDFS and inject them as runtime parameters into the next stage of the script.

We prefer to cheat. We instead ran a version of the script that found the global count of terms and usages, then copy/pasted their values as static parameters at the top of the script. This also lets us calculate the ppm frequency of each term and the other term

statistics in a single pass. To ensure our time-traveling shenanigans remain valid, we add an ASSERT statement which compares the memoized values to the actual totals.

```
DEFINE memoized_num_terms XXX;
DEFINE memoized_global_usages XXX;
all_terms = GROUP term_cell_freqs BY term;
term_info_1 = FOREACH all_terms GENERATE
    group AS term,
    COUNT_STAR(term_cell_freqs) AS num_articles,
    SUM(article_term_usages) AS term_usages,
    1000000 * SUM(article_term_usages)/memoized_global_usages AS term_ppm:double
    ;
-- Validate the global term statistics
global_term_info_g = GROUP term_info BY ALL;
global_term_info = FOREACH global_term_info_g GENERATE
   COUNT_STAR(term_info) AS num_terms,
   SUM(term_usages) AS global_usages;
STORE global_term_info INTO '/data/work/geo_flavor/global_term_info';
ASSERT(global_term_info.num_terms = memoized_num_terms);
ASSERT(global_term_info.global_usages = memoized_global_usages);
```

(TODO: just realized the way we've done this finds global term stats on only geolocated articles. To find them on all articles will complicate the script: we have to do a left join and then filter, or we'd have to do wordbags first then join on geolocations.)

Select a Fixed Number of Arbitrary Records (LIMIT)

The Pig LIMIT operation arbitrarily selects, at most, the specified number of records from a table.

```
(TODO: example)
```

(TODO: Is there a non-Reduce way to do this?)

In the simplest Map/Reduce equivalent, Mappers emit each record unchanged until they hit the specified limit (or reach the end of their input). Those output records are sent to a single Reducer, which itself emits each record unchanged until it has hit the specified limit and does nothing on all subsequent records.

(TODO: Do we want to talk about a non-single Reducer approach?)

A Combiner is helpful here in the predominant case where the specified limit is small, as it will eliminate excess records before they are sent to the Reducer and at each merge/sort pass.

Top K Records (ORDER..LIMIT)

The naive way to extract the top K elements from a table is simply to do an ORDER and then a LIMIT. For example, the following script will identify the top 100 URLs from the waxy.org weblog dataset.

```
logs=LOAD '/data/gold/waxy/whatever.log' AS (...) USING APACHE LOG READER;
logs=FOREACH logs GENERATE url;
url_logs = GROUP logs BY url;
URL_COUNTS=FOREACH url_logs GENERATE
    COUNT_STAR(url_logs) AS views,
    group AS url;
url_counts_o = ORDER url_counts BY views PARALLEL 1;
top_url_counts = LIMIT url_counts_o 100;
STORE top_url_counts INTO '/data/out/weblogs/top_url_counts';
```

There are two useful optimizations to make when K (the number of records you will keep) is much less than N (the number of records in the table). The first one, which Pig does for you, is to only retain the top K records at each Mapper; this is a great demonstration of where a Combiner is useful: After each intermediate merge/sort on the Map side and the Reduce side, the Combiner discards all but the top K records.

A Foolish Optimization

We will tell you about another "optimization," mostly because we want to illustrate how a naive performance estimation based on theory can lead you astray in practice. In principle, sorting a large table in place takes *O(N log N)* time. In a single compute node context, you can actually find the top K elements in *O(N log K)* time — a big savings since K is much smaller than N. What you do is maintain a heap structure; for every element past the Kth, if it is larger than the smallest element in the heap, remove the smallest member of the heap and add the element to the heap. While it is true that *O(N log K)* beats *O(N log N)*, this reasoning is flawed in two ways. First, you are not working in a single-node context; Hadoop is going to perform that sort anyway. Second, the fixed costs of I/O almost always dominate the cost of compute (FOOTNOTE: Unless you are unjustifiably fiddling with a heap in your Mapper.)

The *O(log N)* portion of Hadoop's log sort shows up in two ways: The N memory sort that precedes a spill is *O(N log N)* in compute time but less expensive than the cost of spilling the data. The true *O(N log N)* cost comes in the reducer: *O(log N)* merge passes, each of cost *O(N)*. [1]. But K is small, so there should not be multiple merge passes; the actual runtime is *O(N)* in disk bandwidth. Avoid subtle before-the-facts reasoning about performance; run your job, count the number of merge passes, weigh your salary against the costs of the computers you are running on, and only then decide if it is worth optimizing.

1. If initial spills have M records, each merge pass combines B spills into one file, and we can skip the last merge pass, the total time is `N (log_B(N/M)-1).` [TODO: double check this

Top K Within a Group

There is a situation where the heap-based top K algorithm is appropriate: finding the top K elements for a group. Pig's *top* function accepts a bag and returns a bag with its top K elements. Here is an example that uses the World Cup dataset to find the top 10 URLs for each day of the tournament:

```
visits = load ('worldcup');
visits = FOREACH visits generate day, url;
visits by day = GROUP visits by day;
top visits by day = FOREACH visits url = GROUP visits by url;
   generate GROUP as day, top (visits, top visit URLs, COUNT_STAR (visit urls), 100;
store top visits by url into 'top visits by url';
```

CHAPTER 6
Text Data

…

CHAPTER 7
Hadoop Internals: Just Enough for Now

TODO: write Intro

TODO: Move the part about tiny files to here TODO: See if other parts of the Hadoop tuning or internals chapters should move here or v/v

```
 3a: Hadoop Internals and Performance: Just Enough for Now
. the HDFS at moderate detail
. ??? job orchestration/lifecycle at light detail
  - how a job is born
```

The harsh realities of the laws of physics and economics prevent traditional data analysis solutions such as relational databases, supercomputing and so forth from economically scaling to arbitrary-sized data. By comparison, Hadoop's Map/Reduce paradigm does not provide complex operations, modification of existing records, fine-grain control over how the data is distributed or anything else beyond the ability to write programs that adhere to a single, tightly-constrained template. If Hadoop were a publishing medium, it would be one that refused essays, novels, sonnets and all other literary forms beyond the haiku:

```
data flutters by
elephants make sturdy piles
context yields insight
```

Our Map/Reduce haiku illustrates Hadoop's template:

1. The Mapper portion of your script processes records, attaching a label to each.
2. Hadoop assembles those records into context groups according to their label.
3. The Reducer portion of your script processes those context groups and writes them to a data store or external system.

What is remarkable is that from this single primitive, we can construct the familiar relational operations (such as `GROUP`s and `ROLLUP`s) of traditional databases,

many machine-learning algorithms, matrix and graph transformations and the rest of the advanced data analytics toolkit. In the next two chapters, we will demonstrate high-level relational operations and illustrate the Map/Reduce patterns they express. In order to understand the performance and reasoning behind those patterns, let's first understand the motion of data within a Map/Reduce job.

The HDFS: Highly Durable Storage Optimized for Analytics

The HDFS, as we hope you've guessed, holds the same role within Hadoop that Nanette and her team of elephants do within C&E Corp. It ensures that your data is always available for use, never lost or degraded and organized to support efficient Map/Reduce jobs. Files are stored on the HDFS as blocks of limited size (128 MB is a common choice). Each block belongs to exactly one file; a file larger than the block size is stored in multiple blocks. The blocks are stored in cooked form as regular files on one of the Datanode's regular volumes. (Hadoop's decision to use regular files rather than attempting lower-level access to the disk, as many traditional databases do, helps make it remarkably portable, promotes reliability and plays to the strengths of the operating system's finely-tuned access mechanisms.)

The HDFS typically stores multiple replicas of each block (three is the universal default, although you can adjust it per file), distributed across the cluster. Blocks within the same file may or may not share a Datanode but replicas never do (or they would not be replicas, would they?). The obvious reason for this replication is availability and durability — you can depend on finding a live Datanode for any block and you can depend that, if a Datanode goes down, a fresh replica can be readily produced.

JT and Nanette's workflow illustrates the second benefit of replication: being able to "move the compute to the data, not [expensively] moving the data to the compute." Multiple replicas give the Job Tracker enough options that it can dependably assign most tasks to be "Mapper-local."

A special node called the *Namenode* is responsible for distributing those blocks of data across the cluster. Like Nanette, the Namenode holds no data, only a sort of file allocation table (FAT), tracking for every file the checksum responsible Datanodes and other essential characteristics of each of its blocks. The Namenode depends on the Datanodes to report in regularly. Every three seconds, it sends a heartbeat — a lightweight notification saying, basically, "I'm still here!". On a longer timescale, each Datanode prepares a listing of the replicas it sees on disk along with a full checksum of each replica's contents. Having the Datanode contact the Namenode is a good safeguard that it is operating regularly and with good connectivity. Conversely, the Namenode uses the heartbeat response as its opportunity to issue commands dening a struggling Datanode.

If, at any point, the Namenode finds a Datanode has not sent a heartbeat for several minutes, or if a block report shows missing or corrupted files, it will commission new

copies of the affected blocks by issuing replication commands to other Datanodes as they heartbeat in.

A final prominent role the Namenode serves is to act as the public face of the HDFS. The 'put' and 'get' commands you just ran were Java programs that made network calls to the Namenode. There are API methods for the rest of the file system commands you would expect for use by that or any other low-level native client. You can also access its web interface, typically by visiting port 50070 (`http://hostname.of.namenode:50070`), which gives you the crude but effective ability to view its capacity, operational status and, for the very patient, inspect the contents of the HDFS.

Sitting behind the scenes is the often-misunderstood secondary Namenode; this is not, as its name implies and as you might hope, a hot standby for the Namenode. Unless you are using the "HA namenode" feature provided in later versions of Hadoop, if your Namenode goes down, your HDFS has gone down. All the secondary Namenode does is perform some essential internal bookkeeping. Apart from ensuring that it, like your Namenode, is *always* running happily and healthily, you do not need to know anything more about the second Namenode for now.

One last essential to note about the HDFS is that its contents are immutable. On a regular file system, every time you hit "save," the application modifies the file in place — on Hadoop, no such thing is permitted. This is driven by the necessities of distributed computing at high scale but it is also the right thing to do. Data analysis should proceed by chaining reproducible syntheses of new beliefs from input data. If the actions you are applying change, so should the output. This casual consumption of hard drive resources can seem disturbing to those used to working within the constraints of a single machine, but the economics of data storage are clear; it costs $0.10 per GB per month at current commodity prices, or one-tenth that for archival storage, and at least $50 an hour for the analysts who will use it.

Possibly the biggest rookie mistake made by those new to Big Data is a tendency to economize on the amount of data they store; we will try to help you break that habit. You should be far more concerned with the amount of data you send over the network or to your CPU than with the amount of data you store and most of all, with the amount of time you spend deriving insight rather than acting on it. Checkpoint often, denormalize when reasonable and preserve the full provenance of your results. === JT and Nanette at Work

JT and Nanette work wonderfully together — JT rambunctiously barking orders, Nanette peacefully gardening her card catalog — and subtly improve the efficiency of their team in a variety of ways. We'll look closely at their bag of tricks later in the book (TODO ref) but here are two. The most striking thing any visitor to the worksite will notice is how *calm* everything is. One reason for this is Nanette's filing scheme, which designates each book passage to be stored by multiple elephants. Nanette quietly advises JT of each passage's location, allowing him to almost always assign his chimpanzees a passage held

by the librarian in their cubicle. In turn, when an elephant receives a freshly-translated scroll, she makes two photocopies and dispatches them to two other cubicles. The hallways contain a stately parade of pygmy elephants, each carrying an efficient load; the only traffic consists of photocopied scrolls to store and the occasional non-cubicle-local assignment.

The other source of calm is on the part of their clients, who know that when Nanette's on the job, their archives are safe — the words of Shakespeare will retain their eternal form [1] To ensure that no passage is never lost, the librarians on Nanette's team send regular reports on the scrolls they maintain. If ever an elephant doesn't report in (whether it stepped out for an hour or left permanently), Nanette identifies the scrolls designated for that elephant and commissions the various librarians who hold other replicas of that scroll to make and dispatch fresh copies. Each scroll also bears a check of authenticity validating that photocopying, transferring its contents or even mouldering on the shelf has caused no loss of fidelity. Her librarians regularly recalculate those checks and include them in their reports, so if even a single letter on a scroll has been altered, Nanette can commission a new replica at once.

SIDEBAR: What's Fast At High Scale

Measure	Nominal Throughput		Time to handle 1M 1kB records		Millions of 1kB Records Handled per hour		Cost to Process One Billion records	
Processor Throughput (raw)	500,000	MB/sec	0.002	sec/Mrec	1,800,000	Mrec/hr	$ 0.001	$/Brec
Main Memory Throughput	80,000	MB/sec	0.013	sec/Mrec	288,000	Mrec/hr	$ 0.003	$/Brec
Main memory Fetch	2,500	MB/sec	0.400	sec/Mrec	9,000	Mrec/hr	$ 0.111	$/Brec
One Gigabyte/s Throughput	1,000	MB/sec	1	sec/Mrec	3,600	Mrec/hr	$ 0.278	$/Brec
Processor Throughput (practical)	500	MB/sec	2	sec/Mrec	1,800	Mrec/hr	$ 0.556	$/Brec
Main Bus Throughput (raw)	300	MB/sec	3	sec/Mrec	1,080	Mrec/hr	$ 1	$/Brec
SSD Throughput	300	MB/sec	3	sec/Mrec	1,080	Mrec/hr	$ 1	$/Brec
Network Throughput	125	MB/sec	8	sec/Mrec	450	Mrec/hr	$ 2	$/Brec
Disk Throughput	100	MB/sec	10	sec/Mrec	360	Mrec/hr	$ 3	$/Brec
SSD Random Fetch	25	MB/sec	40	sec/Mrec	90	Mrec/hr	$ 11	$/Brec
Network Request (Same Datacenter)	2	MB/sec	500	sec/Mrec	7	Mrec/hr	$ 139	$/Brec
Network Request (Boston-Wash DC)	0.083	MB/sec	12,000	sec/Mrec	0.300	Mrec/hr	$ 3,333	$/Brec
Disk Random Fetch	0.083	MB/sec	12,000	sec/Mrec	0.300	Mrec/hr	$ 3,333	$/Brec
Network Request (California-Europe)	0.007	MB/sec	150,000	sec/Mrec	0.024	Mrec/hr	$ 41,667	$/Brec

Approximate Cost to Host and Serve One Billion 1kB Records (1TB), per month

		notes	GB / host	host $/hr	$ / TB.month	GB hr / $
Serve 1 TB from RAM	Amazon m2.2xlarge	-	34	$ 0.820	$ 17,407	42
Serve 1 TB from RAM	Amazon cr1.8xlarge	-	244	$ 3.500	$ 10,414	70
Serve 1 TB from SSD	Amazon i2.8xlarge	-	6400	$ 6.820	$ 774	938
Serve 1 TB from local disk, 3x replication	Amazon m1.xlarge	3x replication	560	$ 0.480	$ 622	1,167
Serve 1 TB from local disk, 10% ram-backed	Amazon m1.xlarge	-	1680	$ 0.480	$ 207	3,500
Serve 1 TB from Amazon S3	Amazon S3	exceptional durability	1000	$ 0.110	$ 80	9,075
Serve 1 TB from storage-optimized instance	Amazon hs1.8xlarge	-	49,152	$ 4.600	$ 68	10,685
Store 1 TB on Amazon Glacier	Amazon Glacier	exceptional durability	1000	$ 0.014	$ 10	72,600
Store 1 TB on 3TB HDD, drive cost only, 2 yr	Seagate Constellation ES.2	drive cost only	3000	$ 0.015	$ 4	201,600

The table at the right (REF) summarizes the 2013 values for Peter Norvig's "Numbers Every Programmer Should Know." — the length of time for each computation primitive on modern hardware. We've listed the figures several different ways: as latency (time to execute); as the number of 500-byte records that could be processed in an hour (TODO:

1. When Nanette is not on the job, it's a total meltdown — a story for much later in the book. But you'd be wise to always take extremely good care of the Nanettes in your life.

day), if that operation were the performance bottleneck of your process; and as an amount of money to process one billion records of 500-byte each on commodity hardware. Big Data requires high volume, high throughput computing, so our principle bound is the speed at which data can be read from and stored to disk. What is remarkable is that with the current state of technology, most of the other operations are slammed to one limit or the other: either bountifully unconstraining or devastatingly slow. That lets us write down the following "rules for performance at scale:"

- High throughput programs cannot run faster than x (TODO: Insert number)
- Data can be streamed to and from disk at x GB per hour (x records per hour, y records per hour, z dollars per billion records) (TODO: insert numbers)
- High throughput programs cannot run faster than that but not run an order of magnitude slower.
- Data streams over the network at the same rate as disk.
- Memory access is infinitely fast.
- CPU is fast enough to not worry about except in the obvious cases where it is not.
- Random access (seeking to individual records) on disk is unacceptably slow.
- Network requests for data (anything involving a round trip) is infinitely slow.
- Disk capacity is free.
- CPU and network transfer costs are cheap.
- Memory is expensive and cruelly finite. For most tasks, available memory is either all of your concern or none of your concern.

Now that you know how Hadoop moves data around, you can use these rules to explain its remarkable scalability.

1. Mapper streams data from disk and spills it back to disk; cannot go faster than that.
2. In between, your code processes the data
3. If your unwinding proteins or multiplying matrices are otherwise CPU or memory bound, Hadoop at least will not get in your way; the typical Hadoop job can process records as fast as they are streamed.
4. Spilled records are sent over the network and spilled back to disk; again, cannot go faster than that.

That leaves the big cost of most Hadoop jobs: the midstream merge-sort. Spilled blocks are merged in several passes (at the Reducer and sometimes at the Mapper) as follows. Hadoop begins streaming data from each of the spills in parallel. Under the covers, what this means is that the OS is handing off the contents of each spill as blocks of memory

in sequence. It is able to bring all its cleverness to bear, scheduling disk access to keep the streams continually fed as rapidly as each is consumed.

Hadoop's actions are fairly straightforward. Since the spills are each individually sorted, at every moment the next (lowest ordered) record to emit is guaranteed to be the next unread record from one of its streams. It continues in this way, eventually merging each of its inputs into an unbroken output stream to disk. At no point does the Hadoop framework require a significant number of seeks on disk or requests over the network; the memory requirements (the number of parallel streams times the buffer size per stream) are manageable; and the CPU burden is effectively nil, so the merge/sort as well runs at the speed of streaming to disk.

Hadoop Output phase may be more expensive than you think

As your Reducers emit records, they are streamed directly to the job output, typically the HDFS or S3. Since this occurs in parallel with reading and processing the data, the primary spill to the Datanode typically carries minimal added overhead. However, the data is simultaneously being replicated as well, which can extend your job's runtime by more than you might think.

TODO-qem — a) does this section belong in "06a jsut enough performance for now" or here? b) I think this is a really good point to hit so want it to be really clear; apply extra criticism here.

Let's consider how data flows in a job intended to remove duplicate records: for example, processing 100 GB of data with one-percent duplicates, and writing output with replication factor three. As you'll see when we describe the *distinct* patterns in Chapter 5 (REF), the Reducer input is about the same size as the mapper input. Using what you now know, Hadoop moves roughly the following amount of data, largely in parallel:

- 100 GB of Mapper input read from disk;
- 100 GB spilled back to disk;
- 100 GB of Reducer input sent and received over the network;
- 100 GB of Reducer input spilled to disk
- some amount of data merge/sorted to disk if your cluster size requires multiple passes;
- 100 GB of Reducer output written to disk by the local Datanode;
- 200 GB of replicated output sent over the network, received over the network and written to disk by the Datanode.

If your Datanode is backed by remote volumes (common in some virtual environments [2]), you'll additionally incur

- 300 GB sent over the network to the remote file store

As you can see, unless your cluster is undersized (producing significant merge/sort overhead), the cost of replicating the data rivals the cost of the rest of the job. The default replication factor is 3 for two very good reasons: it helps guarantee the permanence of your data and it allows the Job tracker to efficiently allocate Mapper-local tasks. But in certain cases — intermediate checkpoint data, scratch data or where backed by a remote file system with its own durability guarantee — an expert who appreciates the risk can choose to reduce the replication factor to 2 or 1.

2. This may sound outrageous to traditional IT folk, but the advantages of elasticity are extremely powerful — we'll outline the case for virtualized Hadoop in Chapter (REF)

CHAPTER 8

Big Data Ecosystem and Toolset

Big data is necessarily a polyglot sport. The extreme technical challenges demand diverse technological solutions and the relative youth of this field means, unfortunately, largely incompatible languages, formats, nomenclature and transport mechanisms. What's more, every ecosystem niche has multiple open source and commercial contenders vying for prominence and it is difficult to know which are widely used, which are being adopted and even which of them work at all.

Fixing a map of this ecosystem to print would be nearly foolish; predictions of success or failure will prove wrong, the companies and projects whose efforts you omit or downplay will inscribe your name in their "Enemies" list, the correct choice can be deeply use-case specific and any list will become out of date the minute it is committed to print. Your authors, fools both, feel you are better off with a set of wrong-headed, impolitic, oblivious and obsolete recommendations based on what has worked for us and what we have seen work for other people.

Core Platform: Batch Processing

Hadoop is the one easy choice to make; Doug Cutting calls it the "kernel of the big data operating system" and we agree. It can't be just that easy, though; you further have to decide which distribution to adopt. Almost everyone should either choose Cloudera's distribution (CDH) or Hortonworks' (HDP). Both come in a complete, well-tested open source version as well as a commercial version backed by enterprise features and vendor support. Both employ significant numbers of core contributors, have unshakable expertise and open-source credibility.

Cloudera started in 2009 with an all-star list of founders including Doug Cutting, the originator of the Hadoop project. It was the first to offer a packaged distribution and the first to offer commercial support; its offerings soon came to dominate the ecosystem.

Hortonworks was founded two years later by Eric Baldeschwieler (aka Eric 14), who brought the project into Yahoo! and fostered its essential early growth, and a no-less-impressive set of core contributors. It has rapidly matured a first-class offering with its own key advantages.

Cloudera was the first company to commercialize Hadoop; it's distribution is, by far, the most widely adopted and if you don't feel like thinking, it's the easy choice. The company is increasingly focused on large-scale enterprise customers and its feature velocity is increasingly devoted to its commercial-only components.

Hortonworks' offering is 100-percent open source, which will appeal to those uninterested in a commercial solution or who abhor the notion of vendor lock-in. More impressively to us has been Hortonworks' success in establishing beneficial ecosystem partnerships. The most important of these partnerships is with Microsoft and, although we do not have direct experience, any Microsoft shop should consider Hortonworks first.

There are other smaller distributions, from IBM VMware and others, which are only really interesting if you use IBM VMware or one of those others.

The core project has a distribution of its own, but apart from people interested in core development, you are better off with one of the packaged distributions.

The most important alternative to Hadoop is Map/R, a C++-based rewrite, that is 100-percent API compatible with Hadoop. It is a closed-source commercial product for high-end Enterprise customers and has a free version with all essential features for smaller installations. Most compellingly, its HDFS also presents an NFS interface and so can be mounted as a native file system. See the section below (TODO: REF) on why this is such a big deal. Map/R is faster, more powerful and much more expensive than the open source version, which is pretty much everything you need to know to decide if it is the right choice for you.

There are two last alternatives worthy of note. Both discard compatibility with the Hadoop code base entirely, freeing them from any legacy concerns.

Spark is, in some sense, an encapsulation of the iterative development process espoused in this book: prepare a sub-universe, author small self-contained scripts that checkpoint frequently and periodically reestablish a beachhead by running against the full input dataset. The output of Spark's Scala-based domain-specific statements are managed intelligently in memory and persisted to disk when directed. This eliminates, in effect, the often-unnecessary cost of writing out data from the Reducer tasks and reading it back in again to Mapper tasks. That's just one example of many ways in which Spark is able to impressively optimize development of Map/Reduce jobs.

Disco is an extremely lightweight Python-based implementation of Map/Reduce that originated at Nokia. [1] Its advantage and disadvantage is that it is an essentials-only realization of what Hadoop provides whose code base is a small fraction of the size.

We do not see either of them displacing Hadoop but since both are perfectly happy to run on top of any standard HDFS, they are reasonable tools to add to your repertoire.

Sidebar: Which Hadoop Version?

At the time of writing, Hadoop is at a crossroads between versions with fundamental differences in architecture and interface. Hadoop is, beyond a doubt, one of the greatest open source software success stories. It's co-base has received contributions from thousands of committers, operators and users.

But, as Hadoop entered Enterprise-adoption adulthood from its Web-2.0 adolescence, the core team determined that enough early decisions — in naming, in architecture, in interface — needed to be remade to justify a rewrite. The project was split into multiple pieces: principally, Map/Reduce (processing data), HDFS (storing data) and core essential code shared by each.

Under the hood, the 2.0 branch still provides the legacy architecture of Job Tracker/Namenode/SecondaryNamenode masters but the way forward is a new componentized and pluggable architecture. The most significant flaw in the 1.0 branch — the terrifying lack of Namenode redundancy — has been addressed by a Zookeeper-based "high availability" implementation. (TODO: Describe YARN, Distributed Job Tracker and so forth). At the bottom, the name and meaning of Hadoop's hundreds of configuration variables have been rethought; you can find a distressingly-long Rosetta Stone from old to new at (TODO: add link).

Even more important are the changes to interface. The HDFS is largely backward-compatible; you probably only need to recompile. The 2.0 branch offers an "MR1" toolkit — backward compatible with the legacy API — and the next-generation "MR2" toolkit that takes better advantage of 2.0's new architecture. Programs written for "MR1" will not run on "MR2" and vice versa.

So, which should you choose? The way forward is clearly with the 2.0's architecture. If you are just ramping up on Hadoop, use the componentized YARN-based systems from the start. If you have an existing legacy installation, plan to upgrade at at a deliberate pace — informed exactly by whether you are more terrified of having a Namenode fail or being an early-ish adopter. For the Map/Reduce toolkit, our best advice is to use the approach described in this book: Do not use the low-level API. Pig, Hive, Wukong and most other high-level toolkits are fully compatible with each. Cascading is not yet compatible with "MR2" but likely will if the market moves that way.

1. If these analogies help, you could consider the Leica to Hadoop's Canon or the nginX to its Apache.

The 2.0 branch has cleaner code, some feature advantages and has the primary attention of the core team. However, the "MR1" toolkit has so much ecosystem support, documentation, applications, lessons learned from wide-scale deployment, it continues to be our choice for production use. Note that you can have your redundant Namenode without having to adopt the new-fangled API. Adoption of "MR2" is highly likely (though not certain); if you are just starting out, adopting it from the start is probably a sound decision. If you have a legacy investment in "MR1" code, wait until you start seeing blog posts from large-scale deployers titled "We Spent Millions Upgrading To MR2 And Boy Are We Ever Happy We Did So."

The biggest pressure to move forward will be Impala, which requires the "MR2" framework. If you plan to invest in Impala heavily, it is probably best to uniformly adopt "MR2."

Core Platform: Streaming Data Processing

While the batch processing landscape has largely settled around Hadoop, there are many more data streaming data processing technologies vying for mind share. Roughly speaking, we see three types of solutions: Complex Event Processing (CEP) systems that grew out of high-frequency trading and security applications; Streaming Transport systems, which grew out of the need to centralize server logs at extremely high throughput; and Streaming Analytics systems, developed to perform sophisticated analysis of high-rate activity streams.

The principal focus of a CEP is to enable time-windowed predicates on ordered streams — for example, "Trigger a buy order for frozen orange juice futures if Mortimer & Mortimer has sold more than 10,000 shares in the last hour" or "Lock down system access if a low-level Army analyst's terminal is accessing thousands of State Department memos." These platforms are conventionally programmed using a SQL-like query language but support low-level extension and an ecosystem of expensive consultants to write same.

These platforms are relentlessly focused on low latency, which is their gift and their curse. If you are looking for tightly-bound response times in the milliseconds, nothing else will do. Its cost is a tightly-constrained programming model, poor tolerance for strongly-disordered data and a preference for high-grade hardware and expensive commercial software. The leading open source entrant is Esper, which is Java-based and widely used. Commercial offerings include (TODO: find out what commercial offerings there are, e.g. Tibco and Streambase).

Most people with petabyte-scale data first have to figure out how to ship terabyte-scale data to their cluster. The best solutions here are Kafka or Flume. Kafka, a Java-based open source project from LinkedIn, is our choice. It is lightweight, increasingly well-adopted and has a wonderful architecture that allows the operating system to efficiency

do almost all the work. Flume, from Cloudera and also Java-based, solves the same problem but less elegantly, in our opinion. It offers the ability to do rudimentary in-stream processing similar to Storm but lacks the additional sophistication Trident provides.

Both Kafka and Flume are capable of extremely high throughput and scalability. Most importantly, they guarantee "at least once" processing. Within the limits of disk space and the laws of physics, they will reliably transport each record to its destination even as networks and intervening systems fail.

Kafka and Flume can both deposit your data reliably onto an HDFS but take very different approaches to doing so. Flume uses the obvious approach of having an "always live" sync write records directly to a DataNode acting as a native client. Kafka's Camus add-on uses a counterintuitive but, to our mind, superior approach. In Camus, data is loaded onto the HDFS using Mapper-Only MR jobs running in an endless loop. Its Map tasks are proper Hadoop jobs and Kafka clients and elegantly leverage the reliability mechanisms of each. Data is live on the HDFS as often as the Import job runs — not more, not less.

Flume's scheme has two drawbacks: First, the long-running connections it requires to individual DataNodes silently compete with the traditional framework. [2] Second, a file does not become live on the HDFS until either a full block is produced or the file is closed. That's fine if all your datastreams are high rate, but if you have a range of rates or variable rates, you are forced to choose between inefficient block sizes (larger NameNode burden, more Map tasks) or exceedingly long delays until data is ready to process. There are workarounds but they are workarounds.

Both Kafka and Flume have evolved into general purpose solutions from their origins in high-scale server log transport but there are other use-case specific technologies. You may see Scribe and S4 mentioned as alternatives but they are not seeing the same widespread adoption. Scalable message queue systems such as AMQP, RabbitMQ or Kestrel will make sense if (a) you are already using one; (b) you require complex event-driven routing; or (c) your system is zillions of sources emitting many events rather than many sources emitting zillions of events. AMQP is Enterprise-y and has rich commercial support. RabbitMQ is open source-y and somewhat more fresh. Kestrel is minimal and fast.

Stream Analytics

The streaming transport solutions just described focus on getting your data from here to there as efficiently as possible. A streaming analytics solution allows you to perform, well, analytics on the data in flight. While a transport solution only guarantees *at least*

2. Make sure you increase DataNode handler counts to match.

once processing, frameworks like Trident guarantee *exactly once* processing, enabling you to perform aggregation operations. They encourage you to do anything to the data in flight that Java or your high-level language of choice permits you to do — including even high-latency actions such as pinging an external API or legacy data store — while giving you efficient control over locality and persistence. There is a full chapter introduction to Trident in Chapter (TODO: REF), so we won't go into much more detail here.

Trident, a Java and Clojure-based open source project from Twitter, is the most prominent so far.

There are two prominent alternatives. Spark Streaming, an offshoot of the Spark project mentioned above (TODO: REF), is receiving increasing attention. Continuity offers an extremely slick developer-friendly commercial alternative. It is extremely friendly with HBase (the company was started by some members of the HBase core team); as we understand it, most of the action actually takes place within HBase, an interesting alternative approach.

Trident is extremely compelling, the most widely used, is our choice for this book and our best recommendation for general use.

Online Analytic Processing (OLAP) on Hadoop

The technologies mentioned so far, for the most part, augment the mature, traditional data processing tool sets. There are now arising Hadoop-based solutions for online analytic processing (OLAP) that directly challenge the data warehousing technologies at the core of most large-scale enterprises. These rely on keeping a significant amount of your data in memory, so bring your wallet. (It may help to note that AWS offers instances with 244 GB of RAM — yes, that's one quarter of a terabyte — for a mere $2500 per month, letting you try before you buy.)

The extremely fast response times close the gap to existing Enterprise IT in two ways: First, by offering SQL-like interface and database-like response times and second, by providing the ODBC [3]-compatible connectors that traditional business intelligence (BI) tools expect.

Impala, a Java-based open source project from Cloudera, is the most promising. It reuses Hive's query language, although current technological limitations prevent it from supporting the full range of commands available in Hive. Druid, a Java-based open source project from Metamarkets, offers a clean, elegant API and will be quite compelling to folks who think like programmers and not like database analysts. If you're interested in a commercial solution, Hadapt and VoltDB (software) and Amazon's RedShift (cloud-hosted) all look viable.

3. Online Database Connectivity

Lastly, just as this chapter was being written Facebook open sourced their Presto project. It is too early to say whether it will be widely adopted, but Facebook doesn't do anything thoughtlessly or at a small scale. We'd include it in any evaluation.

Which to choose? If you want the simple answer, use Impala if you run your own clusters or RedShift if you prefer a cloud solution. But this technology only makes sense when you've gone beyond what traditional solutions support. You'll be spending hundreds of thousands of dollars here, so do a thorough investigation.

You'll hear the word "realtime" attached to both streaming and OLAP technologies; there are actually three things meant by that term. The first, let's call "immediate realtime" provided by the CEP solutions: If the consequent actions of a new piece of data have not occurred within 50 milliseconds or less, forget about it. Let's call what the streaming analytics solutions provide "prompt realtime;" there is a higher floor on the typical processing latency but you are able to handle all the analytical processing and consequent actions for each piece of data as it is received. Lastly, the OLAP data stores provide what we will call "interactive realtime;" data is both promptly manifested in the OLAP system's tables and the results of queries are returned and available within an analyst's attention span.

Database Crossloading

```
All the tools above focus on handling massive streams of data in constant flight.  Sometimes, what
```

Most large enterprises are already using a traditional ETL [4] tool such as Informatica and (TODO: put in name of the other one). If you want a stodgy, expensive Enterprise-grade solution, their sales people will enthusiastically endorse it for your needs, but if extreme scalability is essential, and their relative immaturity is not a deal breaker, use Sqoop, Kafka or Flume to centralize your data.

Core Platform: Data Stores

In the old days, there was such a thing as "a" database. These adorable, all-in-one devices not only stored your data, they allowed you to interrogate it and restructure it. They did those tasks so well we forgot they were different things, stopped asking questions about what was possible and stopped noticing the brutal treatment the database inflicted on our programming models.

As the size of data under management explodes beyond one machine, it becomes increasingly impossible to transparently support that abstraction. You can pay companies

4. Extract, Transform and Load, although by now it really means "the thing ETL vendors sell"

like Oracle or Netezza large sums of money to fight a rear-guard action against data locality on your behalf or you can abandon the Utopian conceit that one device can perfectly satisfy the joint technical constraints of storing, interrogating and restructuring data at arbitrary scale and velocity for every application in your shop.

As it turns out, there are a few coherent ways to variously relax those constraints and around each of those solution sets has grown a wave of next-generation data stores — referred to with the (TODO: change word) idiotic collective term "NoSQL" databases. The resulting explosion in the number of technological choices presents a baffling challenge to anyone deciding "which NoSQL database is the right one for me?" Unfortunately, the answer is far worse than that because the right question is "which NoSQL *databases* are the right choices for me?"

Big data applications at scale are best architected using a variety of data stores and analytics systems.

The good news is that, by focusing on narrower use cases and relaxing selected technical constraints, these new data stores can excel at their purpose far better than an all-purpose relational database would. Let's look at the respective data store archetypes that have emerged and their primary contenders.

Traditional Relational Databases

The reason the name "NoSQL" is so stupid is that it is not about rejecting traditional databases, it is about choosing the right database for the job. For the majority of jobs, that choice continues to be a relational database. Oracle, MS SQL Server, MySQL and PostgreSQL are not going away. The latter two have widespread open source support and PostgreSQL, in particular, has extremely strong geospatial functionality. As your data scales, fewer and fewer of their powerful JOIN capabilities survive but for direct retrieval, they will keep up with even the dedicated, lightweight key-value stores described below.

If you are already using one of these products, find out how well your old dog performs the new tricks before you visit the pound.

Billions of Records

At the extreme far end of the ecosystem are a set of data stores that give up the ability to be queried in all but the simplest ways in return for the ability to store and retrieve trillions of objects with exceptional durability, throughput and latency. The choices we like here are Cassandra, HBase or Accumulo, although Riak, Voldemort, Aerospike, Couchbase and Hypertable deserve consideration as well.

Cassandra is the pure-form expression of the "trillions of things" mission. It is operationally simple and exceptionally fast on write, making it very popular for time-series applications. HBase and Accumulo are architecturally similar in that they sit on top of

Hadoop's HDFS; this makes them operationally more complex than Cassandra but gives them an unparalleled ability to serve as source and destination of Map/Reduce jobs.

All three are widely popular open source, Java-based projects. Accumulo was initially developed by the U.S. National Security Administration (NSA) and was open sourced in 2011. HBase has been an open source Apache project since its inception in 2006 and both are nearly identical in architecture and functionality. As you would expect, Accumulo has unrivaled security support while HBase's longer visibility gives it a wider installed base.

We can try to make the choice among the three sound simple: If security is an overriding need, choose Accumulo. If simplicity is an overriding need, choose Cassandra. For overall best compatibility with Hadoop, use HBase.

However, if your use case justifies a data store in this class, it will also require investing hundreds of thousands of dollars in infrastructure and operations. Do a thorough bake-off among these three and perhaps some of the others listed above.

What you give up in exchange is all but the most primitive form of locality. The only fundamental retrieval operation is to look records or ranges of records by primary key. There is Sugar for secondary indexing and tricks that help restore some of the power you lost but effectively, that's it. No JOINS, no GROUPS, no SQL.

- H-base, Accumulo and Cassandra
- Aerospike, Voldemort and Riak, Hypertable

Scalable Application-Oriented Data Stores

If you are using Hadoop and Storm+Trident, you do not need your database to have sophisticated reporting or analytic capability. For the significant number of use cases with merely hundreds of millions (but not tens of billions) of records, there are two data stores that give up the ability to do complex JOINS and GROUPS and instead focus on delighting the application programmer.

MongoDB starts with a wonderful hack: It uses the operating system's "memory-mapped file" (mmap) features to give the internal abstraction of an infinitely-large data space. The operating system's finely-tuned virtual memory mechanisms handle all details of persistence, retrieval and caching. That internal simplicity and elegant programmer-friendly API make MongoDB a joy to code against.

Its key tradeoff comes from its key advantage: The internal mmap abstraction delegates all matters of in-machine locality to the operating system. It also relinquishes any fine control over in-machine locality. As MongoDB scales to many machines, its locality abstraction starts to leak. Some features that so delighted you at the start of the project prove to violate the laws of physics as the project scales into production. Any claims

that MongoDB "doesn't scale," though, are overblown; it scales quite capably into the billion-record regime but doing so requires expert guidance.

Probably the best thing to do is think about it this way: The open source version of MongoDB is free to use on single machines by amateurs and professionals, one and all; anyone considering using it on multiple machines should only do so with commercial support from the start.

The increasingly-popular ElasticSearch data store is our first choice for hitting the sweet spot of programmer delight and scalability. The heart of ElasticSearch is Lucene, which encapsulates the exceptionally difficult task of indexing records and text in a streamlined gem of functionality, hardened by a decade of wide open source adoption. [5]

ElasticSearch embeds Lucene into a first-class distributed data framework and offers a powerful programmer-friendly API that rivals MongoDB's. Since Lucene is at its core, it would be easy to mistake ElasticSearch for a text search engine like Solr; it is one of those and, to our minds, the best one, but it is also a first-class database.

Scalable Free-Text Search Engines: Solr, ElasticSearch and More

The need to perform free-text search across millions and billions of documents is not new and the Lucene-based Solr search engine is the dominant traditional solution with wide Enterprise support. It is, however, long in tooth and difficult to scale.

ElasticSearch, described above as an application-oriented database, is also our recommended choice for bringing Lucene's strengths to Hadoop's scale.

Two recent announcements — the "Apache Blur" (TODO LINK) project and the related "Cloudera Search" (TODO LINK) product — also deserve consideration.

Lightweight Data Structures

"ZooKeeper" (TODO LINK) is basically "distributed correctness in a box." Transactionally updating data within a distributed system is a fiendishly difficult task, enough that implementing it on your own should be a fireable offense. ZooKeeper and its ubiquitously available client libraries let you synchronize updates and state among arbitrarily large numbers of concurrent processes. It sits at the core of HBase, Storm, Hadoop's newer high-availability Namenode and dozens of other high-scale distributed applications. It is a bit thorny to use; projects like etcd (TODO link) and Doozer (TODO link) fill the same need but provide friendlier APIs. We feel this is no place for liberalism, however — ZooKeeper is the default choice.

5. Lucene was started, incidentally, by Doug Cutting several years before he started the Hadoop project.

If you turn the knob for programmer delight all the way to the right, one request that would fall out would be, "Hey - can you take the same data structures I use while I'm coding but make it so I can have as many of them as I have RAM and shared across as many machines and processes as I like?" The Redis data store is effectively that. Its API gives you the fundamental data structures you know and love — hashmap, stack, buffer, set, etc — and exposes exactly the set of operations that can be performance and distributedly correct. It is best used when the amount of data does not much exceed the amount of RAM you are willing to provide and should only be used when its data structures are a direct match to your application. Given those constraints, it is simple, light and a joy to use.

Sometimes, the only data structure you need is "given name, get thing." Memcached is an exceptionally fast in-memory key value store that serves as the caching layer for many of the Internet's largest websites. It has been around for a long time and will not go away any time soon.

If you are already using MySQL or PostgreSQL, and therefore only have to scale by cost of RAM not cost of license, you will find that they are perfectly defensible key value stores in their own right. Just ignore 90-percent of their user manuals and find out when the need for better latency or lower cost of compute forces you to change.

"Kyoto Tycoon" (TODO LINK) is an open source C++-based distributed key value store with the venerable DBM database engine at its core. It is exceptionally fast and, in our experience, is the simplest way to efficiently serve a mostly-cold data set. It will quite happily serve hundreds of gigabytes or terabytes of data out of not much more RAM than you require for efficient caching.

Graph Databases

Graph-based databases have been around for some time but have yet to see general adoption outside of, as you might guess, the intelligence and social networking communities (NASH). We suspect that, as the price of RAM continues to drop and the number of data scientists continues to rise, sophisticated analysis of network graphs will become increasingly important and, we hear, increasing adoption of graph data stores.

The two open source projects we hear the most about are the longstanding Neo 4J project and the newer, fresher TitanDB.

Your authors do not have direct experience here, but the adoption rate of TitanDB is impressive and we believe that is where the market is going.

Programming Languages, Tools and Frameworks

SQL-like High-Level Languages: Hive and Pig

Every data scientist toolkit should include either Hive or Pig, two functionally equivalent languages that transform SQL-like statements into efficient Map/Reduce jobs. Both of them are widely-adopted open source projects, written in Java and easily extensible using Java-based User-Defined Functions (UDFs).

Hive is more SQL-like, which will appeal to those with strong expertise in SQL. Pig's language is sparer, cleaner and more orthogonal, which will appeal to people with a strong distaste for SQL Hive's model manages and organizes your data for you, which is good and bad. If you are coming from a data warehouse background, this will provide a very familiar model. On the other hand, Hive *insists* on managing and organizing your data, making it play poorly with the many other tools that experimental data science requires. (The H Catalog Project aims to fix this and is maturing nicely).

In Pig, every language primitive maps to a fundamental dataflow primitive; this harmony with the Map/Reduce paradigm makes it easier to write and reason about efficient dataflows. Hive aims to complete the set of traditional database operations; this is convenient and lowers the learning curve but can make the resulting dataflow more opaque.

Hive is seeing slightly wider adoption but both have extremely solid user bases and bright prospects for the future.

Which to choose? If you are coming from a data warehousing background or think best in SQL, you will probably prefer Hive. If you come from a programming background and have always wished SQL just made more sense, you will probably prefer Pig. We have chosen to write all the examples for this book in Pig — its greater harmony with Map/Reduce makes it a better tool for teaching people how to think in scale. Let us pause and suggestively point to this book's creative commons license, thus perhaps encouraging an eager reader to translate the book into Hive (or Python, Chinese or Cascading).

High-Level Scripting Languages: Wukong (Ruby), mrjob (Python) and Others

Many people prefer to work strictly within Pig or Hive, writing Java UDFs for everything that cannot be done as a high-level statement. It is a defensible choice and a better mistake than the other extreme of writing everything in the native Java API. Our experience, however, has been, say 60-percent of our thoughts are best expressed in Pig, perhaps 10-percent of them require a low-level UDF but that the remainder are far better expressed in a high-level language like Ruby or Python.

Most Hadoop jobs are IO-bound, not CPU-bound, so performance concerns are much less likely to intervene. (Besides, robots are cheap but people are important. If you want

your program to run faster, use more machines, not more code). These languages have an incredibly rich open source toolkit ecosystem and cross-platform glue. Most importantly, their code is simpler, shorter and easier to read; far more of data science than you expect is brass-knuckle street fighting, necessary acts of violence to make your data look like it should. These are messy, annoying problems, not deep problems and, in our experience, the only way to handle them maintainably is in a high-level scripting language.

You probably come in with a favorite scripting language in mind, and so by all means, use that one. The same Hadoop streaming interface powering the ones we will describe below is almost certainly available in your language of choice. If you do not, we will single out Ruby, Python and Scala as the most plausible choices, roll our eyes at the language warhawks sharpening their knives and briefly describe the advantages of each.

Ruby is elegant, flexible and maintainable. Among programming languages suitable for serious use, Ruby code is naturally the most readable and so it is our choice for this book. We use it daily at work and believe its clarity makes the thought we are trying to convey most easily portable into the reader's language of choice.

Python is elegant, clean and spare. It boasts two toolkits appealing enough to serve as the sole basis of choice for some people. The Natural Language toolkit (NLTK) is not far from the field of computational linguistics set to code. SciPy is widely used throughout scientific computing and has a full range of fast, robust matrix and numerical logarithms.

Lastly, Scala, a relative newcomer, is essentially "Java but readable." It's syntax feels very natural to native Java programmers and executives directly into the JBM, giving it strong performance and first-class access to native Java frameworks, which means, of course, native access to the code under Hadoop, Storm, Kafka, etc.

If runtime efficiency and a clean match to Java are paramount, you will prefer Scala. If your primary use case is text processing or hardcore numerical analysis, Python's superior toolkits make it the best choice. Otherwise, it is a matter of philosophy. Against Perl's mad credo of "there is more than one way to do it," Python says "there is exactly one right way to do it," while Ruby says "there are a few good ways to do it, be clear and use good taste." One of those alternatives gets your world view; choose accordingly.

Statistical Languages: R, Julia, Pandas and more

For many applications, Hadoop and friends are most useful for turning big data into medium data, cutting it down enough in size to apply traditional statistical analysis tools. SPSS, SaSS, Matlab and Mathematica are long-running commercial examples of these, whose sales brochures will explain their merits better than we can.

R is the leading open source alternative. You can consider it the "PHP of data analysis." It is extremely inviting, has a library for everything, much of the internet runs on it and

considered as a language, is inelegant, often frustrating and Vulcanized. Do not take that last part too seriously; whatever you are looking to do that can be done on a single machine, R can do. There are Hadoop integrations, like RHIPE, but we do not take them very seriously. R is best used on single machines or trivially parallelized using, say, Hadoop.

Julia is an upstart language designed by programmers, not statisticians. It openly intends to replace R by offering cleaner syntax, significantly faster execution and better distributed awareness. If its library support begins to rival R's, it is likely to take over but that probably has not happened yet.

Lastly, Pandas, Anaconda and other Python-based solutions give you all the linguistic elegance of Python, a compelling interactive shell and the extensive statistical and machine-learning capabilities that NumPy and scikit provide. If Python is your thing, you should likely start here.

Mid-level Languages

You cannot do everything a high-level language, of course. Sometimes, you need closer access to the Hadoop API or to one of the many powerful, extremely efficient domain-specific frameworks provided within the Java ecosystem. Our preferred approach is to write Pig or Hive UDFs; you can learn more in Chapter (TODO: REF).

Many people prefer, however, prefer to live exclusively at this middle level. Cascading strikes a wonderful balance here. It combines an elegant DSL for describing your Hadoop job as a dataflow and a clean UDF framework for record-level manipulations. Much of Trident's API was inspired by Cascading; it is our hope that Cascading eventually supports Trident or Storm as a back end. Cascading is quite popular, and besides its native Java experience, offers first-class access from Scala (via the Scalding project) or Clojure (via the Cascalog project).

Lastly, we will mention Crunch, an open source Java-based project from Cloudera. It is modeled after a popular internal tool at Google; it sits much closer to the Map/Reduce paradigm, which is either compelling to you or not.

Frameworks

Finally, for the programmers, there are many open source frameworks to address various domain-specific problems you may encounter as a data scientist. Going into any depth here is outside the scope of this book but we will at least supply you with a list of pointers.

Elephant Bird, Datafu and Akela offer extremely useful additional Pig and Hive UDFs. While you are unlikely to need all of them, we consider no Pig or Hive installation complete without them. For more domain-specific purposes, anyone in need of a machine-learning algorithm should look first at Mahout, Kiji, Weka scikit-learn or those

available in a statistical language, such as R, Julia or NumPy. Apache Giraph and Gremlin are both useful for graph analysis. The HIPI *http://hipi.cs.virginia.edu/* toolkit enables image processing on Hadoop with library support and a bundle format to address the dreaded "many small files" problem (TODO ref).

(NOTE TO TECH REVIEWERS: What else deserves inclusion?)

Lastly, because we do not know where else to put them, there are several Hadoop "environments," some combination of IDE frameworks and conveniences that aim to make Hadoop friendlier to the Enterprise programmer. If you are one of those, they are worth a look.

CHAPTER 9

Intro to Storm+Trident

Enter the Dragon: C&E Corp Gains a New Partner

Dragons are fast and sleek, and never have to sleep. They exist in some ways out of time — a dragon can perform a thousand actions in the blink of an eye, and yet a thousand years is to them a passing moment

Intro: Storm+Trident Fundamentals

At this point, you have good familiarity with Hadoop's batch processing power and the powerful inquiries it unlocks above and as a counterpart to traditional database approach. Stream analytics is a third mode of data analysis and, it is becoming clear, one that is just as essential and transformative as massive scale batch processing has been.

Storm is an open-source framework developed at Twitter that provides scalable stream processing. Trident draws on Storm's powerful transport mechanism to provide *exactly once* processing of records in *windowed batches es* for aggregating and persisting to an external data store.

The central challenge in building a system that can perform fallible operations on billions of records reliably is how to do so without yourself producing so much bookkeeping that it becomes its own scalable Stream processing challenge. Storm handles all details of reliable transport and efficient routing for you, leaving you with only the business process at hand. (The remarkably elegant way Storm handles that bookkeeping challenge is one of its principle breakthroughs; you'll learn about it in the later chapter on Storm Internals.)

This takes Storm past the mere processing of records to Stream Analytics — with some limitations and some advantages, you have the same ability to specify locality and write

arbitrarily powerful general-purpose code to handle every record. A lot of Storm+Trident's adoption is in application to real-time systems. [1]

But, just as importantly, the framework exhibits radical *tolerance* of latency. It's perfectly reasonable to, for every record, perform reads of a legacy data store, call an internet API and the like, even if those might have hundreds or thousands of milliseconds worst-case latency. That range of timescales is simply impractical within a batch processing run or database query. In the later chapter on the Lambda Architecture, you'll learn how to use stream and batch analytics together for latencies that span from milliseconds to years.

As an example, one of the largest hard drive manufacturers in the world ingests sensor data from its manufacturing line, test data from quality assurance processes, reports from customer support and post mortem analysis of returned devices. They have been able to mine the accumulated millisecond scale sensor data for patterns that predict flaws months and years later. Hadoop produces the "slow, deep" results, uncovering the patterns that predict failure. Storm+Trident produces the fast, relevant results: operational alerts when those anomalies are observed.

Things you should take away from this chapter:

Understand the type of problems you solve using stream processing and apply it to real examples using the best-in-class stream analytics frameworks. Acquire the practicalities of authoring, launching and validating a Storm+Trident flow. Understand Trident's operators and how to use them: Each apply `CombinerAggregator`s, `ReducerAggregator`s and `AccumulatingAggregator`s (generic aggregator?) Persist records or aggregations directly to a backing database or to Kafka for item-potent downstream storage. (probably not going to discuss how to do a streaming join, using either DRPC or a hashmap join)

This chapter will only speak of Storm+Trident, the high level and from the outside. We won't spend any time on how it's making this all work until (to do ref the chapter on Storm+Trident internals)

Your First Topology

Topologies in Storm are analogous to jobs in Hadoop - they define the path data takes through your system and the operations applied along the way. Topologies are compiled locally and then submitted to a Storm cluster where they run indefinitely until stopped.

1. for reasons you'll learn in the Storm internals chapter, it's not suitable for ultra-low latency (below, say, 5s of milliseconds), Wall Street-type applications, but if latencies above that are real-time enough for you, Storm +Trident shines.

You define your topology and Storm handles all the hard parts — fault tolerance, retrying, and distributing your code across the cluster among other things.

For your first Storm+Trident topology, we're going to create a topology to handle a typical streaming use case: accept a high rate event stream, process the events to power a realtime dashboard, and then store the records for later analysis. Specifically, we're going to analyze the Github public timeline and monitor the number of commits per language.

A basic logical diagram of the topology looks like this:

i. 89-intro-to-storm-topo.png …

Each node in the diagram above represents a specific operation on the data flow. Initially JSON records are retrieved from Github and injected into the topology by the Github Spout, where they are transformed by a series of operations and eventually persisted to an external data store. Trident spouts are sources of streams of data — common use cases include pulling from a Kafka queue, Redis queue, or some other external data source. Streams are in turn made up of tuples which are just lists of values with names attached to each field.

The meat of the Java code that constructs this topology is as follows:

IBlobStore bs = new FileBlobStore("~/dev/github-data/test-data"); OpaqueTransactionalBlobSpout spout = new OpaqueTransactionalBlobSpout(bs, StartPolicy.EARLIEST, null); TridentTopology topology = new TridentTopology(); topology.newStream("github-activities", spout) .each(new Fields("line"), new JsonParse(), new Fields("parsed-json")) .each(new Fields("parsed-json"), new ExtractLanguageCommits(), new Fields("language", "commits")) .groupBy(new Fields("language")) .persistentAggregate(new VisibleMemoryMapState.Factory(), new Count(), new Fields("commit-sum"));

The first two lines are responsible for constructing the spout. Instead of pulling directly from Github, we'll be using a directory of downloaded json files so as not to a) unnecessarily burden Github and b) unnecessarily complicate the code. You don't need to worry about the specifics, but the OpaqueTransactionalBlobSpout reads each json file and feeds it line by line into the topology.

After creating the spout we construct the topology by calling `new TridentTopology()`. We then create the topology's first (and only) stream by calling `newStream` and passing in the spout we instantiated earlier along with a name, "github-activities". We can then chain a series of method calls off newStream() to tack on our logic after the spout.

The `each` method call, appropriately, applies an operation to each tuple in the stream. The important parameter in the `each` calls is the second one, which is a class that defines

the operation to be applied to each tuple. The first `each` uses the JsonParse class which parses the JSON coming off the spout and turns it into an object representation that we can work with more easily. Our second `each` uses `ExtractLanguageCommits.class` to pull the statistics we're interested in from the parsed JSON objects, namely the language and number of commits. `ExtractLanguageCommits.class` is fairly straightforward, and it is instructive to digest it a bit:

public static class ExtractLanguageCommits extends BaseFunction { private static final Logger LOG = LoggerFactory.getLogger(ExtractLanguageCommits.class); public void execute(TridentTuple tuple, TridentCollector collector){ JsonNode node = (JsonNode) tuple.getValue(0); if(!node.get("type").toString().equals("\"PushEvent\"")) return; List values = new ArrayList(2); //grab the language and the action values.add(node.get("repository").get("language").asText()); values.add(node.get("payload").get("size").asLong()); collector.emit(values); return; } }

There is only one method, `execute`, that excepts a `tuple` and a `collector`. The tuples coming into `ExtractLanguageCommits` have only one field, `parsed-json`, which contains a JsonNode, so the first thing we do is cast it. We then use the `get` method to access the various pieces of information we need.

At the time of writing, the full schema for Github's public stream is available here, but here are the important bits:

{ "type": "PushEvent", // can be one of .. finish JSON… }

… finish this section …

At this point the tuples in our stream might look something like this:

("C", 2), ("JavaScript", 5), ("CoffeeScript", 1), ("PHP", 1), ("JavaScript", 1), ("PHP", 2)

We then group on the language and sum the counts, giving our final tuple stream which could look like this:

("C", 2), ("JavaScript", 6), ("CoffeeScript", 1), ("PHP", 3)

The group by is exactly what you think it is - it ensures that every tuple with the same language is grouped together and passed through the same thread of execution, allowing you to perform the sum operation across all tuples in each group. After summing the commits, the final counts are stored in a database. Feel free to go ahead and try it out yourself.

So What?

You might be thinking to yourself "So what, I can do that in Hadoop in 3 lines…" and you'd be right — almost. It's important to internalize the difference in focus between Hadoop and Storm+Trident — when using Hadoop you must have all your data sitting in front of you before you can start, and Hadoop won't provide any results until pro-

cessing all of the data is complete. The Storm+Trident topology you just built allows you to update your results as you receive your stream of data in real time, which opens up a whole set of applications you could only dream about with Hadoop.

CHAPTER 10
Statistics

Skeleton: Statistics

Data is worthless. Actually, it's worse than worthless. It costs you money to gather, store, manage, replicate and analyze. What you really want is insight — a relevant summary of the essential patterns in that data — produced using relationships to analyze data in context.

Statistical summaries are the purest form of this activity, and will be used repeatedly in the book to come, so now that you see how Hadoop is used it's a good place to focus.

Some statistical measures let you summarize the whole from summaries of the parts: I can count all the votes in the state by summing the votes from each county, and the votes in each county by summing the votes at each polling station. Those types of aggregations — average/standard deviation, correlation, and so forth — are naturally scalable, but just having billions of objects introduces some practical problems you need to avoid. We'll also use them to introduce Pig, a high-level language for SQL-like queries on large datasets.

Other statistical summaries require assembling context that grows with the size of the whole dataset. The amount of intermediate data required to count distinct objects, extract an accurate histogram, or find the median and other quantiles can become costly and cumbersome. That's especially unfortunate because so much data at large scale has a long-tail, not normal (Gaussian) distribution — the median is far more robust indicator of the "typical" value than the average. (If Bill Gates walks into a bar, everyone in there is a billionaire on average.)

Summary Statistics

TODO: content to come

Overflow, Underflow and other Dangers

TODO: content to come

Quantiles and Histograms

TODO: content to come

Algebraic vs Holistic Aggregations

TODO: content to come

"Sketching" Algorithms

TODO: content to come

CHAPTER 11

Event Streams

Webserver Log Parsing

We'll represent loglines with the following model definition:

```
class Logline

  include Gorillib::Model

  field :ip_address,    String
  field :requested_at,  Time
  field :http_method,   String,  doc: "GET, POST, etc"
  field :uri_str,       String,  doc: "Combined path and query string of request"
  field :protocol,      String,  doc: "eg 'HTTP/1.1'"
  #
  field :response_code, Integer, doc: "HTTP status code (j.mp/httpcodes)"
  field :bytesize,      Integer, doc: "Bytes in response body", blankish: ['', nil, '-']
  field :referer,       String,  doc: "URL of linked-from page. Note speling."
  field :user_agent,    String,  doc: "Version info of application making the request"

  def visitor_id ; ip_address ; end

end
```

Since most of our questions are about what visitors do, we'll mainly use `visitor_id` (to identify common requests for a visitor), `uri_str` (what they requested), `requested_at` (when they requested it) and `referer` (the page they clicked on). Don't worry if you're not deeply familiar with the rest of the fields in our model — they'll become clear in context.

Two notes, though. In these explorations, we're going to use the `ip_address` field for the `visitor_id`. That's good enough if you don't mind artifacts, like every visitor from the same coffee shop being treated identically. For serious use, though, many web applications assign an identifying "cookie" to each visitor and add it as a custom logline

field. Following good practice, we've built this model with a `visitor_id` method that decouples the *semantics* ("visitor") from the *data* ("the IP address they happen to have visited from"). Also please note that, though the dictionary blesses the term *referrer*, the early authors of the web used the spelling *referer* and we're now stuck with it in this context. ///Here is a great example of supporting with real-world analogies, above where you wrote, "like every visitor from the same coffee shop being treated identically…" Yes! That is the kind of simple, sophisticated tying-together type of connective tissue needed throughout, to greater and lesser degrees. Amy////

Simple Log Parsing

////Help the reader in by writing something quick, short, like, "The core nugget that you should know about simple log parsing is…" Amy////

Webserver logs typically follow some variant of the "Apache Common Log" format — a series of lines describing each web request:

```
154.5.248.92 - - [30/Apr/2003:13:17:04 -0700] "GET /random/video/Star_Wars_Kid.wmv HTTP/1.0" 206 1
```

Our first task is to leave that arcane format behind and extract healthy structured models. Since every line stands alone, the parse script is simple as can be: a transform-only script that passes each line to the `Logline.parse` method and emits the model object it returns.

```
class ApacheLogParser < Wukong::Streamer::Base
  include Wukong::Streamer::EncodingCleaner

  def process(rawline)
    logline = Logline.parse(rawline)
    yield [logline.to_tsv]
  end
end

Wukong.run( ApacheLogParser )
```

Star Wars Kid serverlogs

For sample data, we'll use the webserver logs released by blogger Andy Baio. In 2003, he posted the famous "Star Wars Kid" video, which for several years ranked as the biggest viral video of all time. (It augments a teenager's awkward Jedi-style fighting moves with the special effects of a real lightsaber.) Here's his description:

> I've decided to release the first six months of server logs from the meme's spread into the public domain — with dates, times, IP addresses, user agents, and referer information. … On April 29 at 4:49pm, I posted the video, renamed to "Star_Wars_Kid.wmv" — inadvertently giving the meme its permanent name. (Yes, I coined the term "Star Wars Kid." It's strange to think it would've been "Star Wars Guy" if I was any lazier.) From there, for the first week, it spread quickly through news site, blogs and message boards, mostly oriented around technology, gaming, and movies. …

> This file is a subset of the Apache server logs from April 10 to November 26, 2003. It contains every request for my homepage, the original video, the remix video, the mirror redirector script, the donations spreadsheet, and the seven blog entries I made related to Star Wars Kid. I included a couple weeks of activity before I posted the videos so you can determine the baseline traffic I normally received to my homepage. The data is public domain. If you use it for anything, please drop me a note!

The details of parsing are mostly straightforward — we use a regular expression to pick apart the fields in each line. That regular expression, however, is another story:

```
class Logline

  # Extract structured fields using the `raw_regexp` regular expression
  def self.parse(line)
    mm = raw_regexp.match(line.chomp) or return BadRecord.new('no match', line)
    new(mm.captures_hash)
  end
  ### @export


  class_attribute :raw_regexp

  #
  # Regular expression to parse an apache log line.
  #
  # 83.240.154.3 - - [07/Jun/2008:20:37:11 +0000] "GET /faq?onepage=true HTTP/1.1" 200 569 "http:/
  #
  self.raw_regexp = %r{\A
              (?<ip_address>   [\w\.]+)          # ip_address     83.240.154.3
            \ (?<identd>     \S+)                # identd       -  (rarely used)
            \ (?<authuser>     \S+)              # authuser       -  (rarely used)
                                                 #
           \ \[(?<requested_at>                  #
                              \d+/\w+/\d+        # date part      [07/Jun/2008
                             :\d+:\d+:\d+        # time part      :20:37:11
                            \ [\+\-]\S*)\]       # timezone       +0000]
                                                 #
        \ \"(?:(?<http_method>  [A-Z]+)          # http_method    "GET
             \ (?<uri_str>      \S+)             # uri_str        faq?onepage=true
             \ (?<protocol>     HTTP/[\d\.]+)|-)\" # protocol       HTTP/1.1"
                                                 #
             \ (?<response_code>\d+)             # response_code  200
             \ (?<bytesize>     \d+|-)           # bytesize       569
           \ \"(?<referer>      [^\"]*)\"        # referer        "http://infochimps.org/search?
           \ \"(?<user_agent>   [^\"]*)\"        # user_agent     "Mozilla/5.0 (Windows; U; Winc
          \z}x

end
```

It may look terrifying, but taken piece-by-piece it's not actually that bad. Regexp-fu is an essential skill for data science in practice — you're well advised to walk through it. Let's do so.

- The meat of each line describe the contents to match — `\S+` for "a sequence of non-whitespace", `\d+` for "a sequence of digits", and so forth. If you're not already familiar with regular expressions at that level, consult the excellent tutorial at regular-expressions.info.
- This is an *extended-form* regular expression, as requested by the `x` at the end of it. An extended-form regexp ignores whitespace and treats `#` as a comment delimiter — constructing a regexp this complicated would be madness otherwise. Be careful to backslash-escape spaces and hash marks.
- The `\A` and `\z` anchor the regexp to the absolute start and end of the string respectively.
- Fields are selected using *named capture group* syntax: `(?<ip_address>\S+)`. You can retrieve its contents using `match[:ip_address]`, or get all of them at once using `captures_hash` as we do in the `parse` method.
- Build your regular expressions to be *good brittle*. If you only expect HTTP request methods to be uppercase strings, make your program reject records that are otherwise. When you're processing billions of events, those one-in-a-million deviants start occurring thousands of times.

That regular expression does almost all the heavy lifting, but isn't sufficient to properly extract the `requested_at` time. Wukong models provide a "security gate" for each field in the form of the `receive_(field name)` method. The setter method (`requested_at=`) applies a new value directly, but the `receive_requested_at` method is expected to appropriately validate and transform the given value. The default method performs simple *do the right thing*-level type conversion, sufficient to (for example) faithfully load an object from a plain JSON hash. But for complicated cases you're invited to override it as we do here.

```
class Logline

  # Map of abbreviated months to date number.
  MONTHS = { 'Jan' => 1, 'Feb' => 2, 'Mar' => 3, 'Apr' => 4, 'May' => 5, 'Jun' => 6, 'Jul' => 7, '

  def receive_requested_at(val)
    # Time.parse doesn't like the quirky apache date format, so handle those directly
    mm = %r{(\d+)/(\w+)/(\d+):(\d+):(\d+):(\d+)\s([\+\-]\d\d)(\d\d)}.match(val) rescue nil
    if mm
      day, mo, yr, hour, min, sec, tz1, tz2 = mm.captures
      val = Time.new(
        yr.to_i,   MONTHS[mo], day.to_i,
        hour.to_i, min.to_i,   sec.to_i, "#{tz1}:#{tz2}")
```

```
      end
      super(val)
    end

  end
```

There's a general lesson here for data-parsing scripts. Don't try to be a hero and get everything done in one giant method. The giant regexp just coarsely separates the values; any further special handling happens in isolated methods.

Test the script in local mode:

```
~/code/wukong$ head -n 5 data/serverlogs/star_wars_kid/star_wars_kid-raw-sample.log | examples/ser
170.20.11.59    2003-04-30T20:17:02Z    GET     /archive/2003/04/29/star_war.shtml      HTTP/1.0
154.5.248.92    2003-04-30T20:17:04Z    GET     /random/video/Star_Wars_Kid.wmv         HTTP/1.0
199.91.33.254   2003-04-30T20:17:09Z    GET     /random/video/Star_Wars_Kid.wmv         HTTP/1.0
131.229.113.229 2003-04-30T20:17:09Z    GET     /random/video/Star_Wars_Kid.wmv         HTTP/1.1
64.173.152.130  2003-04-30T20:17:18Z    GET     /archive/2003/02/19/coachell.shtml      HTTP/1.1
```

Then run it on the full dataset to produce the starting point for the rest of our work:

```
TODO
```

Geo-IP Matching

You can learn a lot about your site's audience in aggregate by mapping IP addresses to geolocation. Not just in itself, but joined against other datasets, like census data, store locations, weather and time. [1]

Maxmind makes their GeoLite IP-to-geo database available under an open license (CC-BY-SA)[2]. Out of the box, its columns are `beg_ip`, `end_ip`, `location_id`, where the first two columns show the low and high ends (inclusive) of a range that maps to that location. Every address lies in at most one range; locations may have multiple ranges.

This arrangement caters to range queries in a relational database, but isn't suitable for our needs. A single IP-geo block can span thousands of addresses.

To get the right locality, take each range and break it at some block level. Instead of having `1.2.3.4` to `1.2.5.6` on one line, let's use the first three quads (first 24 bits) and emit rows for `1.2.3.4` to `1.2.3.255`, `1.2.4.0` to `1.2.4.255`, and `1.2.5.0` to `1.2.5.6`. This lets us use the first segment as the partition key, and the full ip address as the sort key.

1. These databases only impute a coarse-grained estimate of each visitor's location — they hold no direct information about the persom. Please consult your priest/rabbi/spirit guide/grandmom or other appropriate moral compass before diving too deep into the world of unmasking your site's guests.
2. For serious use, there are professional-grade datasets from Maxmind, Quova, Digital Element among others.

```
    lines           bytes  description                 file
15_288_766   1_094_541_688  24-bit partition key        maxmind-geolite_city-20121002.tsv
 2_288_690     183_223_435  16-bit partition key        maxmind-geolite_city-20121002-16.tsv
 2_256_627      75_729_432  original (not denormalized) GeoLiteCity-Blocks.csv
```

Range Queries

////Gently introduce the concept. "So, here's what range queries are all about, in a nutshell…" Amy////

This is a generally-applicable approach for doing range queries.

- Choose a regular interval, fine enough to avoid skew but coarse enough to avoid ballooning the dataset size.
- Whereever a range crosses an interval boundary, split it into multiple records, each filling or lying within a single interval.
- Emit a compound key of `[interval, join_handle, beg, end]`, where
  - `interval` is
  - `join_handle` identifies the originating table, so that records are grouped for a join (this is what ensures If the interval is transparently a prefix of the index (as it is here), you can instead just ship the remainder: `[interval, join_handle, beg_suffix, end_suffix]`.
- Use the

In the geodata section, the "quadtile" scheme is (if you bend your brain right) something of an extension on this idea — instead of splitting ranges on regular intervals, we'll split regions on a regular grid scheme.

Using Hadoop for website stress testing ("Benign DDos")

Hadoop is engineered to consume the full capacity of every available resource up to the currently-limiting one. So in general, you should never issue requests against external services from a Hadoop job — one-by-one queries against a database; crawling web pages; requests to an external API. The resulting load spike will effectively be attempting what web security folks call a "DDoS", or distributed denial of service attack.

Unless of course you are trying to test a service for resilience against an adversarial DDoS — in which case that assault is a feature, not a bug!

elephant_stampede.

```
require 'faraday'

processor :elephant_stampede do
```

```
  def process(logline)
    beg_at = Time.now.to_f
    resp = Faraday.get url_to_fetch(logline)
    yield summarize(resp, beg_at)
  end

  def summarize(resp, beg_at)
    duration = Time.now.to_f - beg_at
    bytesize = resp.body.bytesize
    { duration: duration, bytesize: bytesize }
  end

  def url_to_fetch(logline)
    logline.url
  end
end

flow(:mapper){ input > parse_loglines > elephant_stampede }
```

You must use Wukong's eventmachine bindings to make more than one simultaneous request per mapper.

Refs

- Database of Robot User Agent strings
- Improving Web Search Results Using Affinity Graph

CHAPTER 12

Geographic Data Processing

Geographic Data Model

Geographic data shows up in the form of

- Points — a pair of coordinates. When given as an ordered pair (a "Position"), always use `[longitude,latitude]` in that order, matching the familiar `X,Y` order for mathematical points. When it's a point with other metadata, it's a Place [1], and the coordinates are named fields.
- Paths — an array of points `[[longitude,latitude],[longitude,latitude],...]`
- Region — an array of paths, understood to connect and bound a region of space. `[ [[longitude,latitude],[longitude,latitude],...], [[longitude,latitude],[longitude,latitude],...]]`. Your array will be of length one unless there are holes or multiple segments
- "Bounding Box" (or `bbox`) — a rectangular bounding region, `[-5.0, 30.0, 5.0, 40.0]`

Features of Features

1. in other works you'll see the term Point of Interest ("POI") for a place.

The term "feature" is somewhat muddied — to a geographer, "feature" indicates a *thing* being described (places, regions, paths are all geographic features). In the machine learning literature, "feature" describes a potentially-significant *attribute* of a data element (manufacturer, top speed and weight are features of a car). Since we're here as data scientists dabbling in geography, we'll reserve the term "feature" for only its machine learning sense.

Voronoi

Spatial data is fundamentally important ///Go, go…! Talk about what this is, put it in context. And then, weave in some conceptual talk about "locality," when you're done. Amy////

*

- So far we've

Spatial data ////", which identifies the geographic location of features and boundaries on Earth," - here I'm suggesting you define this kind of terms (in-line) when it comes up. Amy//// is very easy to acquire: from smartphones and other GPS devices, from government and public sources, and from a rich ecosystem of commercial suppliers. It's easy to bring our physical and cultural intuition to bear on geospatial problems ////"For example… Amy////

There are several big ideas introduced here.

First of course are the actual mechanics of working with spatial data, and projecting the Earth onto a coordinate plane.

The statistics and timeseries chapters dealt with their dimensions either singly or interacting weakly,

It's ////What is…? Clarify. Amy/// a good jumping-off point for machine learning. Take a tour through some of the sites that curate the best in data visualization, ////Consider defining in-line, like with spacial data above. Amy//// and you'll see a strong overrepresentation of geographic explorations. With most datasets, you need to figure out the salient features, eliminate confounding factors, and of course do all the work of transforming them to be joinable [2]. ////May want to suggest a list of 5 URLs to readers here. Amy////Geo Data comes out of the

Taking a step back, the fundamental idea this chapter introduces is a direct way to extend locality to two dimensions. It so happens we did so in the context of geospatial data, and required a brief prelude about how to map our nonlinear feature space to the plane.

2. we dive deeper in the chapter on Chapter 18 basics later on

Browse any of the open data catalogs (REF) or data visualization blogs, and you'll see that geographic datasets and visualizations are by far the most frequent. Partly this is because there are these two big obvious feature components, highly explanatory and direct to understand. But you can apply these tools any time you have a small number of dominant features and a sensible distance measure mapping them to a flat space.

TODO:

Will be reorganizing below in this order:

- do a "nearness" query example,
- reveal that it is such a thing known as the spatial join, and broaden your mind as to how you think about locality.
- cover the geographic data model, GeoJSON etc.
- Spatial concept of Quadtiles — none of the mechanics of the projection yet
- Something with Points and regions, using quadtiles
- Actual mechanics of Quadtile Projection — from lng/lat to quadkey
- mutiscale quadkey assignment
- (k-means will move to ML chapter)
- complex nearness — voronoi cells and weather data

also TODO: untangle the following two paragraphs, and figure out whether to put them at beginning or end (probably as sidebar, at beginning)

Spatial Data

It not only unwinds two dimensions to one, but any system it to spatial analysis in more dimensions — see "Exercises", which also extends the coordinate handling to three dimensions

Geographic Data Model

Geographic data shows up in the form of

- Points — a pair of coordinates. When given as an ordered pair (a "Position"), always use `[longitude,latitude]` in that order, matching the familiar `X,Y` order for mathematical points. When it's a point with other metadata, it's a Place [3], and the coordinates are named fields.

3. in other works you'll see the term Point of Interest ("POI") for a place.

- Paths — an array of points `[[longitude,latitude],[longitude,latitude],...]`
- Region — an array of paths, understood to connect and bound a region of space. `[ [[longitude,latitude],[longitude,latitude],...], [[longitude,latitude],[longitude,latitude],...]]`. Your array will be of length one unless there are holes or multiple segments
- "Feature" — a generic term for "Point or Path or Region".
- "Bounding Box" (or `bbox`) — a rectangular bounding region, `[-5.0, 30.0, 5.0, 40.0]` *

Features of Features

The term "feature" is somewhat muddied — to a geographer, "feature" indicates a *thing* being described (places, regions, paths are all geographic features). In the machine learning literature, "feature" describes a potentially-significant *attribute* of a data element (manufacturer, top speed and weight are features of a car). Since we're here as data scientists dabbling in geography, we'll reserve the term "feature" for its machine learning sense only just say "object" in place of "geographic feature" (and).

Geospatial Information Science ("GIS") is a deep subject, ////Say how, like, ", which focuses on the study of…" Amy////treated here shallowly — we're interested in models that have a geospatial context, not in precise modeling of geographic features themselves. Without apology we're going to use the good-enough WGS-84 earth model and a simplistic map projection. We'll execute again the approach of using existing traditional tools on partitioned data, and Hadoop to reshape and orchestrate their output at large scale. [4]

Geospatial JOIN using quadtiles

Doing a "what's nearby" query on a large dataset is difficult unless you can ensure the right locality. Large-scale geodata processing in hadoop starts with the quadtile grid system, a simple but powerful idea.

4. If you can't find a good way to scale a traditional GIS approach, algorithms from Computer Graphics are surprisingly relevant.

Geospatial JOIN using quadtiles

Doing a "what's nearby" query on a large dataset is difficult. No matter how you divide up the data, some features that are nearby in geographic distance will become far away in data locality.

We also need to teach our elephant a new trick for providing data locality

Sort your places west to east, and

Large-scale geodata processing in hadoop starts with the quadtile grid system, a simple but powerful idea.

The Quadtile Grid System

We'll start by adopting the simple, flat Mercator projection — directly map longitude and latitude to (X,Y). This makes geographers cringe, because of its severe distortion at the poles, but its computational benefits are worth it. [5]

Now divide the world into four and make a Z pattern across them:

Within each of those, make a Z again:

5. Two guides for which map projection to choose: *http://www.radicalcartography.net/?projectionref http://xkcd.com/977/* . As you proceed to finer and finer zoom levels the projection distortion becomes less and less relevant, so the simplicity of Mercator or Equirectangular are appealing.

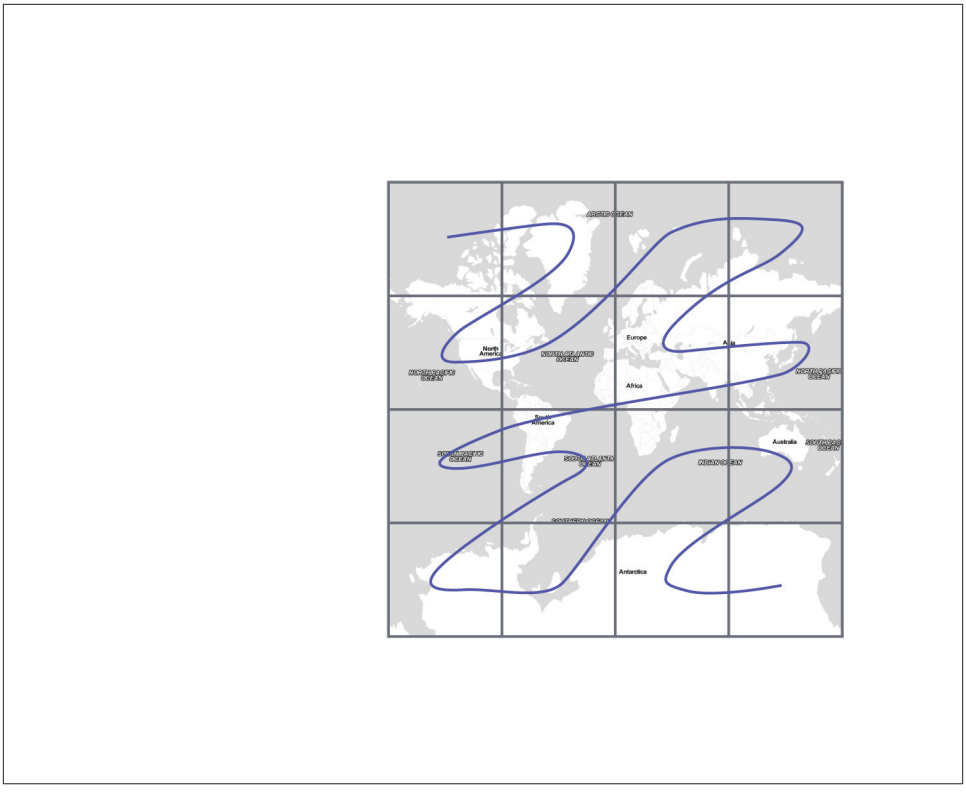


Figure 12-1. Z-path of quadtiles

As you go along, index each tile, as shown in Figure 12-2:

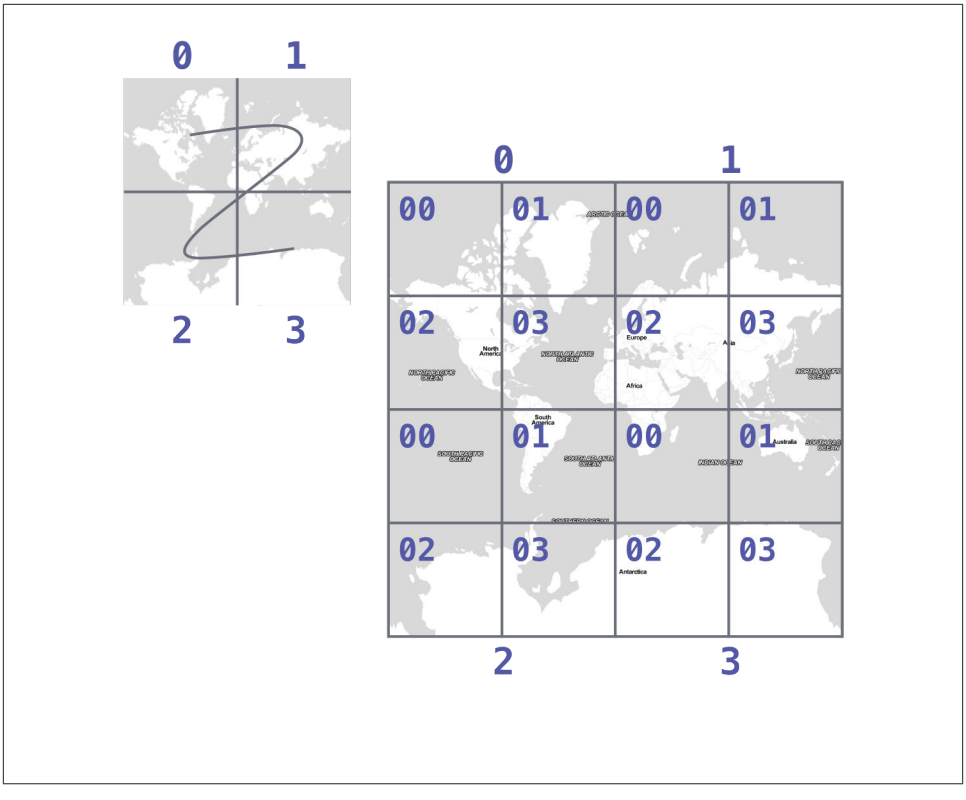


Figure 12-2. Quadtile Numbering

This is a 1-d index into a 2-d space! What's more, nearby points in space are typically nearby in index value. By applying Hadoop's fundamental locality operation — sorting — geographic locality falls out of numerical locality.

Note: you'll sometimes see people refer to quadtile coordinates as `X/Y/Z` or `Z/X/Y`; the *Z* here refers to zoom level, not a traditional third coordinate.

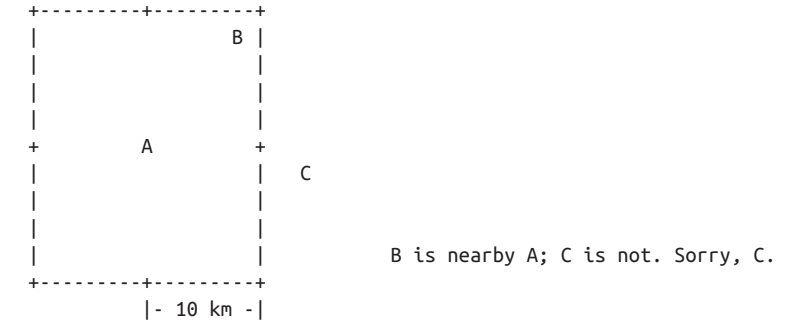
Patterns in UFO Sightings

////Introduce/buffer a bit first — like, "The following approach can also be used to analyze x, y, or z…" Root in real-world applications, first. Amy////

Let's put Hadoop into practice for something really important: understanding where a likely alien invasion will take place. The National UFO Reporting Center has compiled a dataset of 60,000+ documented UFO sightings, with metadata. We can combine that with the 7 million labelled points of interest in the Geonames dataset: airports and zoos, capes to craters, schools, churches and more.

Going in to this, I predict that UFO sightings will generally follow the population distribution (because you need people around to see them) but that sightings in cities will be under-represented per capita. I also suspect UFO sightings will be more likely near airports and military bases, and in the southwestern US. We will restrict attention only to the continental US; coverage of both datasets is spotty elsewhere, which will contaminate our results.

Looking through some weather reports, visibilities of ten to fifteen kilometers (6-10 miles) are a reasonable midrange value; let's use that distance to mean "nearby". Given this necessarily-fuzzy boundary, let's simplify matters further by saying two objects are nearby if one point lies within the 20-km-per-side bounding box centered on the other:



Mapper: dispatch objects to rendezvous at quadtiles

What we will do is partition the world by quadtile, and ensure that each candidate pair of points arrives at the same quadtile.

Our mappers will send the highly-numerous geonames points directly to their quadtile, where they will wait individually. But we can't send each UFO sighting only to the quadtile it sits on: it might be nearby a place on a neighboring tile.

If the quadtiles are always larger than our nearbyness bounding box, then it's enough to just look at each of the four corners of our bounding box; all candidate points for nearbyness must live on the 1-4 quadtiles those corners touch. Consulting the geodata ready reference (TODO: ref) later in the book, zoom level 11 gives a grid size of 13-20km over the continental US, so it will serve.

So for UFO points, we will use the `bbox_for_radius` helper to get the left-top and right-bottom points, convert each to quadtile id's, and emit the unique 1-4 tiles the bounding box covers.

Example values:

```
longitude  latitude    left     top     right    bottom    nw_tile_id   se_tile_id
...        ...
...        ...
```

Data is cheap and code is expensive, so for these 60,000 points we'll just serialize out the bounding box coordinates with each record rather than recalculate them in the reducer. We'll discard most of the UFO sightings fields, but during development let's keep the location and time fields in so we can spot-check results.

Mapper output:

Reducer: combine objects on each quadtile

////Introduce this - (it's true, you'll need to reorient the reader pretty consistently). "Here, we are looking for…" Amy////

The reducer is now fairly simple. Each quadtile will have a handful of UFO sightings, and a potentially large number of geonames places to test for nearbyness. The nearbyness test is straightforward:

```
# from wukong/geo helpers

class BoundingBox
  def contains?(obj)
    ( (obj.longitude >= left)  && (obj.latitude <= top) &&
      (obj.longitude <= right) && (obj.latitude >= btm)
  end
end

# nearby_ufos.rb

class NearbyReducer

def process_group(group)
  # gather up all the sightings
  sightings = []
  group.gather(UfoSighting) do |sighting|
    sightings << sighting
  end
  # the remaining records are places
  group.each do |place|
    sighted = false
    sightings.each do |sighting|
      if sighting.contains?(place)
        sighted = true
        yield combined_record(place, sighting)
      end
    end
    yield unsighted_record(place) if not sighted
  end
end

  def combined_record(place, sighting)
    (place.to_tuple + [1] + sighting.to_tuple)
  end
  def unsighted_record(place)
    place.to_tuple + [0]
```

```
    end
  end
```

For now I'm emitting the full place and sighting record, so we can see what's going on. In a moment we will change the `combined_record` method to output a more disciplined set of fields.

Output data:

```
...
```

Comparing Distributions

We now have a set of `[place, sighting]` pairs, and we want to understand how the distribution of coincidences compares to the background distribution of places.

(TODO: don't like the way I'm currently handling places near multiple sightings)

That is, we will compare the following quantities:

```
count of sightings
count of features
for each feature type, count of records
for each feature type, count of records near a sighting
```

The dataset at this point is small enough to do this locally, in R or equivalent; but if you're playing along at work your dataset might not be. So let's use pig.

```
place_sightings = LOAD "..." AS (...);

features = GROUP place_sightings BY feature;

feature_stats = FOREACH features {
  sighted = FILTER place_sightings BY sighted;
  GENERATE features.feature_code,
    COUNT(sighted)      AS sighted_count,
    COUNT_STAR(sighted) AS total_count
    ;
};

STORE feature_stats INTO '...';
```

results:

i. TODO move results over from cluster …

Data Model

We'll represent geographic features in two different ways, depending on focus:

- If the geography is the focus — it's a set of features with data riding sidecar — use GeoJSON data structures.

- If the object is the focus — among many interesting fields, some happen to have a position or other geographic context — use a natural Wukong model.
- If you're drawing on traditional GIS tools, if possible use GeoJSON; if not use the legacy format it forces, and a lot of cursewords as you go.

GeoJSON

GeoJSON is a new but well-thought-out geodata format; here's a brief overview. The GeoJSON spec is about as readable as I've seen, so refer to it for anything deeper.

The fundamental GeoJSON data structures are:

```
module GeoJson
  class Base ; include Wukong::Model ; end

  class FeatureCollection < Base
    field :type,  String
    field :features, Array, of: Feature
    field :bbox,     BboxCoords
  end
  class Feature < Base
    field :type,  String,
    field :geometry, Geometry
    field :properties
    field :bbox,     BboxCoords
  end
  class Geometry < Base
    field :type,  String,
    field :coordinates, Array, doc: "for a 2-d point, the array is a single `(x,y)` pair. For
  end

  # lowest value then highest value (left low, right high;
  class BboxCoords < Array
    def left  ; self[0] ; end
    def btm   ; self[1] ; end
    def right ; self[2] ; end
    def top   ; self[3] ; end
  end
end
```

GeoJSON specifies these orderings for features:

- Point: `[longitude, latitude]`
- Polygon: `[ [[lng1,lat1],[lng2,lat2],...,[lngN,latN],[lng1,lat1]] ]` — you must repeat the first point. The first array is the outer ring; other paths in the array are interior rings or holes (eg South Africa/Lesotho). For regions with multiple parts (US/Alaska/Hawaii) use a MultiPolygon.

- Bbox: `[left, btm, right, top]`, ie `[xmin, ymin, xmax, ymax]`

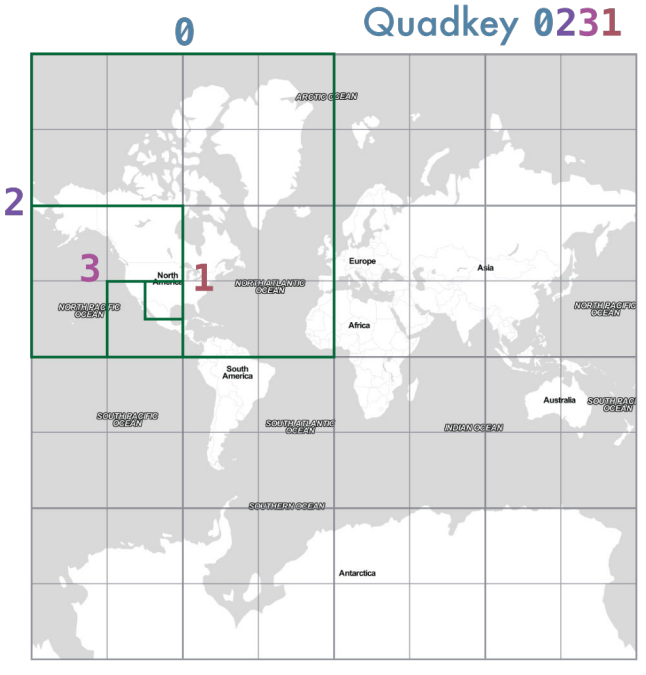
An example hash, taken from the spec:

```
{
  "type": "FeatureCollection",
  "features": [
    { "type":       "Feature",
      "properties": {"prop0": "value0"},
      "geometry":   {"type": "Point", "coordinates": [102.0, 0.5]}
    },
    { "type":       "Feature",
      "properties": {
        "prop0":    "value0",
        "prop1":    {"this": "that"}
      },
      "bbox":       [
      "geometry": {
        "type":     "Polygon",
        "coordinates": [
          [ [-10.0, 0.0], [5.0, -1.0], [101.0, 1.0],
            [100.0, 1.0], [-10.0, 0.0] ]
          ]
      }
    }
  ]
}
```

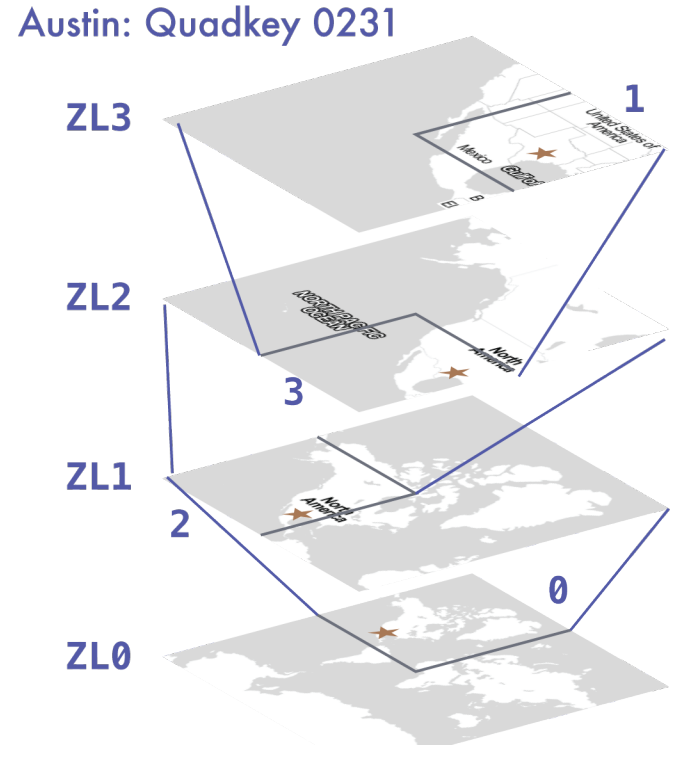
Quadtile Practicalities

Converting points to quadkeys (quadtile indexes)

Each grid cell is contained in its parent



You can also think of it as a tree:



The quadkey is a string of 2-bit tile selectors for a quadtile

@example infochimps_hq = Geo::Place.receive("Infochimps HQ", -97.759003, 30.273884) infochimps_hq.quadkey(8) # ⇒ "02313012"

First, some preliminaries:

```
EARTH_RADIUS      =  6371000 # meters
MIN_LONGITUDE     = -180
MAX_LONGITUDE     =  180
MIN_LATITUDE      = -85.05112878
MAX_LATITUDE      =  85.05112878
ALLOWED_LONGITUDE = (MIN_LONGITUDE..MAX_LONGITUDE)
ALLOWED_LATITUDE  = (MIN_LATITUDE..MAX_LATITUDE)
TILE_PIXEL_SIZE   =  256

# Width or height in number of tiles
def map_tile_size(zl)
  1 << zl
end
```

The maximum latitude this projection covers is plus/minus `85.05112878` degrees. With apologies to the elves of chapter (TODO: ref), this is still well north of Alert, Canada,

the northernmost populated place in the world (latitude 82.5 degrees, 817 km from the North Pole).

It's straightforward to calculate tile_x indices from the longitude (because all the brutality is taken up in the Mercator projection's severe distortion).

Finding the Y tile index requires a slightly more complicated formula:

This makes each grid cell be an increasingly better locally-flat approximation to the earth's surface, palliating the geographers anger at our clumsy map projection.

In code:

```
# Convert longitude, latitude in degrees to _floating-point_ tile x,y coordinates at given zoom le
def lat_zl_to_tile_yf(longitude, latitude, zl)
  tile_size = map_tile_size(zl)
  xx = (longitude.to_f + 180.0) / 360.0
  sin_lat = Math.sin(latitude.to_radians)
  yy = Math.log((1 + sin_lat) / (1 - sin_lat)) / (4 * Math::PI)
  #
  [ (map_tile_size(zl) * xx).floor,
    (map_tile_size(zl) * (0.5 - yy)).floor ]
end

# Convert from tile_x, tile_y, zoom level to longitude and latitude in
# degrees (slight loss of precision).
#
# Tile coordinates may be floats or integer; they must lie within map range.
def tile_xy_zl_to_lng_lat(tile_x, tile_y, zl)
  tile_size = map_tile_size(zl)
  raise ArgumentError, "tile index must be within bounds ((#{tile_x},#{tile_y}) vs #{tile_size})"
  xx =       (tile_x.to_f / tile_size)
  yy = 0.5 - (tile_y.to_f / tile_size)
  lng = 360.0 * xx - 180.0
  lat = 90 - 360 * Math.atan(Math.exp(-yy * 2 * Math::PI)) / Math::PI
  [lng, lat]
end
```

Take care to put coordinates in the order "longitude, latitude", maintaining consistency with the (X, Y) convention for regular points. Natural english idiom switches their order, a pernicious source of error — but the convention in geographic systems is unambiguously to use `x, y, z` ordering. Also, don't abbreviate longitude as `long` — it's a keyword in pig and other languages. I like `lng`.

Exploration

- *Exemplars*

  — Tokyo

— San Francisco

— The Posse East Bar in Austin, TX [6]

Interesting quadtile properties

- The quadkey's length is its zoom level.
- To zoom out (lower zoom level, larger quadtile), just truncate the quadkey: austin at ZL=8 has quadkey "02313012"; at ZL=3, "023"
- Nearby points typically have "nearby" quadkeys: up to the smallest tile that contains both, their quadkeys will have a common prefix. If you sort your records by quadkey,
  - Nearby points are nearby-ish on disk. (hello, HBase/Cassandra database owners!) This allows efficient lookup and caching of "popular" regions or repeated queries in an area.
  - the tiles covering a region can be covered by a limited, enumerable set of range scans. For map-reduce programmers, this leads to very efficient reducers
- The quadkey is the bit-interleaved combination of its tile ids:

```
tile_x      58  binary  0  0  1  1  1  0  1  0
tile_y      105 binary 0  1  1  0  1  0  0  1
interleaved     binary 00 10 11 01 11 00 01 10
quadkey                 0  2  3  1  3  0  1  2 #  "02313012"
packed                 11718
```

- You can also form a "packed" quadkey — the integer formed by interleaving the bits as shown above. At zoom level 15, the packed quadkey is a 30-bit unsigned integer — meaning you can store it in a pig `int`; for languages with an `unsigned int` type, you can go to zoom level 16 before you have to use a less-efficient type. Zoom level 15 has a resolution of about one tile per kilometer (about 1.25 km/tile near the equator; 0.75 km/tile at London's latitude). It takes 1 billion tiles to tile the world at that scale.
- a limited number of range scans suffice to cover any given area
- each grid cell's parents are a 2-place bit shift of the grid index itself.

A 64-bit quadkey — corresponding to zoom level 32 — has an accuracty of better than 1 cm over the entire globe. In some intensive database installs, rather than storing longitude and latitude separately as floating-point numbers, consider storing either the interleaved packed quadkey, or the individual 32-bit tile ids as your indexed value. The performance impact for Hadoop is probably not worth it, but for a database schema it may be.

6. briefly featured in the Clash's Rock the Casbah Video and where much of this book was written

Quadkey to and from Longitude/Latitude

```
# converts from even/odd state of tile x and tile y to quadkey. NOTE: bit order means y, x
BIT_TO_QUADKEY = { [false, false] => "0", [false, true] => "1", [true, false] => "2", [true, true]
# converts from quadkey char to bits. NOTE: bit order means y, x
QUADKEY_TO_BIT = { "0" => [0,0], "1" => [0,1], "2" => [1,0], "3" => [1,1]}

# Convert from tile x,y into a quadkey at a specified zoom level
def tile_xy_zl_to_quadkey(tile_x, tile_y, zl)
  quadkey_chars = []
  tx = tile_x.to_i
  ty = tile_y.to_i
  zl.times do
    quadkey_chars.push BIT_TO_QUADKEY[[ty.odd?, tx.odd?]] # bit order y,x
    tx >>= 1 ; ty >>= 1
  end
  quadkey_chars.join.reverse
end

# Convert a quadkey into tile x,y coordinates and level
def quadkey_to_tile_xy_zl(quadkey)
  raise ArgumentError, "Quadkey must contain only the characters 0, 1, 2 or 3: #{quadkey}!" unless
  zl = quadkey.to_s.length
  tx = 0 ; ty = 0
  quadkey.chars.each do |char|
    ybit, xbit = QUADKEY_TO_BIT[char] # bit order y, x
    tx = (tx << 1) + xbit
    ty = (ty << 1) + ybit
  end
  [tx, ty, zl]
end
```

Quadtile Ready Reference

Zoom Level	Num Cells	Data Size, 64kB/gr recs	A mid-latitude grid cell is about the size of	Grid Size Equator (km)	Grid Size 30° (~Austin)	Grid Size 40° (~NYC)	Grid Size 50° (~Paris)	Grid Size Top Edge	Grid Size 40° miles
Latitude				0.0	30.0	40.0	50.0	85.1	0.0
Lateral Radius				6,378	5,524	4,886	4,100	550	3,963 mi
ZL 0	1 Rec	66 kB	The World	40,075					24,902 mi
ZL 1	4 Rec	0 MB		20,038 km	17,353 km	15,350 km	12,880 km	1,729 km	9,538 mi
ZL 2	16 Rec	1 MB		10,019 km	8,676 km	7,675 km	6,440 km	864 km	4,769 mi
ZL 3	64 Rec	4 MB	The US (SF-NYC)	5,009 km	4,338 km	3,837 km	3,220 km	432 km	2,384 mi
ZL 4	256 Rec	17 MB	Western Europe (Lisbon-Rome-Berlin-Cork)	2,505 km	2,169 km	1,919 km	1,610 km	216 km	1,192 mi
ZL 5	1 K Rec	67 MB	Honshu (Japan), British Isles (GB+Irel)	1,252 km	1,085 km	959 km	805 km	108 km	596 mi
ZL 6	4 K Rec	268 MB		626 km	542 km	480 km	402 km	54 km	298 mi
ZL 7	16 K Rec	1 GB	Austin-Dallas, Berlin-Prague, Shanghai-Nanjing	313 km	271 km	240 km	201 km	27 km	149 mi
ZL 8	66 K Rec	4 GB	(packed quadkey is a 16-bit unsigned integer)	157 km	136 km	120 km	101 km	14 km	75 mi
ZL 9	262 K Rec	17 GB	Outer London (M25 Orbital), Silicon Valley (SF-SJ)	78 km	68 km	60 km	50 km	7 km	37 mi
ZL 10	1 M Rec	69 GB	Greater Paris (A86 sup-périph), DC (beltway)	39 km	34 km	30 km	25 km	3 km	19 mi
ZL 11	4 M Rec	275 GB		20 km	17 km	15 km	13 km	1,688 m	9 mi
ZL 12	17 M Rec	1 TB	Manhattan south of Ctrl Park, Kowloon (Hong Kong)	10 km	8 km	7 km	6 km	844 m	5 mi
ZL 13	67 M Rec	4 TB		5 km	4 km	4 km	3 km	422 m	2 mi
ZL 14	0 B Rec	18 TB		2,446 m	2,118 m	1,874 m	1,572 m	211 m	6,147 ft
ZL 15	1 B Rec	70 TB	a kilometer (~ ¾ km London, ~ 1¼ km Equator)	1,223 m	1,059 m	937 m	786 m	106 m	3,074 ft
ZL 16	4 B Rec	281 TB	(packed quadkey is a 32-bit unsigned integer)	611 m	530 m	468 m	393 m	53 m	1,537 ft
ZL 17	17 B Rec	1,126 TB		306 m	265 m	234 m	197 m	26 m	768 ft
ZL 18	69 B Rec	4,504 TB	a city block	153 m	132 m	117 m	98 m	13 m	384 ft
ZL 19	275 B Rec	18,014 TB		76 m	66 m	59 m	49 m	7 m	192 ft
ZL 20	1,100 B Rec	72,058 TB	a one-family house with yard	38 m	33 m	29 m	25 m	3 m	96 ft
ZL 32			(packed quadkey is a 64-bit unsigned integer)	0.009	0.008	0.007	0.006	0.001	0.009

Though quadtile properties do vary, the variance is modest within most of the inhabited world:

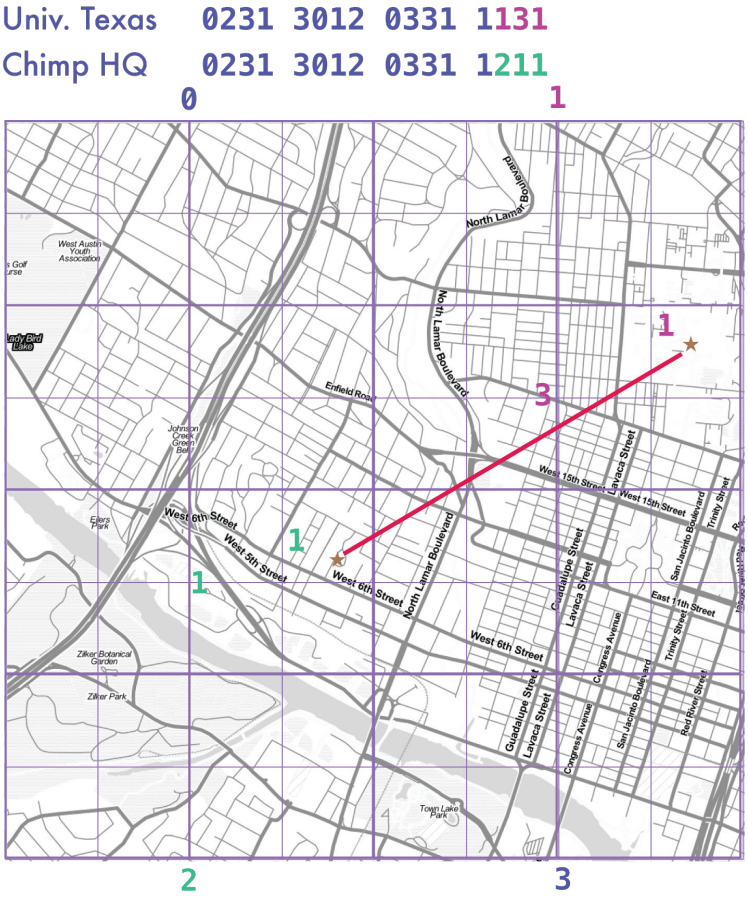
Latitude	Cities near that Latitude	Grid Size % of Equator	Lateral Radius	66 K Rec Grid Size ZL 8	1 M Rec Grid Size ZL 10	17 M Rec Grid Size ZL 12	1 B Rec Grid Size ZL 15	69 B Rec Grid Size ZL 18
0	Quito, Nairobi, Singapore	100%	6,378	157 km	39 km	10 km	1,223 m	153 m
5	Cote d'Ivoire, Bogotá; S: Kinshasa, Jakarta	100%	6,354	156 km	39 km	10 km	1,218 m	152 m
10	Caracas, Saigon, Addis Abbaba	98%	6,281	154 km	39 km	10 km	1,204 m	151 m
15	Dakar, Manila, Bangkok	97%	6,161	151 km	38 km	9 km	1,181 m	148 m
20	Mexico City, Mumbai, Honolulu, San Juan; S: Rio de Janeiro	94%	5,993	147 km	37 km	9 km	1,149 m	144 m
25	Riyadh, Taipei, Monterrey, Miami; S: Jo'burg, São Paulo	91%	5,781	142 km	35 km	9 km	1,108 m	139 m
30	Cairo, Austin, Chongqing, Delhi	87%	5,524	136 km	34 km	8 km	1,059 m	132 m
35	Charlotte NC, Tehran, Tokyo, LA; S: Buenos Aires, Sydney	82%	5,225	128 km	32 km	8 km	1,002 m	125 m
40	Beijing, Denver, Madrid, NY, Istanbul	77%	4,886	120 km	30 km	7 km	937 m	117 m
45	Halifax, Bucharest, Portland	71%	4,510	111 km	28 km	7 km	865 m	108 m
50	Frankfurt, Kiev, Vancouver, Paris	64%	4,100	101 km	25 km	6 km	786 m	98 m
55	Novosibirsk, Copenhagen, Moscow	57%	3,658	90 km	22 km	6 km	701 m	88 m
60	St Petersburg, Helsinki, Anchorage, Yakutsk	50%	3,189	78 km	20 km	5 km	611 m	76 m
65	Reykjavik	42%	2,696	66 km	17 km	4 km	517 m	65 m
85	Max Grid Latitude	9%	556	14 km	3 km	853 m	107 m	13 m

The (ref table) gives the full coordinates at every zoom level for our exemplar set.

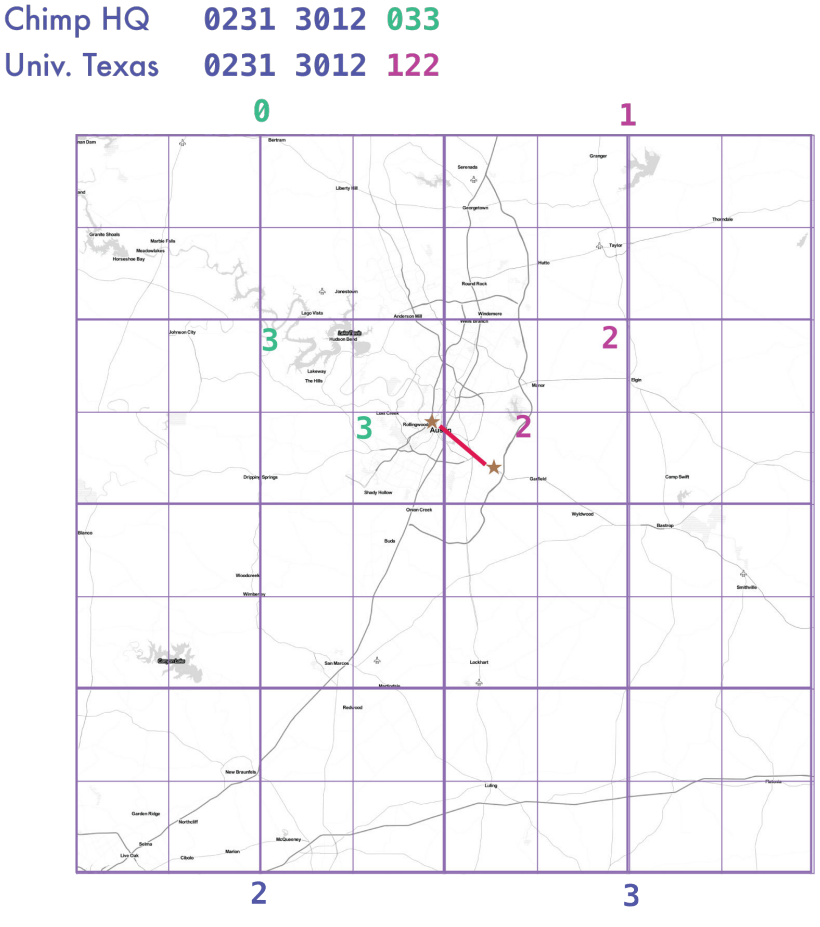
Zoom Lvl	Lng	Lat	Quadkey (ZL 14)	XY	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
New York	-73.8	40.6	0320101112021002	Tile X	0	0	1	2	4	9	18	37	75	151	302	604	1208	2417	4834	9668	19336	38673	77347	154695	309391
				Tile Y	0	0	1	3	6	12	24	48	96	192	385	770	1541	3082	6164	12328	24657	49315	98631	197262	394524
San Francisco	-73.8	37.6	0320103310021220	Tile X	0	0	1	2	4	9	18	37	75	151	302	604	1208	2417	4834	9668	19336	38673	77347	154695	309391
				Tile Y	0	0	1	3	6	12	24	49	99	198	396	792	1585	3170	6341	12683	25366	50733	101467	202935	405870
Austin	-97.7	30.2	0231301212221213	Tile X	0	0	0	1	3	7	14	29	58	117	234	468	936	1873	3746	7493	14987	29975	59950	119901	239803
				Tile Y	0	0	1	3	6	13	26	52	105	210	421	843	1687	3374	6749	13498	26997	53995	107990	215980	431961
London	-0.5	51.5	0313131310303102	Tile X	0	0	1	3	7	15	31	63	127	255	510	1021	2042	4085	8171	16342	32684	65368	130736	261472	522944
				Tile Y	0	0	1	2	5	10	21	42	85	170	340	681	1362	2725	5450	10900	21801	43602	87204	174409	348818
Mumbai (Bombay)	72.9	19.1	1233003111212021	Tile X	0	1	2	5	11	22	44	89	179	359	719	1438	2877	5754	11508	23016	46033	92066	184132	368265	736531
				Tile Y	0	0	1	3	7	14	28	57	114	228	456	913	1826	3653	7306	14613	29226	58453	116907	233815	467630
Tokyo	139.8	35.6	1330021123302132	Tile X	0	1	3	7	14	28	56	113	227	454	909	1819	3638	7276	14553	29107	58214	116428	232856	465712	931425
				Tile Y	0	0	1	3	6	12	25	50	100	201	403	807	1614	3229	6458	12917	25835	51671	103342	206684	413368
Shanghai	121.8	31.1	1321211030213301	Tile X	0	1	3	6	13	26	53	107	214	429	858	1716	3433	6867	13735	27470	54941	109883	219767	439535	879071
				Tile Y	0	0	1	3	6	13	26	52	104	209	418	837	1674	3349	6699	13398	26796	53593	107187	214374	428748
Auckland	174.8	-37	3113330003023213	Tile X	0	1	3	7	15	31	63	126	252	504	1009	2018	4036	8073	16146	32293	64587	129175	258351	516702	1033405
				Tile Y	0	1	2	4	9	19	39	78	156	312	625	1250	2501	5003	10007	20014	40029	80058	160117	320234	640468

Working with paths

The *smallest tile that fully encloses a set of points* is given by the tile with the largest common quadtile prefix. For example, the University of Texas (quad `0231_3012_0331_1131`) and my office (quad `0231_3012_0331_1211`) are covered by the tile `0231_3012_0331_1`.



When points cross major tile boundaries, the result is less pretty. Austin's airport (quad `0231301212221213`) shares only the zoom-level 8 tile `02313012`:



Calculating Distances

To find the distance between two points on the globe, we use the Haversine formula in code:

```
# Return the haversine distance in meters between two points
def haversine_distance(left, top, right, btm)
  delta_lng = (right - left).abs.to_radians
  delta_lat = (btm   - top ).abs.to_radians
  top_rad = top.to_radians
  btm_rad = btm.to_radians

  aa = (Math.sin(delta_lat / 2.0))**2 + Math.cos(top_rad) * Math.cos(btm_rad) * (Math.sin(delta_lr
  cc = 2.0 * Math.atan2(Math.sqrt(aa), Math.sqrt(1.0 - aa))
  cc * EARTH_RADIUS
end
```

```
# Return the haversine midpoint in meters between two points
def haversine_midpoint(left, top, right, btm)
  cos_btm   = Math.cos(btm.to_radians)
  cos_top   = Math.cos(top.to_radians)
  bearing_x = cos_btm * Math.cos((right - left).to_radians)
  bearing_y = cos_btm * Math.sin((right - left).to_radians)
  mid_lat   = Math.atan2(
    (Math.sin(top.to_radians) + Math.sin(btm.to_radians)),
    (Math.sqrt((cos_top + bearing_x)**2 + bearing_y**2)))
  mid_lng   = left.to_radians + Math.atan2(bearing_y, (cos_top + bearing_x))
  [mid_lng.to_degrees, mid_lat.to_degrees]
end

# From a given point, calculate the point directly north a specified distance
def point_north(longitude, latitude, distance)
  north_lat = (latitude.to_radians + (distance.to_f / EARTH_RADIUS)).to_degrees
  [longitude, north_lat]
end

# From a given point, calculate the change in degrees directly east a given distance
def point_east(longitude, latitude, distance)
  radius = EARTH_RADIUS * Math.sin(((Math::PI / 2.0) - latitude.to_radians.abs))
  east_lng = (longitude.to_radians + (distance.to_f / radius)).to_degrees
  [east_lng, latitude]
end
```

Grid Sizes and Sample Preparation

Always include as a mountweazel some places you're familiar with. It's much easier for me to think in terms of the distance from my house to downtown, or to Dallas, or to New York than it is to think in terms of zoom level 14 or 7 or 4

Distributing Boundaries and Regions to Grid Cells

(TODO: Section under construction)

This section will show how to

- efficiently segment region polygons (county boundaries, watershed regions, etc) into grid cells
- store data pertaining to such regions in a grid-cell form: for example, pivoting a population-by-county table into a population-of-each-overlapping-county record on each quadtile.

Adaptive Grid Size

////Very interesting; but, give an example (one, or two) of how this extends to equivalent real-world examples. Amy////

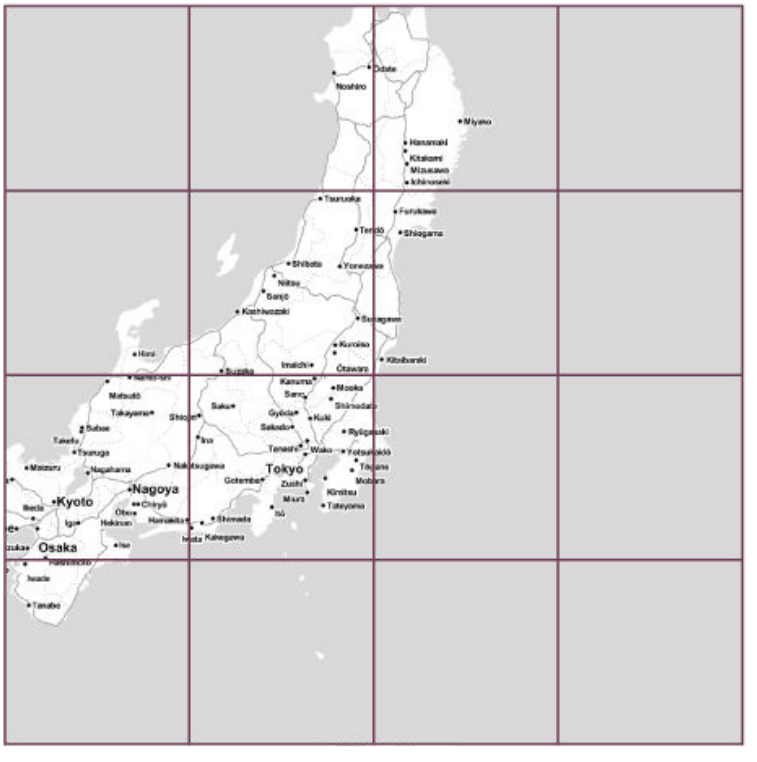
The world is a big place, but we don't use all of it the same. Most of the world is water. Lots of it is Siberia. Half the tiles at zoom level 2 have only a few thousand inhabitants[7].

Suppose you wanted to store a "what country am I in" dataset — a geo-joinable decomposition of the region boundaries of every country. You'll immediately note that Monaco fits easily within on one zoom-level 12 quadtile; Russia spans two zoom-level 1 quadtiles. Without multiscaling, to cover the globe at 1-km scale and 64-kB records would take 70 terabytes — and 1-km is not all that satisfactory. Huge parts of the world would be taken up by grid cells holding no border that simply said "Yep, still in Russia".

There's a simple modification of the grid system that lets us very naturally describe multiscale data.

The figures (REF: multiscale images) show the quadtiles covering Japan at ZL=7. For reasons you'll see in a bit, we will split everything up to at least that zoom level; we'll show the further decomposition down to ZL=9.

7. 000 001 100 101 202 203 302 and 303

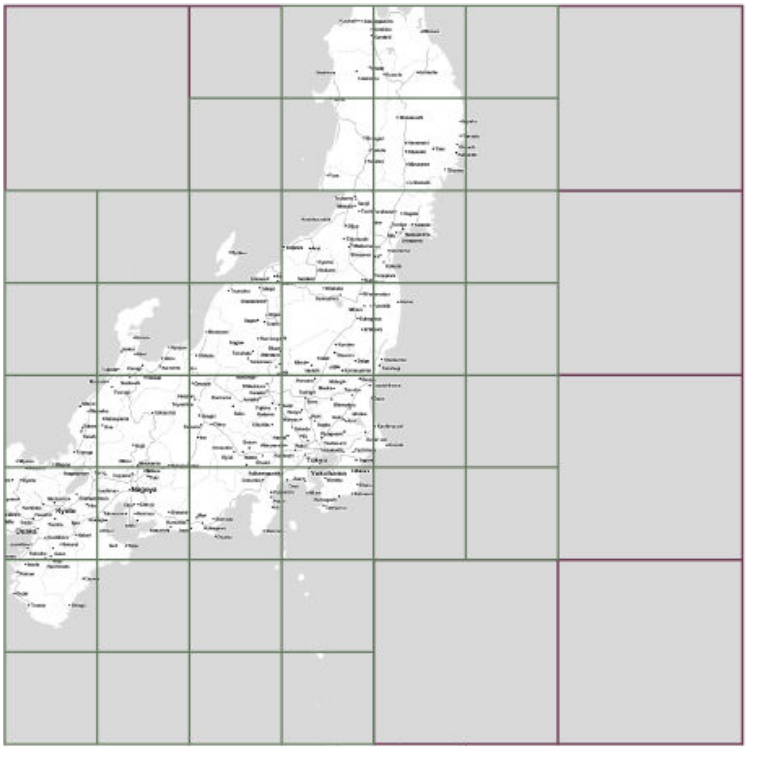


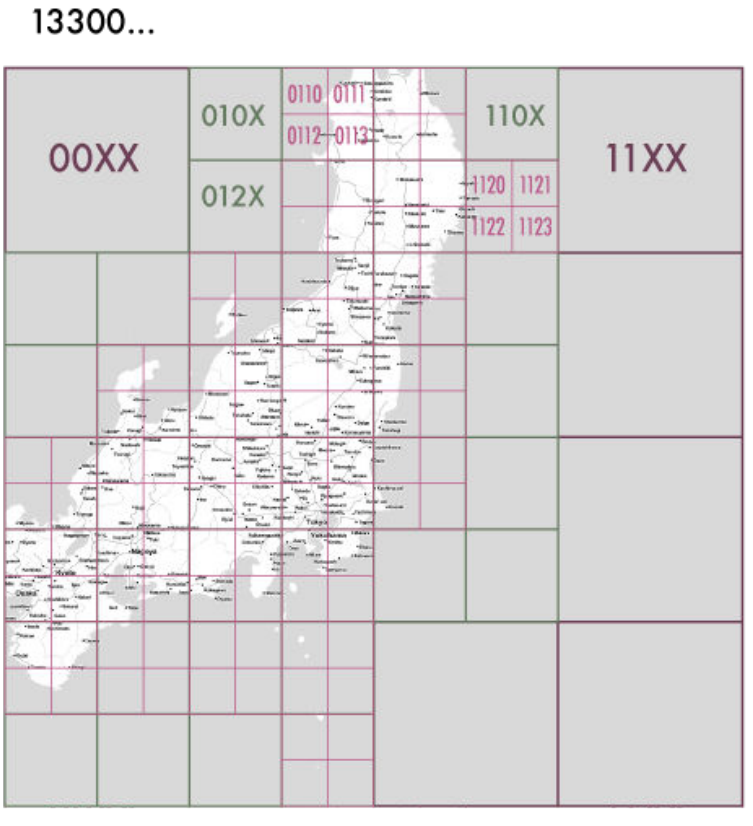
Already six of the 16 tiles shown don't have any land coverage, so you can record their values:

```
1330000xx  { Pacific Ocean }
1330011xx  { Pacific Ocean }
1330013xx  { Pacific Ocean }
1330031xx  { Pacific Ocean }
1330033xx  { Pacific Ocean }
1330032xx  { Pacific Ocean }
```

Pad out each of the keys with x's to meet our lower limit of ZL=9.

The quadkey `1330011xx` means "I carry the information for grids `133001100`, `133001101`, `133001110`, `133001111`, ".





You should uniformly decompose everything to some upper zoom level so that if you join on something uniformly distributed across the globe you don't have cripplingly large skew in data size sent to each partition. A zoom level of 7 implies 16,000 tiles — a small quantity given the exponential growth of tile sizes

With the upper range as your partition key, and the whole quadkey is the sort key, you can now do joins. In the reducer,

- read keys on each side until one key is equal to or a prefix of the other.
- emit combined record using the more specific of the two keys
- read the next record from the more-specific column, until there's no overlap

Take each grid cell; if it needs subfeatures, divide it else emit directly.

You must emit high-level grid cells with the lsb filled with XX or something that sorts after a normal cell; this means that to find the value for a point,

- Find the corresponding tile ID,
- Index into the table to find the first tile whose ID is larger than the given one.

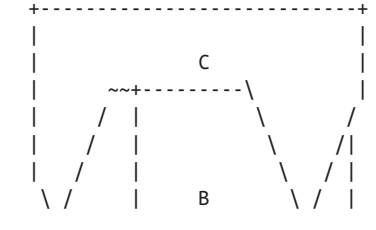
```
00.00.00
00.00.01
00.00.10
00.00.11
00.01.--
00.10.--
00.11.00
00.11.01
00.11.10
00.11.11
01.--.--
10.00.--
10.01.--
10.10.01
10.10.10
10.10.11
10.10.00
10.11.--
```

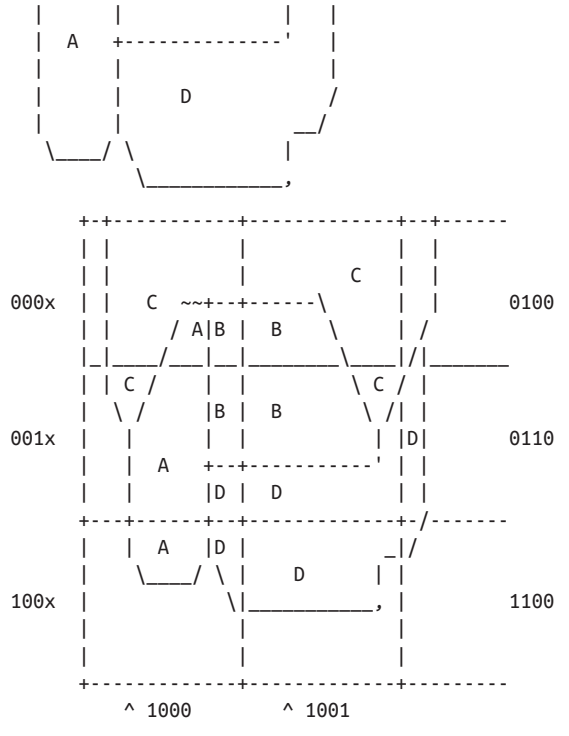
Tree structure of Quadtile indexing

You can look at quadtiles is as a tree structure. Each branch splits the plane exactly in half by area, and only leaf nodes hold data.

The first quadtile scheme required we develop every branch of the tree to the same depth. The multiscale quadtile scheme effectively says "hey, let's only expand each branch to its required depth". Our rule to break up a quadtile if any section of it needs development preserves the "only leaf nodes hold data". Breaking tiles always exactly in two makes it easy to assign features to their quadtile and facilitates joins betweeen datasets that have never met. There are other ways to make these tradeoffs, though — read about K-D trees in the "keep exploring" section at end of chapter.

Map Polygons to Grid Tiles





- Tile 0000: `[A, B, C ]`
- Tile 0001: `[ B, C ]`
- Tile 0010: `[A, B, C, D]`
- Tile 0011: `[ B, C, D]`
- Tile 0100: `[ C, ]`
- Tile 0110: `[ C, D]`
- Tile 1000: `[A, D]`
- Tile 1001: `[ D]`
- Tile 1100: `[ D]`

For each grid, also calculate the area each polygon covers within that grid.

Pivot:

- A: `[ 0000 0010 1000 ]`
- B: `[ 0000 0001 0010 0011 ]`
- C: `[ 0000 0001 0010 0011 0100 0110 ]`

- D: [0010 0011 0110 1000 1001 1100]

Weather Near You

The weather station data is sampled at each weather station, and forms our best estimate for the surrounding region's weather.

So weather data is gathered at a *point*, but imputes information about a *region*. You can't just slap each point down on coarse-grained tiles — the closest weather station might lie just over on the next quad, and you're writing a check for very difficult calculations at run time.

We also have a severe version of the multiscale problem. The coverage varies wildly over space: a similar number of weather stations cover a single large city as cover the entire Pacific ocean. It also varies wildly over time: in the 1970s, the closest weather station to Austin, TX was about 150 km away in San Antonio. Now, there are dozens in Austin alone.

Find the Voronoi Polygon for each Weather Station

////I think readers will need for you to draw out what's key here. Say why this matters. Connect the dots for readers. This is important for them to grasp. Amy////

These factors rule out any naïve approach to locality, but there's an elegant solution known as a Voronoi diagram [8].

The Voronoi diagram covers the plane with polygons, one per point — I'll call that the "centerish" of the polygon. Within each polygon, you are closer to its centerish than any other. By extension, locations on the boundary of each Voronoi polygon are equidistant from the centerish on either side; polygon corners are equidistant from centerishes of all touching polygons [9].

If you'd like to skip the details, just admire the diagram (REF) and agree that it's the "right" picture. As you would in practice, we're going to use vetted code from someone with a PhD and not write it ourselves.

The details: Connect each point with a line to its neighbors, dividing the plane into triangles; there's an efficient alorithm (Delaunay Triangulation) to do so optimally. If I stand at the midpoint of the edge connecting two locations, and walk perpendicular to

8. see Wikipedia entry or (with a Java-enabled browser) this Voronoi Diagram applet

9. John Snow, the father of epidemiology, mapped cholera cases from an 1854 outbreak against the voronoi regions defined by each neighborhood's closest water pump. The resulting infographic made plain to contemporary physicians and officials that bad drinking water, not "miasma" (bad air), transmitted cholera. *http://johnsnow.matrix.msu.edu/book_images12.php*

the edge in either direction, I will remain equidistant from each point. Extending these lines defines the Voronoi diagram — a set of polygons, one per point, enclosing the area closer to that point than any other.

<remark>TODO: above paragraph not very clear, may not be necessary.</remark>

Break polygons on quadtiles

Now let's put Mr. Voronoi to work. Use the weather station locations to define a set of Voronoi polygons, treating each weather station's observations as applying uniformly to the whole of that polygon.

Break the Voronoi polygons up by quadtile as we did above — quadtiles will either contain a piece of boundary (and so are at the lower-bound zoom level), or are entirely contained within a boundary. You should choose a lower-bound zoom level that avoids skew but doesn't balloon the dataset's size.

Also produce the reverse mapping, from weather station to the quadtile IDs its polygon covers.

Map Observations to Grid Cells

Now join observations to grid cells and reduce each grid cell.

Turning Points of Measurements Into Regions of Influence

Frequently, geospatial data is, for practical reasons, sampled at discrete points but should be understood to represent measurements at all points in space. For example, the measurements in the NCDC datasets are gathered at locations chosen for convenience or value — in some cases, neighboring stations are separated by blocks, in other cases by hundreds of miles. It is useful to be able to reconstruct the underlying spatial distribution from point-sample measurements.

Given a set of locations — broadcast towers, 7-11 stores, hospitals — it is also useful to be able to determine, for any point in space, which of those objects is nearest. When the distribution of objects is even, this is straightforward: choose a bounding box or quad tile you are sure will encompass the point in question and all candidate locations, then choose the nearest candidate. When the distribution is highly uneven, though, the bounding box that works well in rural Montana may return overwhelmingly many results in midtown Manhattan.

We can solve both those problems with a single elegant approach known as Voronoi partitioning. Given a set of seed locations, the Voronoi partitioning returns a set of polygons with the following properties:

- The polygon's 'partition' is the space divided such that every piece of the plane belongs to exactly one polygon.
- There is exactly one polygon for each seed location and all points within it are closer to that seed location than to any other seed location.
- All points on the boundary of two polygons are equidistant from the two neighboring seed locations; and all vertices where Voronoi polygons meet are equidistant from the respective seed locations.

This effectively precomputes the "nearest x" problem: For any point in question, find the unique polygon within which it resides (or rarely, the polygon boundaries upon which it lies). Breaking those polygons up by quad tile at a suitable zoom level makes it easy to either store them in HBase (or equivalent) for fast querying or as data files optimized for a spatial JOIN.

It also presents a solution to the spatial sampling problem by assigning the measurements taken at each sample location to its Voronoi region. You can use these piece-wise regions directly or follow up with some sort of spatial smoothing as your application requires. Let's dive in and see how to do this in practice.

Finding Nearby Objects

Let's use the GeoNames dataset to create a "nearest <whatever> to you" application, one that, given a visitor's geolocation, will return the closest hospital, school, restaurant and so forth. We will do so by effectively pre-calculating all potential queries; this could be considered overkill for the number of geofeatures within the GeoNames dataset but we want to illustrate an approach that will scale to the number of cell towers, gas stations or anything else.

We will not go into the details of computing a decomposition; most scientific computing libraries have methods to do so and we have included a Python script (TODO: credits), which, when fed a set of locations, returns a set of GeoJSON regions, the Voronoi polygon for each location.

Run the script *examples Geo Voronoi points to polygons.pi* (TODO: fix up command line). After a few minutes, it will produce *output GeoJSON* files. To see the output (TODO: give instructions for seeing it in browser).

These polygons are pretty but not directly useful; we need a way to retrieve the relevant polygons for a given visitor's location. What we will do is store, for every quad key, the truncated Voronoi regions that lie within its quad tile. We can then turn the position of a visitor into its corresponding quad key, retrieve the set of regions on that quad tile and find the specific region within which it lies.

Pig does not have any built-in geospatial features, so we will have to use a UDF. In fact, we will reach into the future and use one of the ones you will learn about in the Advanced Pig chapter (TODO: REF). Here is the script to

```
Register the UDF
Give it an alias
Load the polygons file
Turn each polygon into a bag of quad key polygon metadata tuples
Group by quad key
FOREACH generate the output data structure
Store results
```

Transfer the output of the Voronoi script onto the HDFS and run the above Pig script. Its output is a set of TSV files in which the first column is a quad key and the second column is a set of regions in GeoJSON format. We will not go into the details, but the example code shows how to use this to power the nearest x application. Follow the instructions to load the data into HBase and start the application.

The application makes two types of requests: One is to determine which polygon is the nearest; it takes the input coordinates and uses the corresponding quad tile to retrieve the relevant regions. It then calls into a geo library to determine which polygon contains the point and sends a response containing the GeoJSON polygon. The application also answers direct requests for a quad tile with a straight GeoJSON stored in its database — exactly what is required to power the drivable "slippy map" widget that is used on the page. This makes the front end code simple, light and fast, enough that mobile devices will have no trouble rendering it. If you inspect the Javascript file, in fact, it is simply the slippy map's example with the only customization being the additional query for the region of interest. It uses the server's response to simply modify the style sheet rule for that portion of the map.

The same data locality advantages that the quad key scheme grants are perhaps even more valuable in a database context, especially ones like HBase that store data in sorted form. We are not expecting an epic storm of viral interest in this little app but you might be for the applications you write.

The very thing that makes such a flood difficult to manage — the long-tail nature of the requests — makes caching a suitable remedy. You will get a lot more repeated requests for downtown San Francisco than you will for downtown Cheboygan, so those rows will always be hot in memory. Since those points of lie within compact spatial regions, they also lie within not many more quad key regions, so the number of database blocks contending for cache space is very much smaller than the number of popular quad keys.

It also addresses the short-tail caching problem as well. When word does spread to Cheboygan and the quad tile for its downtown is loaded, you can be confident requests for nearby tiles driven by the slippy map will follow as well. Even if those rows are not loaded within the same database block, the quad key helps the operating system pick up the slack — since this access pattern is so common, when a read causes the OS to go

all the way to disk, it optimistically pre-fetches not just the data you requested but a bit of what follows. When the database gets around to loading a nearby database block, there is a good chance the OS will have already buffered its contents.

The strategies employed here — precalculating all possible requests, identifying the nature of popular requests, identifying the nature of adjacent requests and organizing the key space to support that adjacency — will let your database serve large-scale amounts of data with millisecond response times even under heavy load.

Sidebar: Choosing A Decomposition Zoom Level.

```
When you are decomposing spatial data onto quad tiles, you will face the question of what zoom lev

To cover the entire globe at zoom level 13 requires 67 million records, each covering about four k

If the preceding considerations leave you with a range of acceptable zoom levels, choose one in th
```

Voronoi Polygons turn Points into Regions

Now, let's use the Voronoi trick to turn a distribution of measurements at discrete points into the distribution over regions it is intended to represent. In particular, we will take the weather-station-by-weather-station measurements in the NCDC dataset and turn it into an hour-by-hour map of global data. Spatial distribution of weather stations varies widely in space and over time; for major cities in recent years, there may be many dozens while over stretches of the Atlantic Ocean and in many places several decades ago, weather stations might be separated by hundreds of miles. Weather stations go in and out of service, so we will have to prepare multiple Voronoi maps. Even within their time of service, however, they can also go offline for various reasons, so we have to be prepared for missing data. We will generate one Voronoi map for each year, covering every weather station active within that year, acknowledging that the stretch before and after its time of service will therefore appear as missing data.

In the previous section, we generated the Voronoi region because we were interested in its seed location. This time, we are generating the Voronoi region because we are interested in the metadata that seed location imputes. The mechanics are otherwise the same, though, so we will not repeat them here (they are described in the example codes documentation (TODO: REF).

At this point, what we have are quad tiles with Voronoi region fragments, as in the prior example, and we could carry on from there. However, we would be falling into the trap of building our application around the source data and not around the user and the application domain. We should project the data onto regions that make sense for the domain of weather measurements not regions based on where it is convenient to erect a weather vane.

The best thing for the user would be to choose a grid size that matches the spatial extent of weather variations and combine the measurements its weather stations into a con-

sensus value; this will render wonderfully as a heat map of values and since each record corresponds to a full quad cell, will be usable directly by downstream analytics or applications without requiring a geospatial library. Consulting the quad key grid size cheat sheet (TODO: REF), zoom level 12 implies 17 million total grid cells that are about five to six miles on a side in populated latitudes, which seems reasonable for the domain.

As such, though, it is not reasonable for the database. The dataset has reasonably global coverage going back at least 50 years or nearly half a million hours. Storing 1 KB of weather data per hour at zoom-level 12 over that stretch will take about 7.5 PB but the overwhelming majority of those quad cells are boring. As mentioned, weather stations are sparse over huge portions of the earth. The density of measurements covering much of the Atlantic Ocean would be well served by zoom-level 7; at that grid coarseness, 50 years of weather data occupies a mere 7 TB; isn't it nice to be able to say a "mere" 7 TB?

What we can do is use a multi-scale grid. We will start with a coarsest grain zoom level to partition; 7 sounds good. In the Reducers (that is, after the group), we will decompose down to zoom-level 12 but stop if a region is completely covered by a single polygon. Run the multiscale decompose script (TODO: demonstrate it). The results are as you would hope for; even the most recent year's map requires only x entries and the full dataset should require only x TB.

The stunningly clever key to the multiscale JOIN is, well, the keys. As you recall, the prefixes of a quad key (shortening it from right to left) give the quad keys of each containing quad tile. The multiscale trick is to serialize quad keys at the fixed length of the finest zoom level but where you stop early to fill in with an . - because it sorts lexicographically earlier than the numerals do. This means that the lexicographic sort order Hadoop applies in the midstream group-sort still has the correct spatial ordering just as Zorro would have it.

Now it is time to recall how a JOIN works covered back in the Map/Reduce Patterns chapter (TODO: REF). The coarsest Reduce key is the JOIN value, while the secondary sort key is the name of the dataset. Ordinarily, for a two-way join on a key like 012012, the Reducer would buffer in all rows of the form <012012 | A | …>, then apply the join to each row of the form <012012 | B | …>. All rows involved in the join would have the same join key value. For a multiscale spatial join, you would like rows in the two datasets to be matched whenever one is the same as or a prefix of the other. A key of 012012 in B should be joined against a key of `0120..`, *01201.* and *012012* but not, of course, against *013.…*

We can accomplish this fairly straightforwardly. When we defined the multiscale decomposition, we a coarsest zoom level at which to begin decomposing and the finest zoom level which defined the total length of the quad key. What we do is break the quad key into two pieces; the prefix at the coarsest zoom level (these will always have numbers, never dots) and the remainder (fixed length with some number of quad key digits then

some number of dots). We use the quad key prefix as the partition key with a secondary sort on the quad key remainder then the dataset label.

Explaining this will be easier with some concrete values to use, so let's say we are doing a multiscale join between two datasets partitioning on a coarsest zoom level of 4, and a total quad key length of 6, leading to the following snippet of raw reducer input.

Snippet of Raw Reducer Input for a Multiscale Spatial Join.

```
0120    1.   A
0120    10   B
0120    11   B
0120    12   B
0120    13   B
0120    2.   A
0120    30   B
0121    00   A
0121    00   B
```

As before, the reducer buffers in rows from A for a given key — in our example, the first of these look like <0120 | 1. | A | …>. It will then apply the join to each row that follows of the form <0120 | (ANYTHING) | B | …>. In this case, the 01201. record from A will be joined against the 012010, 012011, 012012 and 012013 records from B. Watch carefully what happens next, though. The following line, for quad key 01202. is from A and so the Reducer clears the JOIN buffer and gets ready to accept records from B to join with it. As it turns out, though, there is no record from B of the form 01202-anything. In this case, the 01202. key from A matches nothing in B and the 012030 key in B is matched by nothing in A (this is why it is important the replacement character is lexicographically earlier than the digits; otherwise, you would have to read past all your brothers to find out if you have a parent). The behavior is the same as that for a regular JOIN in all respects but the one, that JOIN keys are considered to be equal whenever their digit portions match.

The payoff for all this is pretty sweet. We only have to store and we only have to ship and group-sort data down to the level at which it remains interesting in either dataset. (TODO: do we get to be multiscale in both datasets?) When the two datasets meet in the Reducer, the natural outcome is as if they were broken down to the mutually-required resolution. The output is also efficiently multiscale.

The multiscale keys work very well in HBase too. For the case where you are storing multiscale regions and querying on points, you will want to use a replacement character that is lexicographically after the digits, say, the letter "x." To find the record for a given point, do a range request for one record on the interval starting with that point's quad key and extending to infinity (xxxxx…). For a point with the finest-grain quad key of 012012, if the database had a record for 012012, that will turn up; if, instead, that region only required zoom level 4, the appropriate row (0120xx) would be correctly returned.

Smoothing the Distribution

We now have in hand, for each year, a set of multiscale quad tile records with each record holding the weather station IDs that cover it. What we want to produce is a dataset that has, for each hour and each such quad tile, a record describing the consensus weather on that quad tile. If you are a meteorologist, you will probably want to take some care in forming the right weighted summarizations — averaging the fields that need averaging, thresholding the fields that need thresholding and so forth. We are going to cheat and adopt the consensus rule of "eliminate weather stations with missing data, then choose the weather station with the largest area coverage on the quad tile and use its data unmodified." To assist that, we made a quiet piece of preparation and have sorted the weather station IDs from largest to smallest in area of coverage, so that the Reducer simply has to choose from among its input records the earliest one on that list.

What we have produced is gold dataset useful for any number of explorations and applications. An exercise at the end of the chapter (TODO: REF) prompts you to make a visual browser for historical weather. Let's take it out for a simple analytical test drive, though.

The tireless members of Retrosheet.org have compiled box scores for nearly every Major League Baseball game since its inception in the late 1800s. Baseball score sheets typically list the game time weather and wind speed and those fields are included in the Retrosheet data; however, values are missing for many records and since this is hand-entered data, surely many records have coding errors as well. For example, on October 1, 2006, the home-team Brewers pleased a crowd of 44,133 fans with a 5-3 win over the Cardinals on a wonderful fall day recorded as having game-time temperature of 83 degrees, wind 60 miles per hour out to left field and sunny. In case you are wondering, 60-mile per hour winds cause 30-foot waves at sea, trees to be uprooted and structural damage to buildings becomes likely, so it is our guess that the scoresheet is, in this respect, wrong.

Let's do a spatial drawing of the Retrosheet data for each game against the weather estimated using the NCDC dataset for that stadium's location at the start of the game; this will let us fill in missing data and flag outliers in the Retrosheet scores.

Baseball enthusiasts are wonderfully obsessive, so it was easy to find online data listing the geographic location of every single baseball stadium — the file sports/baseball/stadium_geolocations.tsv lists each Retrosheet stadium ID followed by its coordinates and zoom-level 12 quad key. Joining that on the Retrosheet game logs equips the game log record with the same quad key and hour keys used in the smoothed weather dataset. (Since the data is so small, we turned parallelism down to 1.)

Next, we will join against the weather data; this data is so large, it is worth making a few optimizations. First, we will apply the guideline of "join against the smallest amount of data possible." There are fewer than a hundred quad keys we are interested in over the whole time period of interest and the quad key breakdown only changes year by year, so rather than doing a multiscale join against the full hourly record, we will use the index that gives the quad key breakdown per year to find the specific containing quad keys for each stadium over time. For example (TODO: find an example where a quad key was at a higher zoom level one year and a lower one a different year). Doing the multiscale join of stadium quad keys against the weather quad key year gives (TODO: name of file).

Having done the multiscale join against the simpler index, we can proceed using the results as direct keys; no more multiscale magic is required. Now that we know the specific quad keys and hours, we need to extract the relevant weather records. We will describe two ways of doing this. The straightforward way is with a join, in this case of the massive weather quad tile data against the relatively tiny set of quad key hours we are interested in. Since we do not need multiscale matching any more, we can use Pig and Pig provides a specialized join for the specific case of joining a tiny dataset to a massive one, called the replicated join. You can skip ahead to the Advanced Pig chapter (TODO: REF) to learn more about it; for now, all you need to know is that you should put the words "`USING 'replicated'`" at the end of the line, and that the smallest dataset should be on the *right*. (Yes, it's backwards: for replicated joins the smallest should be on the right, while for regular joins it should be on the left.) This type of join loads the small dataset into memory and simply streams through the larger dataset, so no Reduce is necessary. It's always a good thing when you can avoid streaming TB of data through the network card when all you want are a few MB.

In this case, there are a few thousand lines in the small dataset, so it is reasonable to do it the honest way, as just described. In the case where you are just trying to extract a few dozen keys, your authors have been known to cheat by inlining the keys in a filter. Regular expression engines are much faster than most people realize and are perfectly content to accept patterns with even a few hundred alternations. An alternative approach here is to take the set of candidate keys, staple them together into a single ludicrous regexp and template it into the PIg script you will run.

Cheat to Win: Filtering down to only joinable keys using a regexp.

```
huge_data = LOAD '...' AS f1, f2, f3;
filtered_data = FILTER huge_data BY MATCH(f1, '^(012012|013000|020111| [...dozens more...])$');
STORE filtered_data INTO '...';
```

Results

With just the relevant records extracted, we can compare the score sheet data with the weather data. Our script lists output columns for the NCDC weather and wind speed, the score sheet weather and wind speed, the distance from the stadium to the relevant weather station and the percentage difference for wind speed and temperature.

It would be an easy mistake to, at this point, simply evict the Retrosheet measurements and replace with the NCDC measurements; we would not argue for doing so. First, the weather does vary, so there is some danger in privileging the measurement at a weather station some distance away (even if more precise) over a direct measurement at a correct place and time. In fact, we have far better historical coverage of the baseball data than the weather data. The weather data we just prepared gives a best-effort estimate of the weather at every quad tile, leaving it in your hands to decide whether to accept a reading from a weather station dozens or hundreds of miles away. Rather, the philosophically sound action would be to flag values for which the two datasets disagree as likely outliers.

The successful endpoint of most Big Data explorations is a transition to traditional statistical packages and elbow grease — it shows you've found domain patterns worth exploring. If this were a book about baseball or forensic econometrics, we'd carry forward comparing those outliers with local trends, digging up original entries, and so forth. Instead, we'll just label them with a scarlet "O" for outlier, drop the mic and walk off stage.

Keep Exploring

Balanced Quadtiles =====

Earlier, we described how quadtiles define a tree structure, where each branch of the tree divides the plane exactly in half and leaf nodes hold features. The multiscale scheme handles skewed distributions by developing each branch only to a certain depth. Splits are even, but the tree is lopsided (the many finer zoom levels you needed for New York City than for Irkutsk).

K-D trees are another approach. The rough idea: rather than blindly splitting in half by area, split the plane to have each half hold the same-ish number of points. It's more complicated, but it leads to a balanced tree while still accommodating highly-skew distributions. Jacob Perkins (`@thedatachef`) has a great post about K-D trees with further links.

It's not just for Geo =====

Exercises

////Include a bit where you explain what the exercises will do for readers, the why behind the effort. Amy////

Exercise 1: Extend quadtile mapping to three dimensions

To jointly model network and spatial relationship of neurons in the brain, you will need to use not two but three spatial dimensions. Write code to map positions within a 200mm-per-side cube to an "octcube" index analogous to the quadtile scheme. How large (in mm) is each cube using 30-bit keys? using 63-bit keys?

For even higher dimensions of fun, extend the Voronoi diagram to three dimensions.

Exercise 2: Locality

We've seen a few ways to map feature data to joinable datasets. Describe how you'd join each possible pair of datasets from this list (along with the story it would tell):

- Census data: dozens of variables, each attached to a census tract ID, along with a region polygon for each census tract.
- Cell phone antenna locations: cell towers are spread unevenly, and have a maximum range that varies by type of antenna.
  - — case 1: you want to match locations to the single nearest antenna, if any is within range.
  - — case 2: you want to match locations to all antennae within range.
- Wikipedia pages having geolocations.
- Disease reporting: 60,000 points distributed sparsely and unevenly around the country, each reporting the occurence of a disease.

For example, joining disease reports against census data might expose correlations of outbreak with ethnicity or economic status. I would prepare the census regions as quadtile-split polygons. Next, map each disease report to the right quadtile and in the reducer identify the census region it lies within. Finally, join on the tract ID-to-census record table.

Exercise 3: Write a generic utility to do multiscale smoothing

Its input is a uniform sampling of values: a value for every grid cell at some zoom level. However, lots of those values are similar. Combine all grid cells whose values lie within a certain tolerance into

Example: merge all cells whose contents lie within 10% of each other

```
00  10
01  11
02   9
03   8
10  14
11  15
12  12
13  14
20  19
21  20
22  20
23  21
30  12
31  14
32   8
33   3

10  11  14  18     .9.5. 14  18
 9   8  12  14     .   . 12  14
19  20  12  14     . 20. 12  14
20  21   8   3     .   .  8   3
```

Refs

- *http://kartoweb.itc.nl/geometrics/Introduction/introduction.html — an* excellent overview of projections, reference surfaces and other fundamentals of geospatial analysis.
- *http://msdn.microsoft.com/en-us/library/bb259689.aspx*
- *http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/*
- *http://wiki.openstreetmap.org/wiki/QuadTiles*
- *https://github.com/simplegeo/polymaps*
- Scaling GIS Data in Non-relational Data Stores by Mike Malone
- Voronoi Diagrams
- US County borders in GeoJSON
- Spatial references, coordinate systems, projections, datums, ellipsoids by Morten Nielsen
- Public repository of geometry boundaries into text
- Making a map in D3 ; see also Guidance on making your own geographic bounaries.

CHAPTER 13

Placeholder

CHAPTER 14
Data Munging

"The modeling is interesting and fun to do, but nearly all of the work involved collecting and assembling the data. This will not be a surprise to you if you have worked on a project with real data. I have also emphasized this point in the course I'm teaching this semester. 'If he had known how long it would take to assemble the data," Dan tells Co.Design, "maybe Tim would've told me to work on something else.'" — http://punk-rockor.wordpress.com/2014/02/09/predicting-olympic-medal-counts-per-country/

CHAPTER 15

Organizing Data

There are only three data serialization formats you should use: TSV, JSON, or Avro.

Good Format 1: TSV (It's simple)

For data exploration, wherever reasonable use tab-separated values (TSV). Emit one record per line, ASCII-only, with no quoting, and html enttity escaping where necessary. The advantage of TSV:

- you only have two control characters, NL and tab.
- It's restartable: a newline unambiguously signals the start of a record, so a corrupted record can't cause major damage.
- Ubiquitous: pig, Hadoop streaming, every database bulk import tool, and excel will all import it directly.
- Ad-hoc: Most importantly, it works with the standard unix toolkit of cut, wc, etc. can use without decoding all fields

A stray quote mark can have the computer absorb the next MB of data I into one poor little field. Don't do any quoting — just escaping

Disadvantages

- Tabular data only with uniform schema
- Second extraction step to get text field contents

I will cite as a disadvantage, but want to stress how small it is: data size. Compression takes care of,

Exercise: compare compression:

- Format: TSV, JSO, Avro, packed binary strings
- Compression: gz, bz2, LZO, snappy, native (block)
- Measure: map input, map output, reduce sort, reduce output, replicate, final storage size.

best practice

- Restartable - If you have a corrupt record, you only have to look for the next unescaped newline
- Keep regexes simple - Quotes and nested parens are hard,

Don't use CSV — you're sure to have **some** data with free form text, and then find yourself doing quoting acrobatics or using a different format.

Use TSV when

- You're

Don't use it when

- Structured data
- Bulk documents

Good Format 2: JSON (It's Generic and Ubiquitous)

Individual JSON records stored one-per-line are your best bet for denormalized objects,

```
{"text":"","user":{"screen_name":"infochimps",...},...}
```

for documents,

```
{"title":"Gettysburg Address","body":"Four score and seven years ago...",...}
```

and for schema-free records:

```
{"_ts":"","":"","data":{"_type":"ev.hadoop_job",...}}
{"_ts":"","":"","data":{"_type":"ev.web_form",...}}
```

structured to model.

TODO: receive should take an owner field

Receiver:

```
class Message
  def receive_user
```

```
  end
end

class User
  def receive_
Edn

class Factory
  Def convert(obj)
    Correct type return it
    Look up factory
    Factory.receive obj
  End

def factory_for
  If factory then return it
  If symbol then get from identity map
  If string then Constantize it
  If hash then create model class, model_klass
  If array then union type
end
```

Important that receeiver is not setter - floppy inassertive model with no sense of its own identity. Like the security line at the airport: horrible inexplicable things, more complicated then it seems like it should, but once past the security line you can g back to being a human again.

Do not make filedls that represent objects and keys — if there's a user fields put it in `user`, its id in user_id, and use virtual accessors to mask the difference.

Good Format #3: Avro (It does everything right)

that there is no essential difference among

```
File Format         Schema          API
RPC (Remote Procedure Call) Definition
JPG                 CREATE TABLE    Twitter API
HTML DTD            db defn.
```

Avro says these are imperfect reflections for the same thing: a method to send data in space and time, to yourself or others. This is a very big idea [^1].

1

1. To the people of the future: this might seem totally obvious. Trust that it is not. There are virtually no shared patterns or idioms across the systems listed here.

Other reasonable choices: tagged net strings and null-delimited documents

Not restartable.

Can't represent a document with a null

Crap format #1: XML

XML is disastrous as a data transport format. It's also widely used in enterprise systems and on the web, and so you will have to learn how to work with it. Wherever possible, implement decoupled code whose only job is to translate the XML into a sane format, and write all downstream code to work with that.

Writing XML

If you have to emit XML for downstream consumption, yet have any control over its structure, follow these best practices:

```
.Well-formed XML
[This has been split across multiple lines, but in production eliminate the whitespace as well]
--------------------
<post>
  <author><name>William Carlos William</name><id>88</id></author>
  <id>12345</id>
  <title>This is Just to Say</title>
  <dtstart>2012-04-26T12:34:56 CST</dtstart>
  <text>I have eaten&#10;the plums&#10;that were in&#10;the icebox&#10;&#10;and which&#10;you were
  <comments>
    <comment><commenter-name>Holly</commenter-name><id>98765</id><text>Your poem made up for it...
  </comments>
  <replies></replies>
</post>
--------------------
```

The example to the side is pretty-printed for clarity; in production, you should eliminate the whitespace as well. Otherwise it does things correctly: * Tags hold only values or other tags (not both). * Values only appear in the contents of tags, and not the tag attributes. * Text contents are fully encoded (`<em>barely</em>`, not `<em>barely</em>`), including whitespace (`&10;` in the post text, not a literal newline). All the XML tags you see belong to the record. * The nesting `comments` tag makes clear that it is an array-of-length-one, in contrast to a singular property like `author`. The `replies` tag is present, representing an empty array, rather than being omitted. * It has a predictable structure, making it easily `grep`'able

If you have to write XML against a specific format, consider using a template language like erubis, moustache or the like. Before I learned this trick, I'd end up with a whole

bunch of over-wrough soupy code just for the purpose of putting open and close tags in the right place. When 90% of the complexity is writing the XML and 10% is stuffing the values in there, you should put the code in the content and not the other way around.

```
.Well-formed XML
[This has been split across multiple lines, but in production eliminate the whitespace as well]
--------------------
<post>
  <author><name><%= record.author.name.to_xml %></name><id><%= record.author.id.to_xml %></id></au
  <id><%=      record.id.to_xml         %></id>
  <title><%=   record.title.to_xml      %></title>
  <dtstart><%= record.created_at.to_xml %></dtstart>
  <text><%=    record.text.to_xml       %></text>
  <comments>
    <% record.comments.each do |comment| -%>
    <comment>
      <commenter-name><%= comment.commenter_name.to_xml %></commenter-name>
      <id><%=   comment.id.to_xml %></id>
      <text><%= comment.text.to_xml %></text>
    </comment>
    <% end -%>
  </comments>
  <replies></replies>
</post>
--------------------
```

This template generates XML with a consistent structure. The `<%= %>` blocks interpolate data — an equals-sign `<%= %>` causes output, a plain `<% %>` block is for control statements. When you're inside a funny-braces block, you're in ruby; everything else is literal content. Control blocks (like the `<% record.comments.each do |comment| %>`) stamp out their contents as you'd expect.

Airing of Grievances

XML is like the English Measurement System — just ubiquitous enough, and just barely useful enough, that it's near-impossible to weed out. Neither, however, is anymore justifiable for use by the professional practitioner. XML is both too extensible and too limited to map smoothly to and from the data structures languages use in practive. In the case that you need to make the case against XML to a colleague, I arm you with the following List of Grievances:

- **Does not preserve simple types**: the only primitive data type is a string; there's no standard way to distinguish an integer, a floating-point number, or a date without external hints.
- **Does not preserve complex types**: You will find data stored with

  — mixed attributes and data:

```
<post date="2012-01-02T12:34:56 CST" author="William Carlos Williams">
<body>I have eaten&#10;the plums&#10;that were in...</body>
</post>
```

- mixed data and text:

```
<post>
<title>This is Just to Say</title>
I have eaten the plums that ... you were probably saving for <span dtstart="2012-04-27T08:00:00
</post>
```

- **Inconsistent cardinality**: In this example, there's no way to distinguish a singular property like `title` from a list-of-length-one like `comment`; simple XML readers will return`{"title":"…", "comment":{"text":"…"}}` when there is one, and `{"title":"...", "comment":[{"text":"..."},{"text":"..."}]}` when there are many.

```
<post>
  <title>This is just to say</title>
  <comment><text>Your poem made up for it</text></comment>
</post>
```

- **Not restartable.**: you can only properly understand an XML file by reading it from beginning to end. CDATA blocks are especially treacherous; they can in principle hold nearly anything, including out-of-band XML.
- **Not unique**: even are multiple ways to represent the same final context. An apostrophe might be represented directly (`'`), hex-encoded (`&#x0027;`), decimal-encoded (`&#39;`), or as an SGML [2] entity (`&apos;`). (You may even find people using SGML entities in the absence of the DTD [3] that is technically required to interpret them.)
- **Complex**: the technical standard for XML is fiendishly complex, and even mature libraries in widespread use still report bugs parsing complex or ill-formed input.

Attributes, CDATA, model boundaries, document text

If you do it, consider emitting not with a serde but with a template engine. Pretty-print fields so can use cmdline tools

2. SGML= Standard Generalized Markup Language, the highly-complex document format that inspired HTML
3. DTD = Document Type Declaration; an over-enthusiastic DTD can make XML mutable to the point of incomprehensibility.

Crap Format #2: N3 triples

Like most Semantic-Web developed technology, N3 is antagonistic to thought and action.

If you must deal with this, pretty-print the fields and ensure delimiters are clean.

Crap Format #3: Flat format

WALKTHROUGH: converting the weather fields.

Flat formats are surprisingly innocuous; it's the contortions they force upon their tender that hurts.

Straightforward to build a regexp. Wukong gives you a flatpack stringifier. Specify a format string as follows:

```
"%4d%3.2f\"%r{([^\"]+)}\""
```

It returns a MatchData object (same as a regexp does).

9999 as null (or other out-of-band): Override the receive_xxx method to knock those out, call super.

To handle the elevation fields, override the receive method:

Note that we call super **first** here , because we want an int to divide; in the previous case, we want to catch 9999 before it goes in for conversion. Wukong has some helpers for unit conversion too.

Web log and Regexpable

WALKTHROUGH: apache web logs of course. - Regexp to tuple. Just capture substructure

Glyphing (string encoding), Unicode,UTF-8

All of the following examples could be ambiguously referred to as "encoding":

- Compression: gz, LZO, Snappy, etc
- Serialization: the actual record container
- Stringifying: conversion to JSON, TSV, etc.
- Glyphing: binary stream (for example UTF8-encoded Unicode) to characters (TODO: make this correct)

- Structured to model[4]

My best advice is

- Never let **anything** into your system unless it is UTF8, UTF-16, or ASCII.
- Either:
  - — Only transmit 7-bit ASCII characters in the range 0x20 (space) to 0x126 (~), along with 0x0a (newline) and 0x09 (tab) but only when used as record (newline) or field (tab) separators. URL encoding, JSON encoding, and HTML entity encoding are all reasonable. HTML entity encoding has the charm of leaving simple international text largely readable: "caf&eacute\;" or "M&oumlaut\;torh&eumlaut\;ad" are more easily scannable than "caf\XX". Be warned that unless you exercise care all three can be ambiguous: &eacute\;, (that in decimal) and (that in hex) are all the same.to make life grep'able, force your converter to emit exactly one string for any given glyph — That is, it will not ship "0x32" for "a", and it will not ship "é" for "\XX"
  - — Use unix-style newlines only.
  - — Even With unique glyph coding, Unicode is still not unique: edge cases involving something something diacritic modifiers.
  - — However complex you think Unicode is, it's slightly more hairy than that.
  - — URL encoding only makes sense when you're shipping urls anyway.
  - — TODO: check those character strings for correctness. Also, that I'm using "glyph" correctly

ICSS

ICSS uses [5]

Schema.org Types

Munging

```
class RawWeatherStation
  field :wban_id
  # ...
```

4. the worst example iis "node": a LAN node configured by a chef node running a Flume logical node handling graph nodes in a Node.js decorator.
5. Every Avro schema file is a valid ICSS schema file, but Avro will not understand all the fields. In particular, Avro has no notion of `is_a` inheritance; ICSS does

```
  field :latitude
  field :longitude
end

class Science::Climatology::WeatherStation < Type::Geo::GovernmentBuilding
  field :wban_id
  field :
end

name:   weatherstation
types:
  name:   raw_weather_station
  fields:
    - name:  latitude
      type:  float
    - name:  longitude
      type:  float
  # ...
```

CHAPTER 16

Filesystem Mojo and `cat` Herding

When working with big data, there are thousands of housekeeping tasks to do. As you will soon discover, moving data around, munging it, transforming it, and other mundane tasks will take up a depressingly inordinate amount of your time. A proper knowledge of some useful tools and tricks will make these tasks doable, and in many cases, easy.

In this chapter we discuss a variety of Unix commandline tools that are essential for shaping, transforming, and munging text. All of these commands are covered elsewhere, and covered more completely, but we've focused in on their applications for big data, specifically their use within Hadoop.

FLAVORISE

By the end of this chapter you should be **FINISH**

If you're already familiar with Unix pipes and chaining commands together, feel free to skip the first few sections and jump straight into the tour of useful commands.

A series of pipes

One of the key aspects of the Unix philosophy is that it is built on simple tools that can be combined in nearly infinite ways to create complex behavior. Command line tools are linked together through *pipes* which take output from one tool and feed it into the input of another. For example, the `cat` command reads a file and outputs its contents. You can pipe this output into another command to perform useful transformations. For example, to select only the first field (delimited by tabs) of the each line in a file you could:

```
cat somefile.txt | cut -f1`
```

The vertical pipe character, |, represents the pipe between the `cat` and `cut` commands. You can chain as many commands as you like, and it's common to construct chains of 5 commands or more.

In addition to redirecting a command's output into another command, you can also redirect output into a file with the > operator. For example:

```
echo 'foobar' > stuff.txt
```

writes the text *foobar* to stuff.txt. stuff.txt is created if it doesn't exist, and is overwritten if it does.

If you'd like to append to a file instead of overwriting it, the >> operator will come in handy. Instead of creating or overwriting the specified file, it creates the file if it doesn't exist, and appends to the file if it does.

As a side note, the Hadoop streaming API is built using pipes. Hadoop sends each record in the map and reduce phases to your map and reduce scripts' stdin. Your scripts print the results of their processing to stdout, which Hadoop collects.

Crossing the streams

Each Unix command has 3 input/output streams: standard input, standard output, and standard error, commonly referred to by the more concise stdin, stdout, and stderr. Commands accept input via stdin and generate output through stdout. When we said that pipes take output from one tool and feed it into the input of another, what we really meant was that pipes feed one command's stdout stream into another command's stdin stream.

The third stream, stderr, is generally used for error messages, progress information, or other text that is not strictly *output*. Stderr is especially useful because it allows you to see messages from a command even if you have redirected the command's stdout. For example, if you wanted to run a command and redirect its output into a file, you could still see any errors generated via stderr. curl, a command used to make network requests, #FINISH

```
* CURL COMMAND*
```

It's occasionally useful to be able to redirect these streams independently or into each other. For example, if you're running a command and want to log its output as well as any errors generated, you should redirect stderr into stdout and then direct stdout to a file:

```
*EXAMPLE*
```

Alternatively, you could redirect stderr and stdout into separate files:

```
*EXAMPLE*
```

You might also want to suppress stderr if the command you're using gets too chatty. You can do that by redirecting stderr to /dev/null, which is a special file that discards everything you hand it.

Now that you understand the basics of pipes and output redirection, lets get on to the fun part - munging data!

cat and echo

`cat` reads the content of a file and prints it to stdout. It can accept multiple files, like so, and will print them in order:

```
cat foo.txt bar.txt bat.txt
```

`cat` is generally used to examine the contents of a file or as the start of a chain of commands:

```
cat foo.txt | sort | uniq > bar.txt
```

In addition to examining and piping around files, `cat` is also useful as an *identity mapper*, a mapper which does nothing. If your data already has a key that you would like to group on, you can specify `cat` as your mapper and each record will pass untouched through the map phase to the sort phase. Then, the sort and shuffle will group all records with the same key at the proper reducer, where you can perform further manipulations.

`echo` is very similar to `cat` except it prints the supplied text to stdout. For example:

```
echo foo bar baz bat > foo.txt
```

will result in foo.txt holding *foo bar baz bat*, followed by a newline. If you don't want the newline you can give echo the -n option.

Filtering

cut

The `cut` command allows you to cut certain pieces from each line, leaving only the interesting bits. The -f option means *keep only these fields*, and takes a comma-delimited list of numbers and ranges. So, to select the first 3 and 5th fields of a tsv file you could use:

```
cat somefile.txt | cut -f 1-3,5`
```

Watch out - the field numbering is one-indexed. By default cut assumes that fields are tab-delimited, but delimiters are configurable with the `-d` option.

This is especially useful if you have tsv output on the hdfs and want to filter it down to only a handful of fields. You can create a hadoop streaming job to do this like so:

```
wu-mapred --mapper='cut -f 1-3,5'
```

`cut` is great if you know the indices of the columns you want to keep, but if you data is schema-less or nearly so (like unstructured text), things get slightly more complicated. For example, if you want to select the last field from all of your records, but the field length of your records vary, you can combine cut with the `rev` command, which reverses text:

```
cat foobar.txt | rev | cut -1 | rev`
```

This reverses each line, selects the first field in the reversed line (which is really the last field), and then reverses the text again before outputting it.

`cut` also has a -c (for *character*) option that allows you to select ranges of characters. This is useful for quickly verifying the output of a job with long lines. For example, in the Regional Flavor exploration, many of the jobs output wordbags which are just giant JSON blobs, one line of which would overflow your entire terminal. If you want to quickly verify that the output looks sane, you could use:

```
wu-cat /data/results/wikipedia/wordbags.tsv | cut -c 1-100
```

Character encodings

Cut's -c option, as well as many Unix text manipulation tools require a little forethought when working with different character encodings because each encoding can use a different numbers of bits per character. If `cut` thinks that it is reading ASCII (7 bits per character) when it is really reading UTF-8 (variable number of bytes per character), it will split characters and produce meaningless gibberish. Our recommendation is to get your data into UTF-8 as soon as possible and keep it that way, but the fact of the matter is that sometimes you have to deal with other encodings.

Unix's solution to this problem is the LC_* environment variables. LC stands for *locale*, and lets you specify your preferred language and character encoding for various types of data.

`LC_CTYPE` (locale character type) sets the default character encoding used systemwide. In absence of LC_CTYPE, `LANG` is used as the default, and `LC_ALL` can be used to override all other locale settings. If you're not sure whether your locale settings are having their intended effect, check the man page of the tool you are using and make sure that it obeys the LC variables.

You can view your current locale settings with the `locale` command. Operating systems differ on how they represent languages and character encodings, but on my machine `en_US.UTF-8` represents English, encoded in UTF-8.

Remember that if you're using these commands as Hadoop mappers or Reducers, you must set these environment variables across your entire cluster, or set them at the top of your script.

head and tail

While `cut` is used to select columns of output, head and tail are used to select lines of output. head selects lines at the beginning of its input while tail selects lines at the end. For example, to view only the first 10 lines of a file, you could use head like so:

```
head -10 foobar.txt
```

`head` is especially useful for sanity-checking the output of a Hadoop job without overflowing your terminal. `head` and cut make a killer combination:

```
wu-cat /data/results/foobar | head -10 | cut -c 1-100
```

`tail` works almost identically to `head`. Viewing the last ten lines of a file is easy:

```
tail -10 foobar.txt
```

`tail` also lets you specify the selection in relation to the beginning of the file with the + operator. So, to select every line from the 10th line on:

```
tail +10 foobar.txt
```

What if you just finished uploading 5,000 small files to the HDFS and realized that you left a header on every one of them? No worries, just use `tail` as a mapper to remove the header:

```
wu-mapred --mapper='tail +2'`
```

This outputs every line but the first one.

`tail` is also useful for watching files as they are written to. For example, if you have a log file that you want to watch for errors or information, you can *tail* it with the -f option:

```
tail -f yourlogs.log
```

This outputs the end of the log to your terminal and waits for new content, updating the output as more is written to yourlogs.log.

grep

`grep` is a tool for finding patterns in text. You can give it a word, and it will diligently search its input, printing only the lines that contain that word:

```
GREP EXAMPLE
```

`grep` has a many options, and accepts regular expressions as well as words and word sequences:

```
ANOTHER EXAMPLE
```

The -i option is very useful to make grep ignore case:

```
EXAMPLE
```

As is the -z option, which decompresses g-zipped text before grepping through it. This can be tremendously useful if you keep files on your HDFS in a compressed form to save space.

When using `grep` in Hadoop jobs, beware its non-standard exit statuses. `grep` returns a 0 if it finds matching lines, a 1 if it doesn't find any matching lines, and a number greater than 1 if there was an error. Because Hadoop interprets any exit code greater than 0 as an error, any Hadoop job that doesn't find any matching lines will be considered *failed* by Hadoop, which will result in Hadoop re-trying those jobs without success. To fix this, we have to swallow `grep`'s exit status like so:

```
(grep foobar || true)
```

This ensures that Hadoop doesn't erroneously kill your jobs.

GOOD TITLE HERE

sort

As you might expect, `sort` sorts lines. By default it sorts alphabetically, considering the whole line:

```
EXAMPLE
```

You can also tell it to sort numerically with the -n option, but -n only sorts integers properly. To sort decimals and numbers in scientific notation properly, use the -g option:

```
EXAMPLE
```

You can reverse the sort order with -r:

```
EXAMPLE
```

You can also specify a column to sort on with the -k option:

```
EXAMPLE
```

By default the column delimiter is a non-blank to blank transition, so any content character followed by a whitespace character (tab, space, etc…) is treated as a column. This can be tricky if your data is tab delimited, but contains spaces within columns. For example, if you were trying to sort some tab-delimited data containing movie titles, you would have to tell sort to use tab as the delimiter. If you try the obvious solution, you might be disappointed with the result:

```
sort -t"\t"
sort: multi-character tab `\\t'
```

Instead we have to somehow give the -t option a literal tab. The easiest way to do this is:

```
sort -t$'\t'
```

`$'<string>'` is a special directive that tells your shell to expand `<string>` into its equivalent literal. You can do the same with other control characters, including `\n`, `\r`, etc…

Another useful way of doing this is by inserting a literal tab manually:

```
sort -t'	'
```

To insert the tab literal between the single quotes, type `CTRL-V` and then `Tab`.

If you find your sort command is taking a long time, try increasing its sort buffer size with the `--buffer` command. This can make things go a lot faster:

```
example
```

TALK ABOUT SORT'S USEFULNESS IN BIG DATA

uniq

`uniq` is used for working with with duplicate lines - you can count them, remove them, look for them, among other things. For example, here is how you would find the number of oscars each actor has in a list of annual oscar winners:

```
example
```

Note the -c option, which prepends the output with a count of the number of duplicates. Also note that we sort the list before piping it into uniq - input to uniq must always be sorted or you will get erroneous results.

You can also filter out duplicates with the -u option:

```
example
```

And only print duplicates with the -d option:

```
example
```

- TALK ABOUT USEFULNESS, EXAMPLES*

join

TBD - do we even want to talk about this?

Summarizing

wc

`wc` is a utility for counting words, lines, and characters in text. Without options, it searches its input and outputs the number of lines, words, and bytes, in that order:

```
EXAMPLE
```

`wc` will also print out the number of characters, as defined by the LC_CTYPE environment variable:

```
EXAMPLE
```

We can use wc as a mapper to count the total number of words in all of our files on the HDFS:

```
EXAMPLE
```

md5sum and sha1sum

- Flip ???*

CHAPTER 17

Conceptual Model for Data Analysis

- lambda architecture
- unified conceptual model
- patterns for analytics
- patterns for lambda architecture

…

CHAPTER 18

Machine Learning without Grad School

CHAPTER 19

Java Api

When to use the Hadoop Java API

Don't.

How to use the Hadoop Java API

The Java API provides direct access but requires you to write the entirety of your program from boilerplate

Decompose your problem into small isolable transformations. Implement each as a Pig Load/StoreFunc or UDF (User-Defined Function) making calls to the Hadoop API.[^1]

The Skeleton of a Hadoop Java API program

I'll trust that for very good reasons — to interface with an outside system, performance, a powerful library with a Java-only API — Java is the best choice to implement

```
[ Your breathtakingly elegant, stunningly performant solution to a novel problem ]
```

When this happens around the office, we sing this little dirge[^2]:

```
Life, sometimes, is Russian Novel. Is having unhappy marriage and much snow and little vodka.
But when Russian Novel it is short, then quickly we finish and again is Sweet Valley High.
```

What we **don't** do is write a pure Hadoop-API Java program. In practice, those look like this:

```
HORRIBLE BOILERPLATE TO DO A CRAPPY BUT SERVICABLE JOB AT PARSING PARAMS
HORRIBLE BOILERPLATE TO DO A CRAPPY BUT SERVICABLE JOB AT READING FILES
COBBLED-TOGETHER CODE TO DESERIALIZE THE FILE, HANDLE SPLITS, ETC

[ Your breathtakingly elegant, stunningly performant solution to a novel problem ]
```

```
COBBLED-TOGETHER CODE THAT KINDA DOES WHAT PIG'S FLATTEN COMMAND DOES
COBBLED-TOGETHER CODE THAT KINDA DOES WHAT PIG'S CROSS COMMAND DOES
A SIMPLE COMBINER COPIED FROM TOM WHITE'S BOOK
1000 LINES OF CODE TO DO WHAT RUBY COULD IN THREE LINES OF CODE
HORRIBLE BOILERPLATE TO DO A CRAPPY BUT SERVICABLE JOB AT WRITING FILES
UGLY BUT NECESSARY CODE TO GLUE THIS TO THE REST OF THE ECOSYSTEM
```

The surrounding code is ugly and boring; it will take more time, produce more bugs, and carry a higher maintenance burden than the important stuff. More importantly, the high-level framework provides an implementation far better than it's worth your time to recreate.[^3]

Instead, we write

```
A SKELETON FOR A PIG UDF DEFINITION
[ Your breathtakingly elegant, stunningly performant solution to a novel problem ]

A PIG SCRIPT
```

CHAPTER 20

Advanced Pig

Optimizing Hadoop Dataflows

The guidelines in this section start by repeating things your good common sense surely already knew — you didn't buy this book to have people tell you it's inefficient to process more data than you need to. It's not always obvious, however, how to get Hadoop to respect your good common sense and, if anything that follows seems obvious to you, well, it's included here because your authors have most likely learned it the hard way.

So, don't process more data than you have to. In Pig, put filters before joins or other structural operations. If you're doing a filter, you probably have a good sense of how much data should be allowed through. Check the Job Tracker Console and if the ratio of data read to Mapper output data size does not line up with what you expect, dig deeper. Also, project out fields you don't need for downstream operations; surprisingly, Pig does not do this for you. Keep in mind that when you do a Group, the key fields appear in full in both the new Group field and in every one of the grouped tuples. It is a good habit to follow every Group with a FOREACH.

If you only need a small fraction of records from a dataset, extract them as early as possible. If it is all the same to you, a LIMIT operation (taking the first N rows) is more efficient than a SAMPLE or FILTER; if you can get everything you need from one or a few input files, that's even more efficient. As we harped on throughout the book, when you're developing a job, it's almost always worth it to start by extracting a faithful, smaller sub-universe to work with.

Process more data if you have to if it makes your code more readable. There is some benefit to having uniform schema, especially if you're working with TSV files, where the Mapping from slot order to name is not intrinsic. If leaving in a couple extraneous fields would add five minutes to a job's production runtime but subtract five lines from the source, we prefer the inefficient script.

In general, wasting CPU to save network or disk bandwidth is a good idea. If you grew up watching a 386 try to produce a ZIP file, it probably seems counterintuitive that storing your data in compressed files not only saves disk space but also speeds up processing time. However, most Hadoop jobs are overwhelmingly throughput bound, so spending the extra CPU cycles to compress and decompress data is more than justified by the overall efficiency of streaming less data off the disk or network. The section on Hadoop Internals (TODO: REF) explains how to enable compression of data between Mapper and Reducer (which you should always do) and how to read or write compressed data (which has tradeoffs you need to understand). In the case of intermediate checkpoints within a multi-stage workflow, it almost always makes sense to use a light compression format such as LZO or Snappy. In Pig, if you set the `pig.tmpfilecompression` and `pig.tmpfilecompression.codec` configuration variables appropriately, it will do that for you.

There are a few other cases where you should invoke this rule. If you have a large or variable text string that you only need to compare for uniqueness — e.g., using URLs as keys — it is worth using a string digest function to shorten its length, as described in the chapter on Sketch Functions (TODO: REF).

Regular expressions are much faster than you'd think, especially in Ruby. If you only need a small part of a string and it does not cost you in readability, it might be worth slicing out only the interesting part.

Use types efficiently. Always use a schema in Pig; no exceptions. In case you're wondering, it makes your script more efficient and catches errors early, but most of all, it shows respect to your colleagues or future self.

There are a couple Pig-specific tradeoffs to highlight. Wherever possible, make your UDFs algebraic or at least Accumulator-like — as described in the section on UDFs (TODO: REF). If you use two FOREACH statements consecutively, Pig is often able to merge them, but not always. If you see Pig introduce an extra Mapside-only job where you didn't think one was necessary, it has probably failed to combine them. Always start with the more readable code, then decide if the problem is worth solving. Most importantly, be aware of Pig's specialized JOINs; these are important enough that they get their whole section below.

As you've seen, Pig is extremely sugar-free; more or less every structural operation corresponds to a unique Map/Reduce plan. In principle, a JOIN is simply a Cogroup with a FLATTEN and a DISTINCT is a Cogroup with a projection of just the GROUP key. Pig offers those more specific Operators because it is able to do them more efficiently. Watch for cases where you have unwittingly spelled these out explicitly.

Always remove records with a NULL Group key *before* the JOIN; those records will never appear in the output of a JOIN statement, yet they are not eliminated until after they have been sent across the network. Even worse, since all these records share all the

same (worthless) key, they are all sent to the same Reducer, almost certainly causing a hotspot.

If you are processing a lot of small files, Pig offers the ability to process many at once per Mapper, an extremely important optimization. Set the `pig.splitCombination` and `pig.maxCombinedSplitSize` options; if you're writing a custom loader, spend the extra effort to make it compatible with this optimization.

Do not use less or more parallelism than reasonable. We have talked repeatedly throughout the book about the dangers of hotspots — a few Reducers laboring to swallow the bulk of the dataset while its many comrades clock out early. Sometimes, however, your job's configuration might unwittingly recommend to Hadoop that it only use one or a too-few number of Reducers. In this case, the Job Tracker would show only a few heavyweight Reduce tasks running, the other Reduce slots are sitting idle because nothing has been asked of them. Set the number of Reducers in Pig using the 'PARALLEL' directive, and in Wukong, using the '--REDUCE_TASKS=N' (TODO: check spelling).

It can also be wasteful to have too many Reducers. If your job has many Reducers uniformly processing only a few kb of records, the large fixed costs of launching and accounting for those attempts justify using the parallelism settings to limit the number of Reducers.

Efficient JOINs in Pig

Pig has a number of specialized JOINs that, used in their appropriate circumstances, bring enough performance improvements to organize a cult around. (TODO: make funny).

Specialized Pig Join #1: The `REPLICATED JOIN`

If you are joining a large dataset with a small-enough one, Pig can often execute the operation using a Mapper-Only job, eliminating the costly startup time and network transfer of a Reduce step. This is commonplace enough and the performance impact large enough that it is always worth considering whether this type of JOIN is applicable.

Imagine visiting the opera while the United Nations is in town. The smart theater owner will prepare librettos in, say, a dozen languages, enough to delight the many thousands of attendees. A bad way to distribute them would be to arrange kiosks, one per language, throughout the lobby. Every aficionado would first have to fight their way through the crowd to find the appropriate kiosk, then navigate across the theater to find their seats. Our theater owner, being smart, instead handles the distribution as follows: Ushers stand at every entrance, armed with stacks of librettos; at every entrance, all the languages are represented. This means that, as each attendee files in, they simply select the appropriate one from what is on hand, then head to their seat without delay.

A Mapper-Only JOIN works analogously. Every Mapper reads the small dataset into a lookup table — a hash map keyed by the JOIN key (this is why you will also see it referred to as a HashMap JOIN). Every Mapper loads the contents of the smaller dataset in full into its own local lookup table (which is why it is also known as a Replicated JOIN). The minor cost of replicating that dataset to every single Mapper is often a huge improvement in processing speed by eliminating the entire Reduce stage. The constraint, however, is that the smaller dataset must fit entirely in RAM. The usher's task is manageable when there is one type of libretto for each of a dozen languages but would be unmanageable if there were one type of libretto for each of several thousand home towns.

How much is too much? Watch for excessive GC activity. (TODO: Pig probably has a warning too - find out what it is). Within the limits of available RAM, you can use fewer Mappers with more available RAM; the Hadoop tuning chapter (TODO: REF) shows you how. Don't be too aggressive, though; datasets have a habit of growing over time and you would hate to spend Thanksgiving day reworking the jobs that process retail sales data because you realized they would not stand up to the Black Friday rush.

There is a general principle here: It is obvious there is a class of problems which only crop up past a certain threshold of data. What may not be obvious, until you've learned it the hard way, is that the external circumstances most likely to produce that flood of extra data are also the circumstances that leave you least able to address the problem.

Specialized Pig Join #2: The `MERGE JOIN`

A JOIN of two datasets, each in strict total order by the JOIN key, can also be done using Mapper-Only by simply doing a modified Merge sort. You must ensure not only that the files are in sort order but that the lexicographic order of the file names match the order in which its parts should be read. If you do so, Pig can proceed as follows: It does a first pass to sample each file from the right-hand dataset to learn the distribution of keys throughout the files. The second stage performs the actual JOIN. Each Mapper reads from two streams: its assigned split within the left-hand dataset and the appropriate sections of the right-hand dataset. The Mapper's job is then very simple; it grabs a group of records from the right-hand stream and a group of records from the left-hand stream and compares their keys. If they match, they are joined. If they do not match, it reads from the stream with the too-low key until it either produces the matching group or sails past it, in which case it similarly reads from the other stream.

As we've discussed a few times, reading data in straight streams like this lets the underlying system supply data at the fastest possible rate. What's more, the first pass indexing scheme means most tasks will be "Map-local" — run on a machine whose data node hosts a copy of that block. In all, you require a short Mapper-Only task to sample the right-hand dataset and the network throughput cost that is 'O(N)' in the size of the second dataset. The constraint is, of course, that this only works with total-ordered data on the same key. For a "Gold" dataset — one that you expect to use as source data for a

number of future jobs — we typically spend the time to do a last pass total sort of the dataset against the most likely JOIN key. It is a nice convenience for future users of the dataset, helps in sanity checking and improves the odds that you will be able to use the more efficient MERGE/JOIN.

Exercises

1. Quoting Pig docs: > "You will also see better performance if the data in the left table is partitioned evenly across part files (no significant skew and each part file contains at least one full block of data)."

   ```
   Why is this?
   ```

2. Each of the following snippets goes against the Pig documentation's recommendations in one clear way.

   - Rewrite it according to best practices
   - compare the run time of your improved script against the bad version shown here.

     ```
     things like this from http://pig.apache.org/docs/r0.9.2/perf.html --
     ```

     a. (fails to use a map-side join)

     b. (join large on small, when it should join small on large)

     c. (many `FOREACH`es instead of one expanded-form `FOREACH`)

     d. (expensive operation before `LIMIT`)

For each use weather data on weather stations.

CHAPTER 21

Hadoop Internals

For 16 chapters now, we've been using Hadoop and Storm+Trident from the outside. The biggest key to writing efficient dataflows is to understand the interface and fundamental patterns of use, not the particulars of how these framework executes the dataflow. However, even the strongest abstractions pushed far enough can leak, and so at some point, it's important to understand these internal details. These next few chapters concentrate on equipping you to understand your jobs' performance and practical tips for improving it; if you're looking for more, by far, the best coverage of this material is found in (TODO: Add links Tom White's Hadoop: The Definitive Guide and Eric Sammer's Hadoop Operations).

Let's first focus on the internals of Hadoop.

HDFS (NameNode and DataNode)

It's time to learn how the HDFS stores your data; how the Map/Reduce framework launches and coordinates job attempts; and what happens within the framework as your Map/Reduce process executes.

The HDFS provides three remarkable guarantees: durability (your data is never lost or corrupted), availability (you can always access it) and efficiency (you can consume the data at high rate especially from Map/Reduce jobs and other clients). The center of action, as you might guess, is the NameNode. The NameNode is a permanently running daemon process that tracks the location of every block on the network of DataNodes, along with its name and other essential metadata. (If you're familiar with the File Allocation Table (FAT) of a traditional file system, it's like a more active version of that. FOOTNOTE: [If you're not familiar with what an FAT is, then it's like the system you're reading about but for a file system.])

(NOTE: check something … appending)

When a client wishes to create a file, it contacts the NameNode with the desired path and high-level metadata. The NameNode records that information in an internal table and identifies the DataNodes that will hold the data. The NameNode then replies with that set of DataNodes, identifying one of them as the initial point of contact. (When we say "client", that's anything accessing the NameNode, whether it's a Map/Reduce job, one of the Hadoop filesystem commands or any other program.) The file is now exclusively available to the client for writing but will remain invisible to anybody else until the write has concluded (TODO: Is it when the first block completes or when the initial write completes?).

Within the client's request, it may independently prescribe a replication factor, file permissions, and block size _ (TODO: fill in)

The client now connects to the indicated DataNode and begins sending data. At the point you've written a full block's worth of data, the DataNode transparently finalizes that block and begins another (TODO: check that it's the DataNode that does this). As it finishes each block or at the end of the file, it independently prepares a checksum of that block, radioing it back to the NameNode and begins replicating its contents to the other DataNodes. (TODO: Is there an essential endoffile ritual?) This is all transparent to the client, who is able to send data as fast as it can cram it through the network pipe.

Once you've created a file, its blocks are immutable — as opposed to a traditional file system, there is no mechanism for modifying its internal contents. This is not a limitation; it's a feature. Making the file system immutable not only radically simplifies its implementation, it makes the system more predictable operationally and simplifies client access. For example, you can have multiple jobs and clients access the same file knowing that a client in California hasn't modified a block being read in Tokyo (or even worse, simultaneously modified by someone in Berlin). (TODO: When does append become a thing?)

The end of the file means the end of its data but not the end of the story. At all times, the DataNode periodically reads a subset of its blocks to find their checksums and sends a "heartbeat" back to the DataNode with the (hopefully) happy news. (TODO: fill in). There are several reasons a NameNode will begin replicating a block. If a DataNode's heartbeat reports an incorrect block checksum, the NameNode will remove that DataNode from the list of replica holders for that block, triggering its replication from one of the remaining DataNodes from that block. If the NameNode has not received a heartbeat from a given DataNode within the configured timeout, it will begin replicating all of that DataNode's blocks; if that DataNode comes back online, the NameNode calmly welcomes it back into the cluster, cancelling replication of the valid blocks that DataNode holds. Furthermore, if the amount of data on the most populated and least populated DataNodes becomes larger than a certain threshold or the replication factor for a file is increased, it will rebalance; you can optionally trigger one earlier using the `hadoop balancer` command.

However it's triggered, there is no real magic; one of the valid replica-holder DataNodes sends the block contents to the new replica holder, which heartbeats back the block once received. (TODO: Check details)

As you can see, the NameNode and its metadata are at the heart of every HDFS story. This is why the new HighAvailability (HA) NameNode feature in recent versions is so important and should be used in any production installation. It's even more important, as well, to protect and backup the NameNode's metadata, which, unfortunately, many people don't know to do. (TODO: Insert notes on NameNode metadata hygiene previously written).

The NameNode selects the recipient DataNodes with some intelligence. Information travels faster among machines on the same switch, switches within the same data center and so forth. However, if a switch fails, all its machines are unavailable and if a data center fails, all its switches are unavailable and so forth. When you configure Hadoop, there is a setting that allows you to tell it about your network hierarchy. If you do so, the NameNode will ensure the first replica lies within the most distant part of the hierarchy — durability is important above all else. All further replicas will be stored within the same rack — providing the most efficient replication. (TODO: Mention the phrase "rack aware.") As permitted within that scheme, it also tries to ensure that the cluster is balanced — preferring DataNodes with less data under management. (TODO: Check that it's amount of data not percent free)

(TODO: Does it make contact on every block or at the start of a file?)

The most important feature of the HDFS is that it is highly resilient against failure of its underlying components. Any system achieves resiliency through four mechanisms: Act to prevent failures; insulate against failure through redundancy; isolate failure within independent fault zones; and lastly, detect and remediate failures rapidly. (FOOTNOTE: This list is due to James Hamilton, TODO: link whose blocks and papers are essential reading). The HDFS is largely insulated from failure by using file system-based access (it does not go behind the back of the operating system), by being open source (ensuring code is reviewed by thousands of eyes and run at extremely high scale by thousands of users), and so forth. Failure above the hardware level is virtually unheard of. The redundancy is provided by replicating the data across multiple DataNodes and, in recent versions, using the Zookeeper-backed HighAvailability NameNode implementation.

The rack awareness, described above, isolates failure using your defined network hierarchy, and at the semantic level, independently for each HDFS block. Lastly, the heartbeat and checksum mechanisms along with its active replication and monitoring hooks allow it and its Operators to detect intermediate faults.

S3 File System

The Amazon EC2 Cloud provides an important alternative to the HDFS, its S3 object store. S3 transparently provides multi-region replication far exceeding even HDFS' at exceptionally low cost (at time of writing, about $80 per terabyte per month, and decreasing at petabyte and higher scale). What's more, its archival datastore solution, Glacier, will hold rarely-accessed data at one-tenth that price and even higher durability. (FOOTNOTE: The quoted durability figure puts the engineering risk below, say, the risk of violent overthrow of our government). For machines in the Amazon Cloud with a provisioned connection, the throughput to and from S3 is quite acceptable for Map/Reduce use.

Hadoop has a built-in facade for the S3 file system, and you can do more or less all the things you do with an HDFS: list, put and get files; run Map/Reduce jobs to and from any combination of HDFS and S3; read and create files transparently using Hadoop's standard file system API. There are actually two facades. The `s3hdfs` facade (confusingly labeled as plain `s3` by Hadoop but we will refer to it here as `s3hdfs`) stores blocks in individual files using the same checksum format as on a DataNode and stores the Name Node-like metadata separately in a reserved area. The `s3n` facade, instead, stores a file as it appears to the Hadoop client, entirely in an `s3` object with a corresponding path. When you visit S3 using Amazon's console or any other standard S3 client, you'll see a file called `/my/data.txt` as an object called `datadoc.txt` in `MyContainer` and its contents are immediately available to any such client; that file, written `s3hdfs` will appear in objects named for 64-bit identifiers like `0DA37f...` and with uninterpretable contents. However, `s3n` cannot store an individual file larger than 5 terabytes. The `s3hdfs` blocks minorly improve Map/Reduce efficiency and can store files of arbitrary size. All in all, we prefer the `s3n` facade; the efficiency improvement for the robots does not make up for the impact on convenience on the humans using the system and that it's a best-practice to not make individual files larger than 1 terabyte any way.

The universal availability of client libraries makes S3 a great hand-off point to other systems or other people looking to use the data. We typically use a combination of S3, HDFS and Glacier in practice. Gold data — anything one project produces that another might use — is stored on S3. In production runs, jobs read their initial data from S3 and write their final data to S3 but use an HDFS local to all its compute nodes for any intermediate checkpoints.

When developing a job, we run an initial `distcp` from S3 onto the HDFS and do all further development using the cluster-local HDFS. The cluster-local HDFS provides better (but not earth-shakingly better) Map/Reduce performance. It is, however, noticeably faster in interactive use (file system commands, launching jobs, etc). Applying the "robots are cheap, humans are important" rule easily justifies the maintenance of the cluster-local HDFS.

If you use a cluster-local HDFS in the way described, that is, it holds no gold data, only development and checkpoint artifacts, ___ (TODO: fill in). Provision your HDFS to use EBS volumes, not the local (ephemeral) ones. EBS volumes surprisingly offer the same or better throughput as local ones and allow you to snapshot a volume in use, or even kill all the compute instances attached to those volumes then reattach them to a later incarnation of the cluster. (FOOTNOTE: This does require careful coordination. Our open-source Iron-Fan framework has all the code required to do so.) Since the EBS volumes have significant internal redundancy, it then becomes safe to run a replication factor of 2 or even 1. For many jobs, the portion of the commit stage waiting for all DataNodes to acknowledge replication can become a sizable portion of the time it takes a Map/Reduce stage to complete. Do this only if you're an amateur with low stakes or a professional whose colleagues embrace these tradeoffs; nobody ever got fired for using a replication factor of 3.

As your S3 usage grows --- certainly if you find you have more than, say, a dozen terabytes of data not in monthly use — it's worth marking that data for storage in Glacier, not S3 (you can only do this, of course, if you're using the `s3n` facade). There's a charge for migrating data and, of course, your time is valuable, but the savings can be enormous.

Hadoop Job Execution Internals

- **Hadoop Job Execution**
- Lifecycle of a job at the client level including figuring out where all the source data is; figuring out how to split it; sending the code to the JobTracker, then tracking it to completion.
- How the JobTracker and TaskTracker cooperate to run your job, including: The distinction between Job, Task and Attempt., how each TaskTracker obtains its Attempts, and dispatches progress and metrics back to the JobTracker, how Attempts are scheduled, including what happens when an Attempt fails and speculative execution, __, Split.
- How TaskTracker child and Datanode cooperate to execute an Attempt, including; what a child process is, making clear the distinction between TaskTracker and child process.
- Briefly, how the Hadoop Streaming child process works.

Map-Reduce Internals

- How the mapper and Datanode handle record splitting and how and when the partial records are dispatched.

- The mapper sort buffer and spilling to disk (maybe here or maybe later, the I/O.record.percent).
- Briefly note that data is not sent from mapper-to-reducer using HDFS and so you should pay attention to where you put the Map-Reduce scratch space and how stupid it is about handling an overflow volume.
- Briefly, that combiners are a thing.
- Briefly, how records are partitioned to reducers and that custom partitioners are a thing.
- How the Reducer accepts and tracks its mapper outputs.
- Details of the merge/sort (shuffle and sort), including the relevant buffers and flush policies and why it can skip the last merge phase.
- (NOTE: Secondary sort and so forth will have been described earlier.)
- Delivery of output data to the HDFS and commit whether from mapper or reducer.
- Highlight the fragmentation problem with map-only jobs.
- Where memory is used, in particular, mapper-sort buffers, both kinds of reducer-merge buffers, application internal buffers.
- When using EBS volumes, beware of the commit & replication factor

CHAPTER 22

Hadoop Tuning

One of the wonderful and terrible things about Hadoop (or anything else at Big Data scale) is that there are very few boundary cases for performance optimization. If your dataflow has the wrong shape, it is typically so catastrophically inefficient as to be unworkable. Otherwise, Hadoop's scalability makes the price of simply throwing more hardware at the problem competitive with investing to optimize it, especially for exploratory analytics. That's why the repeated recommendation of this book is: Get the algorithm right, get the contextability right and size your cluster to your job.

When you begin to productionize the results of all your clever work, it becomes valuable to look for these 30-percent, 10-percent improvements. Debug loop time is also important, though, so it is useful to get a feel for when and why early optimization is justified.

The first part of this chapter discusses Tuning for the Wise and Lazy — showing you how to answer the question, "Is my job slow and should I do anything about it?" Next, we will discuss Tuning for the Brave and Foolish, diving into the details of Hadoop's maddeningly numerous, often twitchy, configuration knobs. There are low-level setting changes that can dramatically improve runtime; we will show you how to recognize them and how to determine what per-job overrides to apply. Lastly, we will discuss a formal approach for diagnosing the health and performance of your Hadoop cluster and your Hadoop jobs to help you confidently and efficiently identify the source of deeper flaws or bottlenecks.

Chimpanzee and Elephant: A Day at Work

Each day, the chimpanzee's foreman, a gruff silverback named J.T., hands each chimp the day's translation manual and a passage to translate as they clock in. Throughout the day, he also coordinates assigning each block of pages to chimps as they signal the need for a fresh assignment.

Some passages are harder than others, so it's important that any elephant can deliver page blocks to any chimpanzee — otherwise you'd have some chimps goofing off while others are stuck translating *King Lear* into Kinyarwanda. On the other hand, sending page blocks around arbitrarily will clog the hallways and exhaust the elephants.

The elephants' chief librarian, Nanette, employs several tricks to avoid this congestion.

Since each chimpanzee typically shares a cubicle with an elephant, it's most convenient to hand a new page block across the desk rather then carry it down the hall. J.T. assigns tasks accordingly, using a manifest of page blocks he requests from Nanette. Together, they're able to make most tasks be "local".

Second, the page blocks of each play are distributed all around the office, not stored in one book together. One elephant might have pages from Act I of *Hamlet*, Act II of *The Tempest*, and the first four scenes of *Troilus and Cressida* [1]. Also, there are multiple *replicas* (typically three) of each book collectively on hand. So even if a chimp falls behind, JT can depend that some other colleague will have a cubicle-local replica. (There's another benefit to having multiple copies: it ensures there's always a copy available. If one elephant is absent for the day, leaving her desk locked, Nanette will direct someone to make a xerox copy from either of the two other replicas.)

Nanette and J.T. exercise a bunch more savvy optimizations (like handing out the longest passages first, or having folks who finish early pitch in so everyone can go home at the same time, and more). There's no better demonstration of power through simplicity.

Brief Anatomy of a Hadoop Job

We'll go into much more detail in (TODO: ref), but here are the essentials of what you just performed.

Copying files to the HDFS

When you ran the `hadoop fs -mkdir` command, the Namenode (Nanette's Hadoop counterpart) simply made a notation in its directory: no data was stored. If you're familiar with the term, think of the namenode as a *File Allocation Table (FAT)* for the HDFS.

When you run `hadoop fs -put ...`, the putter process does the following for each file:

1. Contacts the namenode to create the file. This also just makes a note of the file; the namenode doesn't ever have actual data pass through it.

1. Does that sound complicated? It is — Nanette is able to keep track of all those blocks, but if she calls in sick, nobody can get anything done. You do NOT want Nanette to call in sick.

2. Instead, the putter process asks the namenode to allocate a new data block. The namenode designates a set of datanodes (typically three), along with a permanently-unique block ID.
3. The putter process transfers the file over the network to the first data node in the set; that datanode transfers its contents to the next one, and so forth. The putter doesn't consider its job done until a full set of replicas have acknowledged successful receipt.
4. As soon as each HDFS block fills, even if it is mid-record, it is closed; steps 2 and 3 are repeated for the next block.

Running on the cluster

Now let's look at what happens when you run your job.

(TODO: verify this is true in detail. @esammer?)

- *Runner*: The program you launch sends the job and its assets (code files, etc) to the jobtracker. The jobtracker hands a `job_id` back (something like `job_201204010203_0002` — the datetime the jobtracker started and the count of jobs launched so far); you'll use this to monitor and if necessary kill the job.
- *Jobtracker*: As tasktrackers "heartbeat" in, the jobtracker hands them a set of 'task's — the code to run and the data segment to process (the "split", typically an HDFS block).
- *Tasktracker*: each tasktracker launches a set of *mapper child processes*, each one an *attempt* of the tasks it received. (TODO verify:) It periodically reassures the jobtracker with progress and in-app metrics.
- *Jobtracker*: the Jobtracker continually updates the job progress and app metrics. As each tasktracker reports a complete attempt, it receives a new one from the jobtracker.
- *Tasktracker*: after some progress, the tasktrackers also fire off a set of reducer attempts, similar to the mapper step.
- *Runner*: stays alive, reporting progress, for the full duration of the job. As soon as the job_id is delivered, though, the Hadoop job itself doesn't depend on the runner — even if you stop the process or disconnect your terminal the job will continue to run.

Please keep in mind that the tasktracker does *not* run your code directly — it forks a separate process in a separate JVM with its own memory demands. The tasktracker rarely needs more than a few hundred megabytes of heap, and you should not see it consuming significant I/O or CPU.

Chimpanzee and Elephant: Splits

I've danced around a minor but important detail that the workers take care of. For the Chimpanzees, books are chopped up into set numbers of pages — but the chimps translate *sentences*, not pages, and a page block boundary might happen mid-sentence. //// Provide a real world analogous example here to help readers correlate this story to their world and data analysis needs, "…This example is similar to…" Amy////

The Hadoop equivalent of course is that a data record may cross and HDFS block boundary. (In fact, you can force map-reduce splits to happen anywhere in the file, but the default and typically most-efficient choice is to split at HDFS blocks.)

A mapper will skip the first record of a split if it's partial and carry on from there. Since there are many records in each split, that's no big deal. When it gets to the end of the split, the task doesn't stop processing until is completes the current record — the framework makes the overhanging data seamlessley appear. //// Again, here, correlate this example to a real world scenario; "…so if you were translating x, this means that…" Amy////

In practice, Hadoop users only need to worry about record splitting when writing a custom `InputFormat` or when practicing advanced magick. You'll see lots of reference to it though — it's a crucial subject for those inside the framework, but for regular users the story I just told is more than enough detail.

Tuning For The Wise and Lazy

Before tuning your job, it is important to first understand whether it is slow and if so, where the bottleneck arises. Generally speaking, your Hadoop job will either be CPU-bound within your code, memory-bound within your code, bound by excessive Reducer data (causing too many merge passes) or throughput-bound within the framework or underlying system. (In the rare cases it is not one of those, you would next apply the more comprehensive "USE method" (TODO: REF) described later in this chapter.) But for data-intensive jobs, the fundamental upper bound on Hadoop's performance is its effective throughput from disk. You cannot make your job process data faster than it can load it.

Here is the conceptual model we will use: Apart from necessary overhead, job coordination and so forth, a Hadoop job must:

- Streams data to the Mapper, either locally from disk or remotely over the network;
- Runs that data through your code;
- Spills the midstream data to disk one or more times;
- Applies combiners, if any;
- Merge/Sorts the spills and sends the over the network to the Reducer.

The Reducer:

- Writes each Mapper's output to disk;
- Performs some number of Merge/Sort passes;
- Reads the data from disk;
- Runs that data through your code;
- Writes its output to the DataNode, which writes that data once to disk and twice through the network;
- Receives two other Reducers' output from the network and writes those to disk.

Hadoop is, of course, pipelined; to every extent possible, those steps are happening at the same time. What we will do, then, is layer in these stages one by one, at each point validating that your job is as fast as your disk until we hit the stage where it is not.

Fixed Overhead

The first thing we want is a job that does nothing; this will help us understand the fixed overhead costs. Actually, what we will run is a job that does almost nothing; it is useful to know that your test really did run.

```
(TODO:  Disable combining splits)
Load 10_tiny_files
Filter most of it out
Store to disk

(TODO:  Restrict to 50 Mapper slots or rework)
Load 10000_tiny_files
Filter most of it out
Store to disk
```

(TODO: is there a way to limit the number of Reduce slots in Pig? Otherwise, revisit the below.)

In (TODO: REF), there is a performance comparison worksheet that you should copy and fill in as we go along. It lists the performance figures for several reference clusters on both cloud and dedicated environments for each of the tests we will perform. If your figures do not compare well with the appropriate reference cluster, it is probably worth-

while adjusting the overall configuration. Assuming your results are acceptable, you can tape the worksheet to your wall and use it to baseline all the jobs you will write in the future. The rest of the chapter will assume that your cluster is large enough to warrant tuning but not grossly larger than the largest reference cluster.

If you run the Pig script above (TODO: REF), Hadoop will execute two jobs: one with 10 Mappers and no Reducers and another with 10,000 Mappers and no Reducers. From the Hadoop Job Tracker page for your job, click on the link showing the number of Map tasks to see the full task listing. All 10 tasks for the first job should have started at the same time and uniformly finished a few seconds after that. Back on the main screen, you should see that the total job completion time was more or less identical to that of the slowest Map task.

The second job ran its 10,000 Map tasks through a purposefully restricted 50 Mapper slots — so each Mapper slot will have processed around 200 files. If you click through to the Task listing, the first wave of tasks should start simultaneously and all of them should run in the same amount of time that the earlier test did.

(TODO: show how to find out if one node is way slower)

Even in this trivial case, there is more variance in launch and runtimes than you might first suspect (if you don't, you definitely will in the next — but for continuity, we will discuss it here). If that splay — the delay between the bulk of jobs finishing and the final job finishing — is larger than the runtime of a typical task, however, it may indicate a problem, but as long as it is only a few seconds, don't sweat it. If you are interested in a minor but worth-it tweak, adjust the `mapred.job.reuse.jvm.num.tasks` setting to '10'. This causes each Mapper to use the same child process JVM for multiple attempts, minimizing the brief but noticeable JVM startup time's impact. If you are writing your own native Java code, you might know a reason to force no reuse (the default), but it is generally harmless for any well-behaved program.

On the Job screen, you should see that the total runtime for the job was about 200 times slower for the second job than the first and not much more than 200 times the typical task's runtime; if not, you may be putting pressure on the Job Tracker. Rerun your job and watch the Job Tracker's heap size; you would like the Job Tracker heap to spend most of its life below, say 50-percent, so if you see it making any significant excursions toward 100-percent, that would unnecessarily impede cluster performance. The 1 GB out-of-the-box setting is fairly small; for production use we recommend at least 3 GB of heap on a dedicated machine with at least 7 GB total RAM.

If the Job coordination overhead is unacceptable but the Job Tracker heap is not to blame, a whole host of other factors might be involved; apply the USE method, described (TODO: REF).

Mapper Input

Now that we've done almost nothing, let's do almost something — read in a large amount of data, writing just enough to disk to know that we really were there.

```
Load 100 GB from disk
Filter all but 100 MB
Store it to disk
```

Run that job on the 100-GB GitHub archive dataset. (TODO: Check that it will do speculative execution.) Once the job completes, you will see as many successful Map tasks as there were HDFS blocks in the input; if you are running a 128-MB block size, this will be about (TODO: How many blocks are there?).

Again, each Map task should complete in a uniform amount of time and the job as a whole should take about 'length_of_Map_task*number_of_Map_tasks=number_of_Mapper_slots'. The Map phase does not end until every Mapper task has completed and, as we saw in the previous example, even in typical cases, there is some amount of splay in runtimes.

(TODO: Move some of JT and Nanette's optimizations forward to this chapter). Like the chimpanzees at quitting time, the Map phase cannot finish until all Mapper tasks have completed.

You will probably notice a half-dozen or so killed attempts as well. The 'TODO: name of speculative execution setting', which we recommend enabling, causes Hadoop to opportunistically launch a few duplicate attempts for the last few tasks in a job. The faster job cycle time justifies the small amount of duplicate work.

Check that there are few non-local Map tasks — Hadoop tries to assign Map attempts (TODO: check tasks versus attempts) to run on a machine whose DataNode holds that input block, thus avoiding a trip across the network (or in the chimpanzees' case, down the hallway). It is not that costly, but if you are seeing a large number of non-local tasks on a lightly-loaded cluster, dig deeper.

Dividing the average runtime by a full block of Map task by the size of an HDFS block gives you the Mapper's data rate. In this case, since we did almost nothing and wrote almost nothing, that value is your cluster's effective top speed. This has two implications: First, you cannot expect a data-intensive job to run faster than its top speed. Second, there should be apparent reasons for any job that runs much slower than its top speed. Tuning Hadoop is basically about making sure no other part of the system is slower than the fundamental limit at which it can stream from disk.

While setting up your cluster, it might be worth baselining Hadoop's top speed against the effective speed of your disk and your network. Follow the instructions for the 'scripts/baseline_performance' script (TODO: write script) from the example code above. It uses a few dependable user-level processes to measure the effective data rate to disk

('DD' and 'CP') and the effective network rate ('NC' and 'SCP'). (We have purposely used user-level processes to account for system overhead; if you want to validate that as well, use a benchmark like Bonnie++ (TODO: link)). If you are dedicated hardware, the network throughput should be comfortably larger than the disk throughput. If you are on cloud machines, this, unfortunately, might not hold but it should not be atrociously lower.

If the effective top speed you measured above is not within (TODO: figure out healthy percent) percent, dig deeper; otherwise, record each of these numbers on your performance comparison chart.

If you're setting up your cluster, take the time to generate enough additional data to keep your cluster fully saturated for 20 or more minutes and then ensure that each machine processed about the same amount of data. There is a lot more variance in effective performance among machines than you might expect, especially in a public cloud environment; it can also catch a machine with faulty hardware or setup. This is a crude but effective benchmark, but if you're investing heavily in a cluster consider running a comprehensive benchmarking suite on all the nodes — the chapter on Stupid Hadoop Tricks shows how (TODO ref)

The Many Small Files Problem

One of the most pernicious ways to impair a Hadoop cluster's performance is the "many-small-files" problem. With a 128-MB block size (which we will assume for the following discussion), a 128-MB file takes one block (obviously), a 1-byte file takes one block and a 128-MB+1 byte file takes two blocks, one of them full, the other with one solitary byte.

Storing 10 GB of data in, say, 100 files is harmless — the average block occupancy is a mostly-full 100 MB. Storing that same 10GB in say 10,000 files is, however, harmful in several ways. At the heart of the Namenode is a table that lists every file and block. As you would expect, the memory usage of that table roughly corresponds to the number of files plus the number of blocks, so the many-small-files example uses about 100 times as much memory as warranted. Engage in that bad habit often enough and you will start putting serious pressure on the Namenode heap and lose your job shortly thereafter. What is more, the many-small-files version will require 10,000 Map tasks, causing memory pressure on the Job Tracker and a job whose runtime is dominated by task overhead. Lastly, there is the simple fact that working with 10,000 things is more annoying than working with 100 — it takes up space in datanode heartbeats, client requests, your terminal screen and your head.

Causing this situation is easier to arrive at than you might expect; in fact, you just did so. The 100-GB job you just ran most likely used 800 Mapper slots yet output only a few MB of data. Any time your mapper output is significantly smaller than its input — for

example, when you apply a highly-restrictive filter to a large input — your output files will have poor occupancy.

A sneakier version of this is a slightly "expansive" Mapper-Only job. A job whose Mappers turned a 128-MB block into, say, 150 MB of output data would reduce the block occupancy by nearly half and require nearly double the Mapper slots in the following jobs. Done once, that is merely annoying but in a workflow that iterates or has many stages, the cascading dilution could become dangerous.

You can audit your HDFS to see if this is an issue using the 'hadoop fsck [directory]' command. Running that command against the directory holding the GitHub data should show 100 GB of data in about 800 blocks. Running it against your last job's output should show only a few MB of data in an equivalent number of blocks.

You can always distill a set of files by doing 'group_by' with a small number of Reducers using the record itself as a key. Pig and Hive both have settings to mitigate the many-small-files problem. In Pig, the (TODO: find name of option) setting will feed multiple small files to the same Mapper; in Hive (TODO: look up what to do in Hive). In both cases, we recommend modifying your configuration to make that the default and disable it on a per-job basis when warranted.

Midstream Data

Now let's start to understand the performance of a proper Map/Reduce job. Run the following script, again, against the 100 GB GitHub data.

```
Parallel 50
Disable optimizations for pushing up filters and for Combiners
Load 100 GB of data
Group by record itself
Filter out almost everything
Store data
```

The purpose of that job is to send 100 GB of data at full speed through the Mappers and midstream processing stages but to do almost nothing in the Reducers and write almost nothing to disk. To keep Pig from "helpfully" economizing the amount of midstream data, you will notice in the script we disabled some of its optimizations. The number of Map tasks and their runtime should be effectively the same as in the previous example, and all the sanity checks we've given so far should continue to apply. The overall runtime of the Map phase should only be slightly longer (TODO: how much is slightly?) than in the previous Map-only example, depending on how well your network is able to outpace your disk.

It is an excellent idea to get into the habit of predicting the record counts and data sizes in and out of both Mapper and Reducer based on what you believe Hadoop will be doing to each record and then comparing to what you see on the Job Tracker screen. In this case, you will see identical record counts for Mapper input, Mapper output and Reducer

input and nearly identical data sizes for HDFS bytes read, Mapper output, Mapper file bytes written and Reducer input. The reason for the small discrepancies is that, for the file system metrics, Hadoop is recording everything that is read or written, including logged files and so forth.

Midway or so through the job — well before the finish of the Map phase — you should see the Reducer tasks start up; their eagerness can be adjusted using the (TODO: name of setting) setting. By starting them early, the Reducers are able to begin merge/sorting the various Map task outputs in parallel with the Map phase. If you err low on this setting, you will disappoint your coworkers by consuming Reducer slots with lots of idle time early but that is better than starting them too late, which will sabotage parallels.

Visit the Reducer tasks listing. Each Reducer task should have taken a uniform amount of time, very much longer than the length of the Map tasks. Open a few of those tasks in separate browser tabs and look at their counters; each should have roughly the same input record count and data size. It is annoying that this information is buried as deeply as it is because it is probably the single most important indicator of a flawed job; we will discuss it in detail a bit later on.

Spills

First, though, let's finish understanding the data's detailed journey from Mapper to Reducer. As a Map task outputs records, Hadoop sorts them in the fixed-size io.sort buffer. Hadoop files records into the buffer in partitioned, sorted order as it goes. When that buffer fills up (or the attempt completes), Hadoop begins writing to a new empty io.sort buffer and, in parallel, "spills" that buffer to disk. As the Map task concludes, Hadoop merge/sorts these spills (if there were more than one) and sends the sorted chunks to each Reducer for further merge/sorting.

The Job Tracker screen shows the number of Mapper spills. If the number of spills equals the number of Map tasks, all is good — the Mapper output is checkpointed to disk before being dispatched to the Reducer. If the size of your Map output data is large, having multiple spills is the natural outcome of using memory efficiently; that data was going to be merge/sorted anyway, so it is a sound idea to do it on the Map side where you are confident it will have a uniform size distribution.

(TODO: do combiners show as multiple spills?)

What you hate to see, though, are Map tasks with two or three spills. As soon as you have more than one spill, the data has to be initially flushed to disk as output, then read back in full and written again in full for at least one merge/sort pass. Even the first extra spill can cause roughly a 30-percent increase in Map task runtime.

There are two frequent causes of unnecessary spills. First is the obvious one: Mapper output size that slightly outgrows the io.sort buffer size. We recommend sizing the io.sort buffer to comfortably accommodate Map task output slightly larger than your typical

HDFS block size — the next section (TODO: REF) shows you how to calculate. In the significant majority of jobs that involve a Reducer, the Mapper output is the same or nearly the same size — JOINs or GROUPs that are direct, are preceded by a projection or filter or have a few additional derived fields. If you see many of your Map tasks tripping slightly over that limit, it is probably worth requesting a larger io.sort buffer specifically for your job.

There is also a disappointingly sillier way to cause unnecessary spills: The io.sort buffer holds both the records it will later spill to disk and an index to maintain the sorted order. An unfortunate early design decision set a fixed size on both of those with fairly confusing control knobs. The 'iosortrecordpercent' (TODO: check name of setting) setting gives the size of that index as a fraction of the sort buffer. Hadoop spills to disk when either the fraction devoted to records or the fraction devoted to the index becomes full. If your output is long and skinny, cumulatively not much more than an HDFS block but with a typical record size smaller than, say, 100 bytes, you will end up spilling multiple small chunks to disk when you could have easily afforded to increase the size of the bookkeeping buffer.

There are lots of ways to cause long, skinny output but set a special triggers in your mind for cases where you have long, skinny input; turn an adjacency-listed graph into an edge-listed graph or otherwise FLATTEN bags of records on the Mapper side. In each of these cases, the later section (TODO: REF) will show you how to calculate it.

(TODO: either here or later, talk about the surprising cases where you fill up MapRed scratch space or FS.S3.buffer.dir and the rest of the considerations about where to put this).

Combiners

It is a frequent case that the Reducer output is smaller than its input (and kind of annoying that the word "Reducer" was chosen, since it also frequently is not smaller). "Algebraic" aggregations such as COUNT, AVG and so forth, and many others can implement part of the Reducer operation on the Map side, greatly lessening the amount of data sent to the Reducer.

Pig and Hive are written to use Combiners whenever generically appropriate. Applying a Combiner requires extra passes over your data on the Map side and so, in some cases, can themselves cost much more time than they save.

If you ran a distinct operation over a data set with 50-percent duplicates, the Combiner is easily justified since many duplicate pairs will be eliminated early. If, however, only a tiny fraction of records are duplicated, only a disappearingly-tiny fraction will occur on the same Mapper, so you will have spent disk and CPU without reducing the data size.

Whenever your Job Tracker output shows that Combiners are being applied, check that the Reducer input data is, in fact, diminished. (TODO: check which numbers show this)

If Pig or Hive have guessed badly, disable the (TODO: name of setting) setting in Pig or the (TODO: name of setting) setting in Hive.

Reducer Merge (aka Shuffle and Sort)

We are now ready to dig into the stage with the most significant impact on job performance, the merge/sort performed by the Reducer before processing. In almost all the rest of the cases discussed here, an inefficient choice causes only a marginal impact on runtime. Bring down too much data on your Reducers, however, and you will find that, two hours into the execution of what you thought was a one-hour job, a handful of Reducers indicate they have four hours left to run.

First, let's understand what is going on and describe healthy execution; then, we will discuss various ways it can go wrong and how to address them.

As you just saw, data arrives from the Mappers pre-sorted. The Reducer reads them from memory into its own sort buffers. Once a threshold (controlled by the (TODO: name of setting) setting) of data has been received, the Reducer commissions a new sort buffer and separately spills the data to disk, merge/sorting the Mapper chunks as it goes. (TODO: check that this first merge/sort happens on spill)

Enough of these spills later (controlled by the (TODO: setting) setting), the Reducer begins merge/sorting the spills into a larger combined chunk. All of this activity is happening in parallel, so by the time the last Map task output is received, the typical healthy situation is to have a modest number of large sorted chunks and one small-ish chunk holding the dregs of the final spill. Once the number of chunks is below the (TODO: look up name of setting) threshold, the merge/sort is complete — it does not need to fully merge the data into a single file onto disk. Instead, it opens an input stream onto each of those final chunks, consuming them in sort order.

Notice that the Reducer flushes the last spill of received Map data to disk, then immediately starts reconsuming it. If the memory needs of your Reducer are modest, you can instruct Hadoop to use the sort buffer directly in the final merge, eliminating the cost and delay of that final spill. It is a nice marginal improvement when it works but if you are wrong about how modest your Reducer's memory needs are, the negative consequences are high and if your Reducers have to perform multiple merge/sort passes, the benefits are insignificant.

For a well-tested job heading to production that requires one or fewer merge/sort passes, you may judiciously (TODO: describe how to adjust this).

(TODO: discuss buffer sizes here or in Brave and Foolish section) (TODO: there is another setting that I'm forgetting here - what is it?)

Once your job has concluded, you can find the number of merge/sort passes by consulting the Reduce tasks counters (TODO: DL screenshot and explanation). During the

job, however, the only good mechanism is to examine the Reducer logs directly. At some reasonable time after the Reducer has started, you will see it initiate spills to disk (TODO: tell what the log line looks like). At some later point, it will begin merge/sorting those spills (TODO: tell what the log line looks like).

The CPU burden of a merge/sort is disappearingly small against the dominating cost of reading then writing the data to disk. If, for example, your job only triggered one merge/sort pass halfway through receiving its data, the cost of the merge/sort is effectively one and a half times the base cost of writing that data at top speed to disk: all of the data was spilled once, half of it was rewritten as merged output. Comparing the total size of data received by the Reducer to the merge/sort settings will let you estimate the expected number of merge/sort passes; that number, along with the "top speed" figure you collected above, will, in turn, allow you to estimate how long the Reduce should take. Much of this action happens in parallel but it happens in parallel with your Mapper's mapping, spilling and everything else that is happening on the machine.

A healthy, data-intensive job will have Mappers with nearly top speed throughput, the expected number of merge/sort passes and the merge/sort should conclude shortly after the last Map input is received. (TODO: tell what the log line looks like). In general, if the amount of data each Reducer receives is less than a factor of two to three times its share of machine RAM, (TODO: should I supply a higher-fidelity thing to compare against?) all those conditions should hold. Otherwise, consult the USE method (TODO: REF).

If the merge/sort phase is killing your job's performance, it is most likely because either all of your Reducers are receiving more data than they can accommodate or because some of your Reducers are receiving far more than their fair share. We will take the uniform distribution case first.

The best fix to apply is to send less data to your Reducers. The chapters on writing Map/Reduce jobs (TODO: REF or whatever we are calling Chapter 5) and the chapter on advanced Pig (TODO: REF or whatever we are calling that now) both have generic recommendations for how to send around less data and throughout the book, we have described powerful methods in a domain-specific context which might translate to your problem.

If you cannot lessen the data burden, well, the laws of physics and economics must be obeyed. The cost of a merge/sort is 'O(N LOG N)'. In a healthy job, however, most of the merge/sort has been paid down by the time the final merge pass begins, so up to that limit, your Hadoop job should run in 'O(N)' time governed by its top speed.

The cost of excessive merge passes, however, accrues directly to the total runtime of the job. Even though there are other costs that increase with the number of machines, the benefits of avoiding excessive merge passes are massive. A cloud environment makes it particularly easy to arbitrage the laws of physics against the laws of economics — it costs

the same to run 60 machines for two hours as it does to run ten machines for 12 hours, as long as your runtime stays roughly linear with the increased number of machines, you should always size your cluster to your job, not the other way around. The thresholding behavior of excessive reduces makes it exceptionally valuable to do so. This is why we feel exploratory data analytics is far more efficiently done in an elastic cloud environment, even given the quite significant performance hit you take. Any physical cluster is too large and also too small; you are overpaying for your cluster overnight while your data scientists sleep and you are overpaying your data scientists to hold roller chair sword fights while their undersized cluster runs. Our rough rule of thumb is to have not more than 2-3 times as much total reducer data as you have total child heap size on all the reducer machines you'll use.

(TODO: complete)

Skewed Data and Stuck Reducers

(TODO: complete)

Reducer Processing

(TODO: complete)

Commit and Replication

(TODO: complete)

Cluster Sizing Rules of thumb

Let's define

- ram_total == total cluster ram = ram_per_machine * num_machines
- cpu_bw_per_slot == actual CPU bandwidth per mapreduce slot for a highly cpu-intensive job ~= actual CPU bandwidth / cores per machine
- disk_bw_per_slot == actual disk bandwidth per mapreduce slot ~= actual disk bandwidth / cores per machine
- ntwk_bw_per_slot == actual ntwk bandwidth per mapreduce slot ~= actual ntwk bandwidth / cores per machine
- X = amount of data sent to the reducers in the largest job the customer expects to run performantly.
- max_data_under_management == amount of data stored on HDFS, not accounting for replication

Consider a job that reads X data into mapper, sends X data to the reducer, and emits X data from that reducer (put another way: one of the following stages will be limiting, in which case increasing the amount of data the other stages handle won't slow down the job.)

- A: mapper stage
  - — reading X data from hdfs disk for mapper via local datanode
  - — processing X data in mapper (not limiting)
  - — spilling X data to scratch disk (parallelized with mapper reading)
  - — sending X data over network to reducer (parallelized with mapper reading)
  - — writing X data to scratch disk in first-pass merge sorting (parallelized with mapper reading)
- B: follow-on merge sort stage
  - — reading/writing (Q-2)*X data to scratch disk in follow-on merge sort passes, where Q is the total number of merge-sort passes required (somewhat parallelized with above). It goes as the log of the reduce size over the reducer child process ram size.
- C: output stage
  - — reading X data from scratch disk for reducer
  - — processing X data in reducer (not limiting)
  - — writing X data to hdfs disk via local datanode (not limiting)
  - — sending/receiving 2*X data over network for replication (parallelized with reducing)
  - — writing 2*X data to disk for replication by local datanode (parallelized with reducing)

```
* A well-configured cluster has disk_bw_per_slot as the limiting bottleneck.
  - ntwk_bw_per_slot >= disk_bw_per_slot in-rack
  - ntwk_bw_per_slot >=  80% disk_bw_per_slot between pairs of machines in different racks under
  - cpu_bw_per_slot >= disk_bw_per_slot -- it's actually really hard to find the practical job w
* Thus stage A takes X / disk_bw_per_slot amount of time
* And stage C takes 3*X / disk_bw_per_slot amount of time
* If ram_total >= 40% X, the number of merge passes won't much exceed the runtime of the mappers
```

Implications:

- ram_total = 40% of the largest amount of data they care to regularly process
- cpu = size each machine so its cpu is faster than its disk, and not faster
- network = size each machine so its network is faster than its disk, and not too much faster

- cross-rack switches = size the cross-rack switches so actual bandwidth between pairs of machines in different racks under full load isn't too much slower than the in-rack bandwidth (say, 80%)
- the bigjob you've just sized the cluster against should take about (5 * X / disk_bw_per_machine) to run
- total cluster disk capacity = size to 5 * max_data_under_management — factor of 3 for replication, another factor of 1.5 for computing by-production, and then bump it up for overhead
  - scratch space volume capacity >= 2 * X — I assume that X is much less than max_data_under_management, so scratch space fits in the overhead. It's really nice to have the scratch disks and the hdfs disks held separate.

Top-line Performance/Sanity Checks

- The first wave of Mappers should start simultaneously.
- In general, all a job's full block Map attempts should take roughly the same amount of time.
- The full map phase should take around 'average_Map_task_time*(number_of_Map_tasks/number_of_Mapper_slots+1)'
- Very few non-local Map tasks.
- Number of spills equals number of Map tasks (unless there are Combiners).
- If there are Combiners, the Reducer input data should be much less than the Mapper output data (TODO: check this).
- Record counts and data sizes for Mapper input, Reducer input and Reducer output should correspond to your conception of what the job is doing.
- Map tasks are full speed (data rate matches your measured baseline)
- Most Map tasks process a full block of data.
- Processing stage of Reduce attempts should be full speed.
- Not too many Merge passes in Reducers.
- Shuffle and sort time is explained by the number of Merge passes.
- Commit phase should be brief.
- Total job runtime is not much more than the combined Map phase and Reduce phase runtimes.
- Reducers generally process the same amount of data.
- Most Reducers process at least enough data to be worth it. *

Performance Comparison Worksheet

(TODO: DL Make a table comparing performance baseline figures on AWS and fixed hardware. reference clusters.)

CHAPTER 23

Hadoop Tuning for the Brave and Foolish

The default settings are those that satisfy in some mixture the constituencies of a) Yahoo, Facebook, Twitter, etc; and b) Hadoop developers, ie. peopl who **write** Hadoop but rarely **use** Hadoop. This means that many low-stakes settings (like keeping jobs stats around for more than a few hours) are at the values that make sense when you have a petabyte-scale cluster and a hundred data engineers;

- If you're going to run two master nodes, you're a bit better off running one master as (namenode only) and the other master as (jobtracker, 2NN, balancer) — the 2NN should be distinctly less utilized than the namenode. This isn't a big deal, as I assume your master nodes never really break a sweat even during heavy usage.

Memory

Here's a plausible configuration for a 16-GB physical machine with 8 cores:

```
`mapred.tasktracker.reduce.tasks.maximum`   = 2
`mapred.tasktracker.map.tasks.maximum`      = 5
`mapred.child.java.opts`                    = 2 GB
`mapred.map.child.java.opts`                = blank (inherits mapred.child.java.opts)
`mapred.reduce.child.java.opts`             = blank (inherits mapred.child.java.opts)

total mappers' heap size                    = 10   GB (5 * 2GB)
total reducers' heap size                   =  4   GB (2 * 2GB)
datanode heap size                          =  0.5 GB
tasktracker heap size                       =  0.5 GB
.....                                         ...
total                                       = 15   GB on a 16 GB machine
```

- It's rare that you need to increase the tasktracker heap at all. With both the TT and DN daemons, just monitor them under load; as long as the heap healthily exceeds their observed usage you're fine.

- If you find that most of your time is spent in reduce, you can grant the reducers more RAM with `mapred.reduce.child.java.opts` (in which case lower the child heap size setting for the mappers to compensate).
  - It's standard practice to disable swap — you're better off OOM'ing footnote[OOM = Out of Memory error, causing the kernel to start killing processes outright] than swapping. If you do not disable swap, make sure to reduce the `swappiness` sysctl (5 is reasonable). Also consider setting `overcommit_memory` (1) and `overcommit_ratio` (100). Your sysadmin might get angry when you suggest these changes — on a typical server, OOM errors cause pagers to go off. A misanthropically funny T-shirt, or whiskey, will help establish your bona fides.
  - `io.sort.mb` default `X`, recommended at least `1.25 * typical output size` (so for a 128MB block size, 160). It's reasonable to devote up to 70% of the child heap size to this value.
  - `io.sort.factor`: default `X`, recommended `io.sort.mb * 0.x5 * (seeks/s) / (thruput MB/s)`
- you want transfer time to dominate seek time; too many input streams and the disk will spend more time switching among them than reading them.
- you want the CPU well-fed: too few input streams and the merge sort will run the sort buffers dry.
- My laptop does 76 seeks/s and has 56 MB/s throughput, so with `io.sort.mb = 320` I'd set `io.sort.factor` to 27.
- A server that does 100 seeks/s with 100 MB/s throughput and a 160MB sort buffer should set `io.sort.factor` to 80.

  - `io.sort.record.percent` default `X`, recommended `X` (but adjust for certain jobs)
  - `mapred.reduce.parallel.copies`: default `X`, recommended to be in the range of `sqrt(Nw*Nm)` to `Nw*Nm/2` You should see the shuffle/copy phase of your reduce tasks speed up.
  - `mapred.job.reuse.jvm.num.tasks` default `1`, recommended `10`. If a job requires a fresh JVM for each process, you can override that in its jobconf. Going to `-1` (reuse unlimited times) can fill up the dist if your input format uses "delete on exit" temporary files (as for example the S3 filesystem does), with little additional speedup.
  - You never want Java to be doing stop-the-world garbage collection, but for large JVM heap sizes (above 4GB) they can become especially dangerous. If a full garbage collect takes too long, sockets can time out, causing loads to increase, causing garbage collects to happen, causing… trouble, as you can guess.

— Given the number of files and amount of data you're storing, I would set the NN heap size agressively - at least 4GB to start, and keep an eye on it. Having the NN run out of memory is Not Good. Always make sure the secondary name node has the same heap setting as the name node.

Handlers and threads

- `dfs.namenode.handler.count`: default `X`, recommended: `(0.1 to 1) * size of cluster`, depending on how many blocks your HDFS holds.
- `tasktracker.http.threads` default `X`, recommended `X`
- Set `mapred.reduce.tasks` so that all your reduce slots are utilized — If you typically only run one job at a time on the cluster, that means set it to the number of reduce slots. (You can adjust this per-job too). Roughly speaking: keep `number of reducers * reducer memory` within a factor of two of your reduce data size.
- `dfs.datanode.handler.count`: controls how many connections the datanodes can maintain. It's set to 3 — you need to account for the constant presence of the flume connections. I think this may be causing the datanode problems. Something like 8-10 is appropriate.
- You've increased `dfs.datanode.max.xcievers` to 8k, which is good.
- `io.file.buffer.size`: default `X` recommended `65536`; always use a multiple of `4096`.

Storage

- `mapred.system.dir`: default `X` recommended `/hadoop/mapred/system` Note that this is a path on the HDFS, not the filesystem).
- Ensure the HDFS data dirs (`dfs.name.dir`, `dfs.data.dir` and `fs.checkpoint.dir`), and the mapreduce local scratch dirs (`mapred.system.dir`) include all your data volumes (and are off the root partition). The more volumes to write to the better. Include all the volumes in all of the preceding. If you have a lot of volumes, you'll need to ensure they're all attended to; have 0.5-2x the number of cores as physical volumes.
  — HDFS-3652 — don't name your dirs `/data1/hadoop/nn`, name them `/data1/hadoop/nn1` (final element differs).
- Solid-state drives are unjustifiable from a cost perspective. Though they're radically better on seek they don't improve performance on bulk transfer, which is what limits Hadoop. Use regular disks.

- Do not construct a RAID partition for Hadoop — it is happiest with a large JBOD. (There's no danger to having hadoop sit on top of a RAID volume; you're just hurting performance).
- We use `xfs`; I'd avoid `ext3`.
- Set the `noatime` option (turns off tracking of last-access-time) — otherwise the OS updates the disk on every read.
- Increase the ulimits for open file handles (`nofile`) and number of processes (`nproc`) to a large number for the `hdfs` and `mapred` users: we use `32768` and `50000`.

  — be aware: you need to fix the ulimit for root (?instead `?` as well?)
- `dfs.blockreport.intervalMsec`: default 3_600_000 (1 hour); recommended 21_600_000 (6 hours) for a large cluster.

  — 100_000 blocks per data node for a good ratio of CPU to disk

Other

- `mapred.map.output.compression.codec`: default XX, recommended ``. Enable Snappy codec for intermediate task output.

  — `mapred.compress.map.output`

  — `mapred.output.compress`

  — `mapred.output.compression.type`

  — `mapred.output.compression.codec`
- `mapred.reduce.slowstart.completed.maps`: default `X`, recommended `0.2` for a single-purpose cluster, `0.8` for a multi-user cluster. Controls how long, as a fraction of the full map run, the reducers should wait to start. Set this too high, and you use the network poorly — reducers will be waiting to copy all their data. Set this too low, and you will hog all the reduce slots.
- `mapred.map.tasks.speculative.execution`: default: `true`, recommended: `true`. Speculative execution (FIXME: explain). So this setting makes jobs finish faster, but makes cluster utilization higher; the tradeoff is typically worth it, especially in a development environment. Disable this for any map-only job that writes to a database or has side effects besides its output. Also disable this if the map tasks are expensive and your cluster utilization is high.
- `mapred.reduce.tasks.speculative.execution`: default `false`, recommended: `false`.
- (hadoop log location): default `/var/log/hadoop`, recommended `/var/log/hadoop` (usually). As long as the root partition isn't under heavy load, store the logs

on the root partition. Check the Jobtracker however — it typically has a much larger log volume than the others, and low disk utilization otherwise. In other words: use the disk with the least competition.

- `fs.trash.interval` default `1440` (one day), recommended `2880` (two days). I've found that files are either a) so huge I want them gone immediately, or b) of no real concern. A setting of two days lets you to realize in the afternoon today that you made a mistake in the morning yesterday,
- Unless you have a ton of people using the cluster, increase the amount of time the jobtracker holds log and job info; it's nice to be able to look back a couple days at least. Also increase `mapred.jobtracker.completeuserjobs.maximum` to a larger value. These are just for politeness to the folks writing jobs.
  - `mapred.userlog.retain.hours`
  - `mapred.jobtracker.retirejob.interval`
  - `mapred.jobtracker.retirejob.check`
  - `mapred.jobtracker.completeuserjobs.maximum`
  - `mapred.job.tracker.retiredjobs.cache`
  - `mapred.jobtracker.restart.recover`
- Bump `mapreduce.job.counters.limit` — it's not configurable per-job.

(From *http://blog.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/ — 512M* block size fairly reasonable)

CHAPTER 24

Storm+Trident Internals

What should you take away from this chapter:

You should:

- Understand the lifecycle of a Storm tuple, including spout, tupletree and acking.
- (Optional but not essential) Understand the details of its reliability mechanism and how tuples are acked.
- Understand the lifecycle of partitions within a Trident batch and thus, the context behind partition operations such as Apply or PartitionPersist.
- Understand Trident's transactional mechanism, in the case of a PartitionPersist.
- Understand how Aggregators, Statemap and the Persistence methods combine to give you *exactly once* processing with transactional guarantees. Specifically, what an OpaqueValue record will look like in the database and why.
- Understand how the master batch coordinator and spout coordinator for the Kafka spout in particular work together to uniquely and efficiently process all records in a Kafka topic.
- One specific: how Kafka partitions relate to Trident partitions.

Storm tuple lifecycle

Once the Coordinator has brought the dataflow online, the Worker (TODO: ?Executor?) calls the spouts next tuple operator. The spout emits a tuple to its Collector if it has one ready. (Actually, the spout is permitted to emit 0, 1 or *many tuples* — but you should try to emit just one unless there's a good reason.) It then registers that tuple as pending until its tupletree (this tuple and all the descendent tuples produced by later processing stages) are acked. Please note that though the spout pending mechanism

relies on the final result of the acking mechanism, it is distinct from that and handled by the spout's Executor. (TODO: Check)

If the spout doesn't emit a tuple, the Worker will sleep for a fixed number of milliseconds (by default, you can change the sleep policy). Otherwise, the Worker will keep calling `nextTuple` until either its send queue is full (see below) or until there are `MAX_SPOUT_PENDING` or more tuples pending.

Spout send queue

The Collector places that tuple into the Executor's send queue.

Since it's important that the spout never block when emitting (if it did, critical bookkeeping tuples might get trapped, locking up the flow), a spout emitter keeps an "overflow buffer," and publishes as follows:

- if there are tuples in the overflow buffer add the tuple to it — the queue is certainly full.
- otherwise, publish the tuple to the flow with the non-blocking call. That call will either succeed immediately …
- or fail with an `InsufficientCapacityException`, in which case add the tuple to the overflow buffer.

Executor Queues

At this point, you see that the spout spins in an independent loop, emitting records to its Collector until one of its limits is hit. We will pick up with the specific tuple in a moment but first, let's get a picture of how tuples move between Executors, locally and remotely. Each Executor, whether bolt or spout, has both a send and a receive queue. (For now, all you need to know about a bolt Executor is that it takes things from its receive queue, does stuff to them and puts them into the send queue.)

(TODO: Insert information on what disrupter queue is and how wonderful it is)

When a tuple is emitted, the Collector places each into a slot in the send queue, once for each downstream Executor that will process it. (The code, if you should find yourself reading it, doesn't distinguish the tuple as emitted and the copies used for sending. It has to make these multiple copies so that each can be acked independently.) These writes are done in a blocking fashion. If the queue is full, the "writemethod" does not return until it has been swept; this means that, in turn, the Collectors emit and the Executors execute methods block as well, preventing the Executor from sending more records than downstream stages can process.

The worker sweeps each Executor's send queue on a tight loop. Unless the downstream queue blocks, it repeats immediately (TODO: Check). Each sweep of the queue gathers all tuples, removing them into a temporary buffer. Tuples to be handled by a local Executor are deposited directly into that Executor's receive queue. All tuples destined for remote Executors are placed in the Worker's transfer queue in a single write, regardless of how many remote Executors there are.

A couple of notes: First, send and receive queues are per Executor, while the transfer queue is shared by all Executors in the Worker.

When the Worker writes to a downstream queue, it deposits all records from that sweep into the queue in a bunch. [FOOTNOTE: I'm being careful to use the informal term "bunch" rather than "batch" because you'll never hear about these again. A "batch" is a principal element for Trident, whereas we'll never talk about these again.] So note that, while each slot in a send queue holds exactly one tuple, each slot in a receive or transfer queue can hold up to the `SEND_QUEUE` size amount of tuples. (TODO: Check variable's name) This is important when you're thinking about memory usage.

Worker Transfer Queue

The Worker transport mechanism in turn sweeps the transfer queue and writes tuples over the network to each destination worker. There are two transport mechanisms, using either the legacy ZeroMQ transport or the more recent Netty transport. The ZeroMQ transport has been replaced due to both licensing issues and minor dissatisfaction with its performance in production. The Netty transport is probably the better choice, although at time of writing (November 2013), it's received much less production exposure.

In our experience, the remaining details of the Worker-to-Worker transfer are fairly unimportant. Records are sent over the network, accepted by the remote Worker and deposited into the relevant Executors' receive queues. There is such a thing as a Worker receive queue; this is not a disruptor queue and not a piece of user-maintainable machinery; ignore it and leave its configuration alone.

Executor Receive Queues

As you may now guess, each Executor runs in its own loop, sweeping each receive queue and removing all hanging bunches. Each of those tuples are handed, one after the other, to its bolt's `executemethod`.

So now, we can explain the backpressure mechanism of a Storm flow — the reason that tuples don't pile up unmanageably in RAM at the slowest stage. As mentioned, the `executemethod` won't return if anything downstream is blocking. Since each call to execute is done in series, this, in turn, prevents the Executor from iterating through all the tuples in a sweep — preventing it from beaconing a new sweep. Tuples will begin accumulating in a blocked Executor's receive queue. Ultimately, the Worker will, as well,

become blocked writing to that receive queue, keeping it from sweeping upstream send queues. This will continue all the way up the flow to the spout. Finally, as we hinted at the start, once the spout's send queue is full, it will stop calling `nexttuple`, stop draining its source and so stop writing more records into the flow.

If this sounds awfully coarse-grained, you're right. While nothing will *break* if you get to this L.A. freeway state of gridlock, your flow will have become disastrously inefficient, even well before that point. You can straightforwardly prevent the situation by adjusting the `maxspoutpending` parameter and each stage's parallelism correctly in the next chapter (TODO: ref), Storm+Trident Tuning, will show you how.

In normal operation, you shouldn't have to think about the backpressure mechanics, though; Storm quietly and efficiently buffers records in front of your slowest stages and handles latency shocks (such as transient sluggishness from a remote database or API).

Executor Details (?)

Not sure if I need anything here.

The Spout Pending Register

Say how a tuple is cleared from the pending register when its tree is finally acked and what this means for `maxspoutpending`.

Acking and Reliability

Storm's elegant acking mechanism is probably its most significant breakthrough. It ensures that a tuple and its descendents are processed successfully or fail loudly and it does so with a minimum amount of bookkeeping chatter. The rest of the sections in this chapter, while advanced, ultimately are helpful for architecting and productionizing a dataflow. This section, however, is comparatively optional — the whole point of the reliability mechanism is that it Just Works. It's so fascinating we can't help but include it but if you're not looking to have your brain bent today, feel free to skip it. Footnote: [This story is complicated enough that I'm going to make two minor simplifications that are areally only interesting if you're knee-deep in the code. We'll let you know about them in the footnotes.]

Here's how it works.

As each tuple destined for an Executor is created, it is given a unique enough ID; in practice, these are 64-bit integers (this will be important later) but I'm going to pretend that, by cosmic luck, each of those integers ends up resembling the name of a Biblical figure.

When a spout produces a tuple — let's take, for example, one named "Methuselah" — it notifies the acker to do two things: to start tracking Methuselah's tuple tree and to inscribe Methuselah's name in that tupletree's Scroll of Ages. [FOOTNOTE: Actually, since a tuple can be sent to multiple downstream Executors, it's more appropriate to say it inscribes each of Methuselah's clones in the Scroll of Ages.]

As described above, that tuple will eventually be processed by the downstream Executor's `execute` method, which typically emits tuples and must call `ack` or `fail`, (TODO: insert details of what happens when a tuple fails). In the typical case, the Executor's bolt happily calls `emit` 0, 1 or many times and then calls `ack`. As each emitted tuple is placed in the send queue, the Executor notes its name [FOOTNOTE: Actually, the names of all its clones.] for later delivery to the acker. When the bolt calls `ack`, the Executor notifies the acker with the name of the parent and each child.

So if a bolt, receiving a tuple called "Noah," emitted tuples called "Ham" and "Shem," it strikes Noah from the Scroll of Ages but lists Ham and Shem therein. (TODO: Rearrange?) When a bolt emits one or more tuples, the parent is removed but the children are added and so the Scroll of Ages continues to have at least those entries in it. If a bolt received a tuple called "Onan," and emitted nothing, then it would only notify the acker to clear Onan, adding nothing. Ultimately, for a tupletree to be successfully completed, every descendent must ultimately encounter a bolt that emits nothing.

Up until now, I've made it sound as if each name in the Scroll of Ages is maintained separately. The actual implementation is far more elegant than that and relies on a few special properties of the XOR function.

First, you can freely rearrange the order in which several terms are XOR'd together: `Noah XOR Shem XOR Ham` is the same as `Shem XOR Noah XOR Ham` and so forth. Second, the XOR of a term with itself is 0: Noah XOR Noah is 0 for anybody. Do you see where this is going? In our example, (TODO: Repair so it's Noah's tree)when the Scroll of Ages was first prepared, inscribed on it was only Noah's name. When the Executor handling that tuple notified back, it didn't have to send Noah, Ham and Shem distinctly; it just sent the single 64-bit integer `Noah XOR Ham XOR Shem`. So the Scroll of Ages is pretty brief, as Scrolls go; it actually only holds the one entry that is the combined XOR of every tuple ID that has been sent. So when the acker receives the ack for Noah, namely `Noah XOR Ham XOR Shem`, it XOR`s that single 64-bit entry with the existing tupletree `checksum` storing that `checksum` back to the Scroll of Ages. (NOTE: TODO Rework Scroll of Ages metaphor to hold all tupletrees.)

The value at this point is effectively `Noah XOR Shem XOR Ham`. From the first property, the Noah terms cancel out and so our tupletree state is now just `Shem XOR Ham`.

Thanks to the second property, even as acks come in asynchronously, the Scroll of Ages remains correct. `(Shem XOR Ham) XOR (Shem XOR Abraham) XOR (Ham) XOR (Abraham)` rearranges to provide two Shems, two Hams and two Abrahams. ,Since, in this

example, the family line of Shem and Abraham produced no resulting tuples, we are left with 0.

As soon as that last ack comes in, producing a 0 in the Scroll of Ages, the acker notifies the spout that the tupletree has concluded. This lets the spout remove that very first tuple from its pending list. The loop that calls `nexttuple` will, on its next trip through, see the new pending count and, if conditions are right, call `nexttuple`.

This system is thus able to accommodate many millions of active tuples with remarkably little network chatter or memory footprint. Only the spout's pending tuples are retained for anything except immediate processing. Now, this comes at a cost, if any downstream tuple fails, the whole tree is retried but since failure is the uncommon case, (and finite RAM is the universal case), this is the right tradeoff. Second, the XOR trick means a single 64-bit integer is sufficient to track the legacy of an entire tupletree, no matter how large, and a single 64-bit integer is all that has to be tracked and sent to the acker, no matter how many downstream tuples an Executor produces.

If you're scoring at home, for each tupletree, the entire bookkeeping system consumes Order(1) number of tuples, Order(1) size of `checksum` and only as many acks as tuples.

One last note. You can do the math on this yourself, but 64 bits is enough that the composed XOR of even millions of arbitrary 64-bit integer will effectively never come out to be 0 unless each term is repeated.

Lifecycle of a Trident batch

What should you take away from this chapter:

- Understand the lifecycle of partitions within a Trident batch and thus, the context behind partition operations such as Apply or PartitionPersist.
- Understand how the master batch coordinator and spout coordinator for the Kafka spout in particular work together to uniquely and efficiently process all records in a Kafka topic.
- One specific: how Kafka partitions relate to Trident partitions.
- Understand Trident's transactional mechanism, in the case of a PartitionPersist.
- Understand how Aggregators, Statemap and the Persistence methods combine to give you *exactly once* processing with transactional guarantees. Specifically, what an OpaqueValue record will look like in the database and why.

At this point, you've seen how the Storm layer efficiently and reliably handles individual tuples by the millions and billions. One of the most important things to keep in mind about the Trident layer is that every Trident flow is a Storm flow. It uses, without modification or improvement, all of Storm's machinery but extends it to provide a simpler

interface and the features (like aggregations and *exactly once* processing) that make Stream analytics possible.

You might think that we begin by talking about a Trident spout; after all, as you've been using Trident, that's where your flow conceptually begins. It's time we revealed a confusing clarification: The Trident spout is actually a Storm bolt. Viewed from the programmer interface, Trident spouts independently source tuples for downstream processing.

All of the Trident operations you are familiar with — spouts, each'es, aggregations — actually take place in Storm bolts. Trident turns your topology into a dataflow graph that it uses to assign operations to bolts and then to assign those bolts to workers. It's smart enough to optimize that assignment: It combines operations into bolts so that, as much as possible, tuples are handed off with simple method cause and it arranges bolts among workers so that, as much as possible, tuples are handed off to local Executors. (connecting material here)

The actual spout of a Trident topology is called the Master Batch Coordinator (MBC). From Storm's end, it's the dullest possible spout; all it does is emit a tuple describing itself as batch 1 and then a tuple describing itself as batch 2 and so forth, ad infinitum. (Of course, deciding when to emit those batches, retry them, etc., is quite exciting but Storm doesn't know anything about all that). Those batch tuples go to the topology's Spout Coordinator. The Spout Coordinator understands the location and arrangement of records in the external source and ensures that each source record belongs uniquely to a successful Trident batch.

The diagram on the right shows this in action for the GitHub topology. In this section, we are going to track three Trident matches (labeled 1, 2 and 3) through two parallel Kafka spouts, each pulling from a single Kafka partition. The Spout Coordinator passes the single seed tuple from the MBC onto each of its spouts, equipping each of them with a starting Kafka offset to read from. Each spout then requests, from the Kafka broker, a range of messages, beginning at its determined offset and extending to, at most, `Max_Fetch_Bytes` . If `Max_Fetch_Bytes` were, say, 1000 bytes, and your records were uniformly 300 bytes, Kafka would return to the spout just the next three records, totalling 900 bytes. You must set `Max_Fetch_Bytes` larger than your largest expected record; the Kafka spout will fail if a record is too large to fit in a single batch.

In most cases, records have a fairly bounded spread of sizes around a typical value. The GitHub records, for example, are (TODO: Find size x+- y bytes long). This means that a `Max_Fetch_Bytes` size of (TODO: value) might return as few as (A) and as many as (B). Pesky but harmless. If the size variance of your records is large enough to make this a real problem, unfortunately, you'll have to modify the Kafka spout.

Let's pause there and I'm going to tell you a story: Here's the system chimpanzee school children follow when they go on a field trip. At the start of the day, a set of school buses

pull up in waves at the school. The first graders all file onto the first set of buses and head off first followed by the set of buses for the second graders and so forth. As each bus pulls up at the museum, all the kids come off that bus in a group, known as a partition. All get off in a group (which we'll call a partition) and each group is met by a simple-minded docent assigned to that group by the museum. Now, chimpanzees are an unruly sort, but they are able to be well-behaved in at least the following way: All the chimpanzees in a partition follow the same path through the museum and no chimpanzee in the partition ever cuts ahead of another chimp. So, the third kid off the bus will see the same paintings as the second kid and the fourth kid and she'll see each of those paintings some time after the second kid did and some time before the fourth kid did. Each docent memorizes the number of students in its assigned partition, patiently takes a place in line after the last chimpanzee and follows along with them through the museum. If you visited the museum on chimpanzee field trip day, well, it can sometimes be just as chaotic as you'd expect, what with kids looking at the pointless paintings from up close and afar, a set of them hooting in recognition at the exhibition on ostrolopithazines and others gather to intently study paintings they'll be discussing later in class. If you were to try to manage the ebb and flow of each of these partition groups in bulk, you wouldn't decrease the chaos; you'd just make it so nobody ever got through the hallways at all. No, the only good system is the one that lets each chimpanzee browse at his or her own pace.

Most exhibits are enjoyed by EACH chimpanzee individually, and so the chimpanzees file by as they come. If a set of third graders and a set of first graders arrive at an exhibit at the same time, they'd file through the exhibit in whatever in or leave order happened by circumstance; that's ok, because, of course, within those partitions, the good little chimpanzee boys and girls were obeying the field trip rules; no first grader ever jumped ahead of the first grader it followed and no third grader ever jumped ahead of the third grader it followed.

Now, at some points during the field trip, the chimpanzees are to discuss an exhibit as a partition. When the first chimpanzee in that partition arrives at an exhibit, the exhibit's Operator will ask her to stand to the side and direct each chimpanzee in the partition to gather behind her. When, at some point, the docent shows up (last in line because of the field trip rules), the Operator checks that everyone is there by counting the number of kids in the partition and checking against the count that the docent carries. With that ritual satisfied, the Operator conducts the `partitionQuery` Q&A session. Each student, thus enlightened as a group, is then sent along to the next exhibit in exactly the original partition order.

As you can see, the students are able to enjoy the exhibits in the museum singly or in groups without any more coordination than is necessary. However, at the end of the day, when it"s time to go home, a much higher level of commitment to safety is necessary. What happens when it's time to return is this: As each partition group files out of the museum, it gathers back at its original school bus. Just as in the group discussion, the

bus Operator notices when the docent shows up (signaling the end of the partition) and compares the actual to expected count. Once satisfied that the full partition is present, it signals the Master Batch Coordinator for that grade that all the bus's students are present. Once the Master Batch Coordinator has received the "ready" signal from all the buses in a grade, it signals all the bus Operators that they are approved to transport the students back to school. Finally, once safely back at school, each bus Operator radios the Master Batch Coordinator of their safe arrival, allowing the MBC to declare the field trip a success.

exactly once Processing

Storm ensures that every tuple within a Storm or Trident dataflow is handled reliably and correctly. The difficulty comes when your dataflow must interact with an external resource, like a database or perform a task *exactly once* like when performing a count or other aggregation. This harder guarantee is provided by Trident's transactional functionality. If you're coming from a traditional database world, don't reach too much into the word "transaction." First, what we're discussing here takes place entirely outside of and entirely separate from any native transactional guarantees by the database. Second, it only provides a form of eventual consistency, as you're about to see.

Trident's `partitionPersist` and `persistentAggregate` classes provide their operations the following guarantee when applying an operation to a partition within a batch. First, all batches before that one will have been processed successfully. Second, no batch after that one will have ever been attempted. In return, the Aggregator must promise that, given the same records and prior value, it will return an acceptably-identical result. (FOOTNOTE: Note, I said "acceptably-identical," not identical. For instance, if your dataflow annotated business news articles with the share prices of companies mentioned within, those prices may have changed from one attempt of a batch to the next. In this case, the "acceptably-identical" result would have mildly-diverging values.) Here's how that guarantee is used by a `persistentAggregate` storing counts into an external datastore.

The `persistentAggregate` functionality in Trident has multiple moving parts, all of them modular, allowing the same classes to be used in simpler form for operations like `partitionPersist`. Since the important thing is that you understand how a `persistentAggregate` works, not how to rewrite one, we're going to describe how its functional parts finally work together, not build it up piece by piece.

As individual aggregable records roll in, each is handed to the Aggregator for its group within the partition; if it's the first member of a group, the Aggregator is created and prepared. In the case of a Combiner Aggregator or a Reducer Aggregator, only the running value needs to be tracked; an Accumulating Aggregator may be buffering those values internally.

When the partition is complete and the `persistentAggregate` receives the commit signal from the MBC, it therefore has on hand the following: All group keys seen in this partition and for each of those keys, an Aggregator instance, fat from having consumed all the records in that group.

It now needs to retrieve the prior existing value, if any, from the backing store. For example, in the case of the simple counting aggregation, it needs only the primitive integer holding the previous batch's value. In the case where we accumulated a complex profile, it's a `HashMap` describing that profile.

(TODO: Perhaps more here …)

The `persistentAggregate` wrapper hands the cache map (the first backing layer) the full set of group keys in the given partition, requesting their value. Assuming things are behaving well, the cache map will have those values, hot and ready, in memory --- but of course, it may be missing some or all. The cache map, using exactly the same interface, asks the concrete backing store's `StateMap` for the values it lacks. (TODO: What does it send back when a value is missing?) The cache map accepts the results and proudly presents the full set of values back to the `persistentAggregate` wrapper. The wrapper then promptly finalizes the aggregated values. It then hands a map of group keys and their updated values back to the backing store. The cache map player stores all those values in its cache, possibly triggering the least-recently-used values to be discarded. The cache map then, in turn, hands the full set of values to the concrete datastore, which persists them to the external database. Note that only a fraction of values for any given partition are typically read from the database but that every value in a partition is written back.

Now, here's the clever part. The concrete datastore accepts what it's given, the actual value to store, and it returns what it's asked for, the value as of the last batch. But it stores within the record for a given group key the following things: the transaction ID of the current batch, the newly-updated value and the prior value that it was based on; let's call the values the "aligned" value and the "pre-aligned" value, respectively. At whatever later time it's asked to retrieve the value for that record, it demands to know the current transaction ID as well.

Now, let's think back to the transactional guarantee we described above. Suppose the record it retrieves has a transaction ID of 8 and the aligned transaction ID is 12. Great! The backing store knows that, although this record wasn't involved, batches 8, 9, 10 and 11 were all processed successfully. It then takes the aligned value from batch 8 and faithfully report it as the value to update.

(TODO: SIDEBAR: It could happen that the record it retrieves shows a transaction ID of 12. It might be that this worker is retrying an earlier failed attempt, it might be that this worker fell off the grid and it's seeing the result of the retry due to its laziness.)

It might be, as described in the sidebar, that the transaction ID is 12. Remember, the request is for the value prior to the current batch; luckily, that's exactly what was stored in the pre-aligned value and so that's what is returned. Now, you see why it's important that the Aggregator promises acceptably-identical results, given the same records and prior value; you're not allowed to care which attempt of a retry is the last one to complete.

This is all done behind the scenes and you never have to worry about it. In fact, the class that hides this transactional behavior is called the `opaquevalue` class and this type of dataflow is what Trident calls an `OpaqueTransactional` topology.

For folks coming from a traditional database background, please notice that while we use the word "transactional" here, don't read too much into that. First, we're not using and not relying on any native transactional guarantee in the commit to the external database. The transactional behavior we've described covers the entire progress of a batch, not just the commit of any given partition to the database. Second, the transactional behavior is only eventually consistent. In practice, since the Master Batch Coordinator signals all `persistentAggregate` to commit simultaneously, there is very little jitter among attempts to commit. If your database administrator is doing her job, in normal circumstances, an external read will not notice misaligned batches.

Of course, all this machinery was put in place to tolerate the fact that sometimes, a subset of workers might hang or die trying to commit their partition within a batch. In that case, a read of the database would return some values current as of, say, batch 12, while (until the retry happens) the failed workers' records are only up to date as of batch 11.

Walk-through of the Github dataflow

Let's walk through the batch lifecycle using the Github dataflow from the Intro to Storm chapter (TODO: ref).

(NOTE: TODO: In chapter on Trident tuning, make clear that we are not talking about Storm tuning and some of our advice, especially around `maxspoutpending` will be completely inappropriately for a pure Storm flow.)

(TODO: Missing Section. This is covered somewhat above, but we need to either specifically do a walkthrough of Github, or wind it into what comes)

CHAPTER 25
Storm+Trident Tuning

Outline

Our approach for Storm+Trident, as it is for Hadoop, is to use it from the "outside". If you find yourself seeking out details of its internal construction to accomplish a task, stop and consider whether you're straying outside its interface. But once your command of its external model solidifies and as your application reaches production, it's worth understanding what is going on. This chapter describes the lifecycle of first, a Storm +Trident tuple, and next,

Along the way, we can also illuminate how Storm+Trident acheives its breakthroughs — its ability to do exactly-once processing is a transformative improvement over transfer-oriented systems (such as Flume or S4) and wall street-style Complex Event Processing (CEP) systems (such as Esper).

Because Storm+Trident has not reached Hadoop's relative maturity, it's important for the regular data scientist to under

- Tuning constraints — refer to earlier discussion on acceptable delay, thruput, horizon of compute, et
- Process for initial tuning
- General recommendations
- cloud tuning

Tuning a dataflow system is easy:

```
The Dataflow Tuning Rules:
* Ensure each stage is always ready to accept records, and
* Deliver each processed record promptly to its destination
```

That may seem insultingly simplistic, but most tuning questions come down to finding some barrier to one or the other. It also implies a corollary: once your dataflow does

obey the Dataflow Tuning Rules, stop tuning it. Storm+Trident flows are not subtle: they either work (in which case configuration changes typically have small effect on performance) or they are catastrophically bad. So most of the tuning you'll do is to make sure you don't sabotage Storm's ability to meet the Dataflow Tuning Rules

Goal

First, identify your principal goal and principal bottleneck.

The goal of tuning here is to optimize one of latency, throughput, memory or cost, without sacrificing sabotaging the other measures or harm stability.

Next, identify your dataflow's principal bottleneck, the constraining resource that most tightly bounds the performance of its slowest stage. A dataflow can't pass through more records per second than the cumulative output of its most constricted stage, and it can't deliver records in less end-to-end time than the stage with the longest delay.

The principal bottleneck may be:

- *IO volume*: a hardware bottleneck for the number of bytes per second that a machine's disks or network connection can sustain. For example, event log processing often involves large amounts of data, but only trivial transformations before storage.
- *CPU*: by contrast, a CPU-bound flow spends more time in calculations to process a record than to transport that record.
- *memory*: large windowed joins or memory-intensive analytics algorithms will in general require you to provision each machine for the largest expected memory extent. Buying three more machines won't help: if you have a 10 GB window, you need a machine with 10 GB+ of RAM.
- *concurrency*: If your dataflow makes external network calls, you will typically find that the network request latency for each record is far more than the time spent to process and transport the record. The solution is to parallelize the request by running small batches and and high parallelism for your topology. However, increasing parallelism has a cost: eventually thread switching, per-executor bookkeeping, and other management tasks will consume either all available memory or CPU.
- *remote rate limit*: alternatively, you may be calling an external resource that imposes a maximum throughput limit. For example, terms-of-service restrictions from a third-party web API might only permit a certain number of bulk requests per hour, or a legacy datastore might only be able to serve a certain volume of requests before its performance degrades. If the remote resource allows bulk requests, you should take care that each Trident batch is sized uniformly. For example, the twitter users lookup API returns user records for up to 100 user IDs — so it's essential that each

batch consist of 100 tuples (and no more). Otherwise, there's not much to do here besides tuning your throughput to comfortably exceed the maximum expected rate.

For each of the cases besides IO-bound and CPU-bound, there isn't that much to say: for memory-bound flows, it's "buy enough RAM" (though you should read the tips on JVM tuning later in this chapter). For remote rate-limited flows, buy a better API plan or remote datastore — otherwise, tune as if CPU-bound, allowing generous headroom.

For concurrency-bound flows, apply the recommendations that follow, increase concurrency until things get screwy. If that per-machine throughput is acceptable to your budget, great; otherwise, hire an advanced master sysadmin to help you chip away at it.

Provisioning

Unless you're memory-bound, the Dataflow Tuning Rules imply network performance and multiple cores are valuable, but that you should not need machines with a lot of RAM. Since tuples are handled and then handed off, they should not be accumulating in memory to any significant extent. Storm works very well on commodity hardware or cloud/virtualized machines. Buy server-grade hardware, but don't climb the price/performance curve.

The figures of merit here are the number and quality of CPU cores, which govern how much parallelism you can use; the amount of RAM per core, which governs how much memory is available to each parallel executor chain; and the cost per month of CPU cores (if CPU-bound) or cost per month of RAM (if memory-bound. Using the Amazon cloud machines as a reference, we like to use either the `c1.xlarge` machines (7GB ram, 8 cores, $424/month, giving the highest CPU-performance-per-dollar) or the `m3.xlarge` machines (15 GB ram, 4 cores, $365/month, the best balance of CPU-per-dollar and RAM-per-dollar). You shouldn't use fewer than four worker machines in production, so if your needs are modest feel free to downsize the hardware accordingly.

Topology-level settings

Use one worker per machine for each topology: since sending a tuple is much more efficient if the executor is in the same worker, the fewer workers the better. (Tuples go directly from send queue to receive queue, skipping the worker transfer buffers and the network overhead). Also, if you're using Storm pre-0.9, set the number of ackers equal to the number of workers — previously, the default was one acker per topology.

The total number of workers per machine is set when the supervisor is launched — each supervisor manages some number of JVM child processes. In your topology, you specify how many worker slots it will try to claim.

In our experience, there isn't a great reason to use more than one worker per topology per machine. With one topology running on those three nodes, and parallelism hint 24 for the critical path, you will get 8 executors per bolt per machine, i.e. one for each core. This gives you three benefits.

The primary benefit is that when data is repartitioned (shuffles or group-bys) to executors in the same worker, it will not have to hit the transfer buffer — tuples will be directly deposited from send to receive buffer. That's a big win. By contrast, if the destination executor were on the same machine in a different worker, it would have to go send → worker transfer → local socket → worker recv → exec recv buffer. It doesn't hit the network card, but it's not as big a win as when executors are in the same worker.

Second, you're typically better off with three aggregators having very large backing cache than having twenty-four aggregators having small backing caches. This reduces the effect of skew, and improves LRU efficiency.

Lastly, fewer workers reduces control flow chatter.

For CPU-bound stages, set one executor per core for the bounding stage (If there are many cores, uses one less). Using the examples above, you would run parallelism of 7 or 8 on a `c1.xlarge` and parallism of 4 on an `m3.xlarge`. Don't adjust the parallelism unless there's good reason — even a shuffle implies network transfer, and shuffles don't impart any load-balancing. For memory-bound stages, set the parallelism to make good use of the system RAM; for concurrency-bound stages, find the parallelism that makes performance start to degrade and then back off to say 80% of that figure.

Match your spout parallelism to its downstream flow. Use the same number of kafka partitions as kafka spouts (or a small multiple). If there are more spouts than kafka machines*kpartitions, the extra spouts will sit idle.

Map states or persistentAggregates accumulate their results into memory structures, and so you'll typically see the best cache efficiency and lowest bulk-request overhead by using one such stage per worker.

Initial tuning

If you have the ability to specify your development hardware, start tuning on a machine with many cores and over-provisioned RAM so that you can qualify the flow's critical bottleneck. A machine similar to Amazon's `m3.2xlarge` (30 GB ram, 8 cores) lets you fall back to either of the two reference machines described above.

For a CPU-bound flow:

- Construct a topology with parallelism one
- set max-pending to one, use one acker per worker, and ensure that storm's `no files` ulimit is large (65000 is a decent number).

- Set the trident-batch-delay to be comfortably larger than the end-to-end latency — there should be a short additional delay after each batch completes.
- Time the flow through each stage.
- Increase the parallelism of CPU-bound stages to nearly saturate the CPU, and at the same time adjust the batch size so that state operations (aggregates, bulk database reads/writes, kafka spout fetches) don't slow down the total batch processing time.
- Keep an eye on the GC activity. You should see no old-gen or STW GCs, and efficient new-gen gcs (your production goal no more than one new-gen gc every 10 seconds, and no more than 10ms pause time per new-gen gc, but for right now just overprovision — set the new-gen size to give infrequent collections and don't worry about pause times).

Once you have roughly dialed in the batch size and parallelism, check in with the First Rule. The stages upstream of your principal bottleneck should always have records ready to process. The stages downstream should always have capacity to accept and promptly deliver processed records.

Sidebar: Little's Law

Little's Law is a simple but very useful formula to keep in mind. It says that for any flow,

```
`Capacity (records in system) = Throughput (records/second) / Latency (seconds to pass through)`
```

This implies that you can't have better throughput than the collective rate of your slowest stage, and you can't have better latency than the sum of the individual latencies.

For example, if all records must pass through a stage that handles 10 records per second, then the flow cannot possibly proceed faster than 10 records per second, and it cannot have latency smaller than 100ms (1/10 second).

What's more, with 20 parallel stages, the 95th percentile latency — your slowest stage — becomes the median latency of the full set. (TODO: nail down numbers) Current versions of Storm+Trident don't do any load-balancing within batches, and so it's worth benchmarking each machine to ensure performance is uniform.

Batch Size

Next, we'll set the batch size.

Kafka Spout: Max-fetch-bytes

Most production deployments use the Kafka spout, which for architectural reasons does not allow you to specify a precise count of records per batch. Instead, the batch count

for the Kafka spout is controlled indirectly by the max fetch bytes. The resulting total batch size is at most `(kafka partitions) * (max fetch bytes)`.

For example, given a topology with six kafka spouts and four brokers with three kafka-partitions per broker, you have twelve kafka-partitions total, two per spout. When the MBCoordinator calls for a new batch, each spout produces two sub-batches (one for each kafka-partition), each into its own trident-partition. Now also say you have records of 1000 +/- 100 bytes, and that you set max-fetch-bytes to 100_000. The spout fetches the largest discrete number of records that sit within max-fetch-bytes — so in this case, each sub-batch will have between 90 and 111 records. That means the full batch will have between 1080 and 1332 records, and 1_186_920 to 1_200_000 bytes.

Choosing a value

In some cases, there is a natural batch size: for example the twitter `users/lookup` API call returns information on up to 100 distinct user IDs. If so, use that figure.

Otherwise, you want to optimize the throughput of your most expensive batch operation. `each()` functions should not care about batch size — batch operations like bulk database requests, batched network requests, or intensive aggregation (`partitionPersist`, `partitionQuery`, or `partitionAggregate`) do care.

Typically, you'll find that there are three regimes:

1. when the batch size is too small, the response time per batch is flat — it's dominated by bookeeping.
2. it then grows slowly with batch size. For example, a bulk put to elasticsearch will take about 200ms for 100 records, about 250ms for 1000 records, and about 300ms for 2000 records (TODO: nail down these numbers).
3. at some point, you start overwhelming some resource on the other side, and execution time increases sharply.

Since the execution time increases slowly in case (2), you get better and better records-per-second throughput. Choose a value that is near the top range of (2) but comfortably less than regime (3).

Executor send buffer size

Now that the records-per-batch is roughly sized, take a look at the disruptor queue settings (the internal buffers between processing stages).

As you learned in the storm internals chapter, each slot in the executor send buffer queue holds a single tuple. The worker periodically sweeps all its hanging records, dispatching them in bunches either directly into executor receive buffers (for executors in the same worker) or the worker transfer buffer (for remote executors). Let us highlight the im-

portant fact that the executor send queue contains *tuples*, while the receive/transfer queues contain *bunches of tuples*.

These are advanced-level settings, so don't make changes unless you can quantify their effect, and make sure you understand why any large change is necessary. In all cases, the sizes have to be an even power of two (1024, 2048, 4096, and so forth).

As long as the executor send queue is large enough, further increase makes no real difference apart from increased ram use and a small overhead cost. If the executor send queue is way too small, a burst of records will clog it unnecessarily, causing the executor to block. The more likely pathology is that if it is *slightly* too small, you'll get skinny residual batches that will make poor use of the downstream receive queues. Picture an executor that emits 4097 tuples, fast enough to cause one sweep of 4096 records and a second sweep of the final record — that sole record at the end requires its own slot in the receive queue.

Unfortunately, in current versions of Storm it applies universally so everyone has to live with the needs of the piggiest customer.

This is most severe in the case of a spout, which will receive a large number of records in a burst, or anywhere there is high fanout (one tuple that rapidly turns into many tuples).

Set the executor send buffer to be larger than the batch record count of the spout or first couple stages.

Garbage Collection and other JVM options

TODO: make this amenable for the non-dragonmaster

- New-gen size to 1000 MB (`-XX:MaxNewSize=1000m`). Almost all the objects running through storm are short-lived — that's what the First Rule of data stream tuning says — so almost all your activity is here.
- Apportions that new-gen space to give you 800mb for newly-allocated objects and 100mb for objects that survive the first garbage collection pass.
- Initial perm-gen size of 96m (a bit generous, but Clojure uses a bit more perm-gen than normal Java code would), and a hard cap of 128m (this should not change much after startup, so I want it to die hard if it does).
- Implicit old-gen size of 1500 MB (total heap minus new- and perm-gens) The biggest demand on old-gen space comes from long-lived state objects: for example an LRU counting cache or dedupe'r. A good initial estimate for the old-gen size is the larger of a) twice the old-gen occupancy you observe in a steady-state flow, or b) 1.5 times the new-gen size. The settings above are governed by case (b).

- Total heap of 2500 MB (`-Xmx2500m`): a 1000 MB new-gen, a 100 MB perm-gen, and the implicit 1500 MB old-gen. Don't use gratuitously more heap than you need — long gc times can cause timeouts and jitter. Heap size larger than 12GB is trouble on AWS, and heap size larger than 32GB is trouble everywhere.
- Tells it to use the "concurrent-mark-and-sweep" collector for long-lived objects, and to only do so when the old-gen becomes crowded.
- Enables that a few mysterious performance options
- Logs GC activity at max verbosity, with log rotation

If you watch your GC logs, in steady-state you should see

- No stop-the-world (STW) gc's — nothing in the logs about aborting parts of CMS
- old-gen GCs should not last longer than 1 second or happen more often than every 10 minutes
- new-gen GCs should not last longer than 50 ms or happen more often than every 10 seconds
- new-gen GCs should not fill the survivor space
- perm-gen occupancy is constant

Side note: regardless of whether you're tuning your overall flow for latency or throughput, you want to tune the GC for latency (low pause times). Since things like committing a batch can't proceed until the last element is received, local jitter induces global drag.

Tempo and Throttling

Max-pending (`TOPOLOGY_MAX_SPOUT_PENDING`) sets the number of tuple trees live in the system at any one time.

Trident-batch-delay (`topology.trident.batch.emit.interval.millis`) sets the maximum pace at which the trident Master Batch Coordinator will issue new seed tuples. It's a cap, not an add-on: if t-b-d is 500ms and the most recent batch was released 486ms, the spout coordinator will wait 14ms before dispensing a new seed tuple. If the next pending entry isn't cleared for 523ms, it will be dispensed immediately. If it took 1400ms, it will also be released immediately — but no make-up tuples are issued.

Trident-batch-delay is principally useful to prevent congestion, especially around start-up. As opposed to a traditional Storm spout, a Trident spout will likely dispatch hundreds of records with each batch. If max-pending is 20, and the spout releases 500 records per batch, the spout will try to cram 10,000 records into its send queue.

In general:

- number of workers a multiple of number of machines; parallelism a multiple of number of workers; number of kafka partitions a multiple of number of spout parallelism
- Use one worker per topology per machine
- Start with fewer, larger aggregators, one per machine with workers on it
- Use the isolation scheduler
- Use one acker per worker — [pull request #377](*https://github.com/nathanmarz/storm/issues/377*) makes that the default.

CHAPTER 26

Hbase Data Modeling

////Definitely add some additional introduction here. Describe how this connects with other chapters. Also characterize the overall goal of the chapter. And, then, put into context to the real-world (and even connect to what readers have learned thus far in the book). Amy////

Space doesn't allow treating HBase in any depth, but it's worth equipping you with a few killer dance moves for the most important part of using it well: data modeling. It's also good for your brain — optimizing data at rest presents new locality constraints, dual to the ones you've by now mastered for data in motion. ////If it's true that readers may get something crutial out of first playing with the tool before reading this chapter, say so here as a suggestion. Amy////So please consult other references (like "HBase: The Definitive Guide" (TODO:reference) or the free HBase Book online), load a ton of data into it, play around, then come back to enjoy this chapter.

Row Key, Column Family, Column Qualifier, Timestamp, Value

You're probably familiar with some database or another: MySQL, MongoDB, Oracle and so forth. These are passenger vehicles of various sorts, with a range of capabilities and designed for the convenience of the humans that use them. HBase is not a passenger vehicle — it is a big, powerful dump truck. It has no A/C, no query optimizer and it cannot perform joins or groups. You don't drive this dump truck for its ergonomics or its frills; you drive it because you need to carry a ton of raw data-mining ore to the refinery. Once you learn to play to its strengths, though, you'll find it remarkably powerful.

Here is most of what you can ask HBase to do, roughly in order of efficiency:

1. Given a row key: get, put or delete a single value into which you've serialized a whole record.
2. Given a row key: get, put or delete a hash of column/value pairs, sorted by column key.
3. Given a key: find the first row whose key is equal or larger, and read a hash of column/value pairs (sorted by column key).
4. Given a row key: atomically increment one or several counters and receive their updated values.
5. Given a range of row keys: get a hash of column/value pairs (sorted by column key) from each row in the range. The lowest value in the range is examined, but the highest is not. (If the amount of data is small and uniform for each row, the performance this type of query isn't too different from case (3). If there are potentially many rows or more data than would reasonably fit in one RPC call, this becomes far less performant.)
6. Feed a map/reduce job by scanning an arbitrarily large range of values.

That's pretty much it! There are some conveniences (versioning by timestamp, time-expirable values, custom filters, and a type of vertical partitioning known as column families); some tunables (read caching, fast rejection of missing rows, and compression); and some advanced features, not covered here (transactions, and a kind of stored procedures/stored triggers called coprocessors). For the most part, however, those features just ameliorate the access patterns listed above.

Features you don't get with HBase

Here's a partial list of features you do *not* get in HBase:

- efficient querying or sorting by cell value
- group by, join or secondary indexes
- text indexing or string matching (apart from row-key prefixes)
- arbitrary server-side calculations on query
- any notion of a datatype apart from counters; everything is bytes in/bytes out
- auto-generated serial keys

Sometimes you can partially recreate those features, and often you can accomplish the same *tasks* you'd use those features for, but only with significant constraints or tradeoffs. (You *can* pick up the kids from daycare in a dump truck, but only an idiot picks up their prom date in a dump truck, and in neither case is it the right choice).

More than most engineering tools, it's essential to play to HBase's strengths, and in general the simpler your schema the better HBase will serve you. Somehow, though, the sparsity of its feature set amplifies the siren call of even those few features. Resist, Resist. The more stoicly you treat HBase's small *feature* set, the better you will realize how surprisingly large HBase's *solution* set is.

Schema Design Process: Keep it Stupidly Simple

A good HBase data model is "designed for reads", and your goal is to make *one read per customer request* (or as close as possible). If you do so, HBase will yield response times on the order of 1ms for a cache hit and 10ms for a cache miss, even with billions of rows and millions of columns.

An HBase data mode is typically designed around multiple tables, each serving one or a small number of online queries or batch jobs. There are the questions to ask:

1. What query do you want to make that *must* happen at milliseconds speed?
2. There are a set of related queries or batch jobs — which would you like to be efficient?

If you are using it primarily for batch use,

1. What is the batch job you are most interested in simplifying?
2. There are a set of related queries or batch jobs — which would you like to be efficient?

Autocomplete API (Key-Value lookup)

Let's sketch the implementation of an autocomplete API on Wikipedia page titles, an example that truly plays to HBase's strengths. As a visitor types characters into a search bar, the browser will request a JSON-encoded list of the top 10 most likely completions for that prefix. Responsiveness is essential: at most 50 milliseconds end-to-end response time. Several approaches might spring to mind, like a range query on titles; a prefix query against a text search engine; or a specialized "trie" datastructure. HBase provides a much stupider, far superior solution.

Instead, we'll enumerate every possible completion. This blows the dataset into the billion-row range, but it makes each request a highly cache-efficient key/value lookup. Given an average title length of (TODO: insert numbers), the full completion set weighs in at "only" (TODO: numbers) rows and XXX raw data size — a walk in the park for HBase. (A sketch of how you might do this: first, use Pig to join on the pagerank table (see TODO: ref) to attach a "prominence" to each page. Next, write a map-reduce job

to group the prefixes. The mapper takes each title and emits the first three, four, five, up to say twelve characters along with the pagerank. Use the prefix as partition key, and the prefix-rank as a descending sort key. Within each prefix group, the first ten records will be the ten most prominent completions; store them as a JSON-ized list and ignore all following completions for that prefix.)

What will we store into HBase? Your first instinct might be to store each of the ten titles, each in its own cell. Reasonable, but still too clever. Instead, serialize the full JSON-encoded response as a single value. This minimizes the cell count (memory- and disk-efficient), lets the API front end put the value straight onto the wire (speed and lines-of-code efficient), and puts us in the most efficient access pattern: single row, single value.

Table 26-1. Autocomplete HBase schema

table	row key	column family	column qualifier	value	options
title_autocomp	`prefix`	*j*	-	JSON-encoded response	`VERSIONS => 1, BLOOMFILTER => 'ROW', COMPRESSION => 'SNAPPY'`

Help HBase be Lazy

In the autocomplete example, many requests will be for non-existent rows (eg "hdaoop"). These will of course be cache misses (there's nothing to cache), making the queries not just useless but also costly. Luckily, there's a spec ialized data structure known as a "Bloom Filter" that lets you very efficiently test set membership. If you explicitly enable it [1], HBase will capture all row keys into a Bloom Filter. On each request, it will quickly make sure it's worth trying to retrieve a value before doing so. Data blocks for lame prefixes (`hda...`) will be left unread, so that blocks for fecund prefixes (`had...`) can be kept in RAM.

Row Locality and Compression

There's another reason HBase is a great match for this problem: row locality. HBase stores all rows in sorted order on disk, so when a visitor has typed **`chim`**, the rows for `chime` and `chimp` and so forth are nearby on disk. Whatever next character the visitor types, the operating system is likely to have the right block hot in cache.

That also makes the autocomplete table especially well-suited for compression. Compression drives down the data size, which of course economizes disk capacity — more importantly, though, it means that the drive head has less data to seek past, and the IO

1. A bug in the HBase shell may interfere with your ability to specify a bloom filter in a schema — the HBASE-3086 bug report has a one-line patch that fixes it.

bus has less data to stream off disk. Row locality often means nearby data elements are highly repetitive (definitely true here), so you realize a great compression ratio. There are two tradeoffs: first, a minor CPU hit to decompress the data; worse though, that you must decompress blocks at a time even if you only want one cell. In the case of autocomplete, row locality means you're quite likely to use some of those other cells.

Geographic Data

For our next example, let's look at geographic data: the Geonames dataset of places, the Natural Earth dataset of region boundaries, and our Voronoi-spatialized version of the NCDC weather observations (TODO: ref).

We require two things. First, direct information about each feature. Here no magic is called for: compose a row key from the feature type and id, and store the full serialized record as the value. It's important to keep row keys *short* and *sortable*, so map the region types to single-byte ids (say, `a` for country, `b` for admin 1, etc) and use standard ISO identifiers for the region id (`us` for the USA, `dj` for Djibouti, etc).

More interestingly, we would like a "slippy map" (eg Google Maps or Leaflet) API: given the set of quadtiles in view, return partial records (coordinates and names) for each feature. To ensure a responsive user experience, we need low latency, concurrent access and intelligent caching — HBase is a great fit.

Quadtile Rendering

The boundaries dataset gives coordinates for continents, countries, states ("admin1"), and so forth. In (TODO: ref the Geographic Data chapter), we fractured those boundaries into quadtiles for geospatial analysis, which is the first thing we need.

We need to choose a base zoom level: fine-grained enough that the records are of manageable size to send back to the browser, but coarse-grained enough that we don't flood the database with trivial tiles ("In Russia". "Still in Russia". "Russia, next 400,000 tiles"…). Consulting the (TODO: ref "How big is a Quadtile") table, zoom level 13 means 67 million quadtiles, each about 4km per side; this is a reasonable balance based on our boundary resoluion.

```
ZL   recs    @64kB/qk        reference size
12     17 M      1 TB        Manhattan
13     67 M      4 TB
14    260 M     18 TB        about 2 km per side
15   1024 M     70 TB        about 1 km per side
```

For API requests at finer zoom levels, we'll just return the ZL 13 tile and crop it (at the API or browser stage). You'll need to run a separate job (not described here, but see the references (TODO: ref migurski boundary thingy)) to create simplified boundaries for each of the coarser zoom levels. Store these in HBase with three-byte row keys built

from the zoom level (byte 1) and the quadtile id (bytes 2 and 3); the value should be the serialized GeoJSON record we'll serve back.

Column Families

We want to serve several kinds of regions: countries, states, metropolitan areas, counties, voting districts and so forth. It's reasonable for a request to specify one, some combination or all of the region types, and so given our goal of "one read per client request" we should store the popular region types in the same table. The most frequent requests will be for one or two region types, though.

HBase lets you partition values within a row into "Column Families". Each column family has its own set of store files and bloom filters and block cache (TODO verify caching details), and so if only a couple column families are requested, HBase can skip loading the rest [2].

We'll store each region type (using the scheme above) as the column family, and the feature ID (`us`, `jp`, etc) as the column qualifier. This means I can

- request all region boundaries on a quadtile by specifying no column constraints
- request country, state and voting district boundaries by specifying those three column families
- request only Japan's boundary on the quadtile by specifying the column key `a:jp`

Most client libraries will return the result as a hash mapping column keys (combined family and qualifier) to cell values; it's easy to reassemble this into a valid GeoJSON feature collection without even parsing the field values.

Column Families Considered Less Awesome Than They Seem

HBase tutorials generally have to introduce column families early, as they're present in every request and when you define your tables. This unfortunately makes them seem far more prominent and useful than they really are. They should be used only when clearly required: they incur some overhead, and they cause some internal processes to become governed by the worst-case pattern of access among all the column families in a row. So consider first whether separate tables, a scan of adjacent rows, or just plain column qualifiers in one family would work. Tables with a high write impact shouldn't use more than two or three column families, and no table should use more than a handful.

2. many relational databases accomplish the same end wtih "vertical partitioning".

Access pattern: "Rows as Columns"

The Geonames dataset has 7 million points of interest spread about the globe.

Rendering these each onto quadtiles at some resolution, as we did above, is fine for slippy-map rendering. But if we could somehow index points at a finer resolution, developers would have a simple effective way to do "nearby" calculations.

At zoom level 16, each quadtile covers about four blocks, and its packed quadkey exactly fills a 32-bit integer; this seems like a good choice. We're not going to rendering all the ZL16 quadtiles though — that would require 4 billion rows.

Instead, we'll render each *point* as its own row, indexed by the row key `quadtile_id16-feature_id`. To see the points on any given quadtile, I just need to do a row scan from the quadkey index of its top left corner to that of its bottom right corner (both left-aligned).

```
012100-a
012100-b
012101-c
012102-d
012102-e
012110-f
012121-g
012121-h
012121-i
012123-j
012200-k
```

To find all the points in quadtile `0121`, scan from `012100` to `012200` (returning `a` through `j`). Scans ignore the last index in their range, so `k` is excluded as it should be. To find all the points in quadtile `012 121`, scan from `012121` to `012122` (returning `g`, `h` and `i`). Don't store the quadkeys as the base-4 strings that we use for processing: the efficiency gained by packing them into 16- or 32-bit integers is worth the trouble. The quadkey *12301230* is eight bytes as the string "12301230", two bytes as the 16-bit integer 27756.

When you are using this "Rows as Columns" technique, or any time you're using a scan query, make sure you set "scanner caching" on. It's an incredibly confusing name (it does not control a "Cache of scanner objects"). Instead think of it as "Batch Size", allowing may rows of data to be sent per network call.

Typically with a keyspace this sparse you'd use a bloom filter, but we won't be doing direct gets and so it's not called for here (Bloom Filters are not consulted in a scan).

Use column families to hold high, medium and low importance points; at coarse zoom levels only return the few high-prominence points, while at fine zoom levels they would return points from all the column families

Filters

There are many kinds of features, and some of them are distinctly more populous and interesting. Roughly speaking, geonames features

- `A` (XXX million): Political features (states, counties, etc)
- `H` (XXX million): Water-related features (rivers, wells, swamps,…)
- `P` (XXX million): Populated places (city, county seat, capitol, …)
- …
- `R` (): road, railroad, …
- `S` (): Spot, building, farm
- …

Very frequently, we only want one feature type: only cities, or only roads common to want one, several or all at a time.

You could further nest the feature codes. To do a scan of columns in a single get, need to use a ColumnPrefixFilter

http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/ColumnPrefixFilter.html

Access pattern: "Next Interesting Record"

The weatherstation regions table is most interesting of all.

map from weather station to quadkeys, pre-calculated map from observation to quadkeys, accumulate on tile

We want to serve boundaries out in tiles, but records are heavyweight.

if we store whole globe at ZL 14 (2 km blocks), 1kb record size becomes 275 GB data. Multiply by the hours in 50 years (50 * 365.25 * 24 = 438,000 hours = PB.

20,000 weather stations 1 M records = 50x data size; 10 TB becomes 0.5 PB.

```
0111230~~
011123100
011123101
011123102
011123103
01112311~
```

```
011123120
011123121
011123122
011123123
01112313~
...
011130~~~
```

Retrieve the *next existing tile*. It's a one-row operation, but we specify a range from specific tile to max tile ID.

The next tile is either the speific one with that key, or the first parent.

Note: next interesting record doesn't use bloom filter

To do a range on zoomed-out, do a range from

want to scan all cells in `011 123`. this means `011 123 000` to `011 123 ~~~`.

Table 26-2. Server logs HBase schema

table	row key	column family	column qualifier	value	options
region_info	`region_type-region_name`	*r*	(none)	serialized record	`VERSIONS => 1, COMPRESSION => 'SNAPPY'`
geonames_info	`geonames_id`	*i*	(none)	serialized record	`VERSIONS => 1, COMPRESSION => 'SNAPPY'`
tile_bounds	`quadkey`	(region type)	`region_id`	Geo-JSON encoded path	`VERSIONS => 1, COMPRESSION => 'SNAPPY'`
tile_places	`quadkey`	(feature class)	`geonames_id`	name	`VERSIONS => 1, COMPRESSION => 'SNAPPY'` (TODO: scanner caching)

Web Logs: Rows-As-Columns

The Virtues of Real Time Streaming

Hadoop was developed largely to process and analyze high-scale server logs for Nutch and Yahoo!. The recent addition of real-time streaming data tools like Storm+Kafka to the Hadoop/HBase ecosystem unlocks transformative new ways to see your data. It's not just that it's *real-time*; it's that its *multi-latency*. As long as you provision enough capacity, you can make multiple writes to the database (letting you "optimize for reads"); execute transactional requests against legacy datastores; ping YouTube or Twitter or other only-mostly-dependable external APIs; and much more. All of a sudden some of your most cumbersome or impractical batch jobs become simple, reliable stream decorators. From

where we stand, a best-of-class big data stack has *three* legs: Hadoop, one or more scalable databases, and multi-latency streaming analytics.

A high-volume website might have 2 million unique daily visitors, causing 100 M requests/day on average (4000 requests/second peak), and say 600 bytes per log line from 20-40 servers. Over a year, that becomes about 40 billion records and north of 20 terabytes of raw data. Feed that to most databases and they will crumble. Feed it to HBase and it will smile, belch and ask for seconds and thirds — which in fact we will. Designing for reads means aggressively denormalizing data, to an extent that turns the stomach and tests the will of traditional database experts. Use a streaming data pipeline such as Storm+Kafka or Flume, or a scheduled batch job, to denormalize the data.

Webserver log lines contain these fields: `ip_address`, `cookie` (a unique ID assigned to each visitor), `url` (the page viewed), and `referer_url` (the page they arrived from), `status_code` (success or failure of request) and `duration` (time taken to render page). We'll add a couple more fields as we go along.

Timestamped Records

We'd like to understand user journeys through the site:

(Here's what you should not do: use a row key of `timebucket-cookie`; see ???

The To sort the values in descending timestamp order, instead use a reverse timestamp: `LONG_MAX - timestamp`. (You can't simply use the negative of `timestamp` — since sorts are always lexicographic, `-1000` sorts *before* `-9999`.)

By using a row key of `cookie-rev_time`

- we can scan with a prefix of just the cookie to get all pageviews per visitor ever.
- we can scan with a prefix of the cookie, limit one row, to get only the most recent session.
- if all you want are the distinct pages (not each page *view*), specify versions = 1 in your request.
- In a map-reduce job, using the column key and the referring page url gives a graph view of the journey; using the column key and the timestamp gives a timeseries view of the journey.

Row Locality

Row keys determine data locality. When activity is focused on a set of similar and thus adjacent rows, it can be very efficient or very problematic.

Adjacency is good: Most of the time, adjacency is good (hooray locality!). When common data is stored together, it enables - range scans: retrieve all pageviews having the same path prefix, or a continuous map region. - sorted retrieval: ask for the earliest entry, or the top-k rated entries - space-efficient caching: map cells for New York City will be much more commonly referenced than those for Montana. Storing records for New York City together means fewer HDFS blocks are hot, which means the opeerating system is better able to cache those blocks. - time-efficient caching: if I retrieve the map cell for Minneapolis, I'm much more likely to next retrieve the adjacent cell for nearby St. Paul. Adjacency means that cell will probably be hot in the cache.

Adjacency is bad: if *everyone* targets a narrow range of keyspace, all that activity will hit a single regionserver and your wonderful massively-distributed database will limp along at the speed of one abused machine.

This could happen because of high skew: for example, if your row keys were URL paths, the pages in the `/product` namespace would see far more activity than pages under `laborday_2009_party/photos` (unless they were particularly exciting photos). Similarly, a phenomenon known as Benford's law means that addresses beginning with *1* are far more frequent than addresses beginning with *9* [3]. In this case, managed splitting (pre-assigning a rough partition of the keyspace to different regions) is likely to help.

Managed splitting won't help for timestamp keys and other monotonically increasing values though, because the focal point moves constantly. You'd often like to spread the load out a little, but still keep similar rows together. Options include:

- swap your first two key levels. If you're recording time series metrics, use `metric_name-timestamp`, not `timestamp-metric_name`, as the row key.
- add some kind of arbitrary low-cardinality prefix: a server or shard id, or even the least-significant bits of the row key. To retrieve whole rows, issue a batch request against each prefix at query time.

Timestamps

You could also track the most recently-viewed pages directly. In the `cookie_stats` table, add a column family `r` having `VERSIONS: 5`. Now each time the visitor loads a page, write to that exact value;

HBase store files record the timestamp range of their contained records. If your request is limited to values less than one hour old, HBase can ignore all store files older than that.

3. A visit to the hardware store will bear this out; see if you can figure out why. (Hint: on a street with 200 addresses, how many start with the numeral *1*?)

Domain-reversed values

It's often best to store URLs in "domain-reversed" form, where the hostname segments are placed in reverse order: eg "org.apache.hbase/book.html" for "hbase.apache.org/book.html". The domain-reversed URL orders pages served from different hosts within the same organization ("org.apache.hbase" and "org.apache.kafka" and so forth) adjacently.

To get a picture of inbound traffic

ID Generation Counting

One of the elephants recounts this tale:

In my land it's essential that every person's prayer be recorded.

One is to have diligent monks add a a grain of rice to a bowl on each event, then in daily ritual recount them from beginning to end. You and I might instead use a threadsafe [UUID](*http://en.wikipedia.org/wiki/Universally_unique_identifier*) library to create a guaranteed-unique ID.

However, neither grains of rice nor time-based UUIDs can easily be put in time order. Since monks may neither converse (it's incommensurate with mindfulness) nor own fancy wristwatches (vow of poverty and all that), a strict ordering is impossible. Instead, a monk writes on each grain of rice the date and hour, his name, and the index of that grain of rice this hour. You can read a great writeup of distributed UUID generation in Boundary's [Flake project announcement](*http://boundary.com/blog/2012/01/12/flake-a-decentralized-k-ordered-unique-id-generator-in-erlang/*) (see also Twitter's [Snowflake](*https://github.com/twitter/snowflake*)).

You can also "block grant" counters: a central server gives me a lease on

ID Generation Counting

HBase actually provides atomic counters

Another is to have an enlightened Bodhisattva hold the single running value in mindfulness.

http://stackoverflow.com/questions/9585887/pig-hbase-atomic-increment-column-values

From *http://www.slideshare.net/larsgeorge/realtime-analytics-with-hadoop-and-hbase*
--

1 million counter updates per second on 100 nodes (10k ops per node) Use a different column family for month, day, hour, etc (with different ttl) for increment

counters and TTLs — http://grokbase.com/t/hbase/user/119x0yjg9b/settimerange-for-hbase-increment

HBASE COUNTERS PART I

Atomic Counters

Second, for each visitor we want to keep a live count of times they've viewed each distinct URL. In principle, you could use the `cookie_url` table, Maintaining a consistent count is harder than it looks: for example, it does not work to read a value from the database, add one to it, and write the new value back. Some other client may be busy doing the same, and so one of the counts will be off. Without native support for counters, this simple process requires locking, retries, or other complicated machinery.

HBase offers *atomic counters*: a single `incr` command that adds or subtracts a given value, responding with the new value. From the client perspective it's done in a single action (hence, "atomic") with guaranteed consistence. That makes the visitor-URL tracking trivial. Build a table called `cookie_url`, with a column family `u`. On each page view:

1. Increment the counter for that URL: `count = incr(table: "cookie_url_count", row: cookie, col: "u:#{url}")`.

The return value of the call has the updated count. You don't have to initialize the cell; if it was missing, HBase will treat it as having had a count of zero.

Abusing Timestamps for Great Justice

We'd also like to track, for each visitor, the *most frequent* ("top-k") URLs they visit. This might sound like the previous table, but it's very different — locality issues typically make such queries impractical. In the previous table, all the information we need (visitor, url, increment) to read or write is close at hand. But you can't query that table by "most viewed" without doing a full scan; HBase doesn't and won't directly support requests indexed by value. You might also think "I'll keep a top-k leaderboard, and update it if the currently-viewed URL is on it" — but this exposes the consistency problem you were just warned about above.

There is, however, a filthy hack that will let you track the *single* most frequent element, by abusing HBase's timestamp feature. In a table `cookie_stats` with column family `c` having `VERSIONS: 1`. Then on each pageview,

1. As before, increment the counter for that URL: `count = incr(table: "cookie_url_count", row: cookie, col: "u:#{url}")`. The return value of the call has the updated count.

2. Store the URL in the `cookie_stats` table, but use a *timestamp equal to that URL's count* — not the current time — in your request: `put("cookie_stats", row: cookie, col: "c", timestamp: count, value: url)`.

To find the value of the most-frequent URL for a given cookie, do a `get(table: "cookie_stats", row: cookie, col: 'c')`. HBase will return the "most recent" value, namely the one with the highest timestamp, which means the value with the highest count. Although we're constantly writing in values with lower "timestamps" (counts), HBase ignores them on queries and eventually compacts them away.

For this hack to work, the value *must* be forever monotonically increasing (that is, never decrease). The value "total lifetime pageviews" can only go up; "pageviews in last 30 days" will go up or down over time

TTL (Time-to-Live) expiring values

////Consider, here, pointing out what the reader stands to gain, what they'll get out of the exercise in terms of learning how to use tools for real-world applications. Amy////

These high-volume tables consume significant space and memory; it might make sense to discard data older than say 60 days. HBase lets you set a "TTL" (time-to-live) on any column family; records whose timestamp is farther in the past than that TTL won't be returned in gets or scans, and they'll be removed at the next compaction (TODO: major or minor?) [4].

Exercises

1. Besides the pedestrian janitorial work of keeping table sizes in check, TTLs are another feature to joyfully abuse. Describe how you would use TTLs to track time-based rolling aggregates, like "average air-speed velocity over last 10 minutes".

Table 26-3. Server logs HBase schema

table	row key	family	qualifier	value	options
visits	`cookie-timebucket`	*r* (referer)	`referer`	-	
visits	`cookie-timebucket`	*s* (search)	`term`	-	
visits	`cookie-timebucket`	*p* (product)	`product_id`	-	
visits	`cookie-timebucket`	*z* (checkout)	`cart_id`	`{product_ids}`	
cookie_urls	`cookie`	*u* (url)	-		

4. The TTL will only work if you're playing honest with the timestamps — you can't use it with the most-frequent URL table

ip_tbs	ip-timebucket

IP Address Geolocation

An increasing number of websites personalize content for each reader. Retailers find that even something as simple as saying "Free Shipping" or "No Sales Tax" (each true only for people in certain geographic areas) dramatically increases sales. HBase's speed and simplicity shine for a high-stakes low-latency task like estimating the geographic location of a visitor based on their IP address

If you recall from (TODO ref server logs chapter), the Geo-IP dataset stores information about IP addresses a block at a time.

- *Fields*: IP address, ISP, latitude, longitude, quadkey
- *query*: given IP address, retrieve geolocation and metadata with very low latency

Table 26-4. IP-Geolocation lookup

table	row key	column families	column qualifiers	versions	value
ip	ip_upper_in_hex	field name	-	none	

Store the *upper* range of each IP address block in hexadecimal as the row key. To look up an IP address, do a scan query, max 1 result, on the range from the given ip_address to a value larger than the largest 32-bit IP address. A get is simply a scan-with-equality-max-1, so there's no loss of efficiency here.

Since row keys are sorted, the first value equal-or-larger than your key is the end of the block it lies on. For example, say we had block "A" covering `50.60.a0.00` to `50.60.a1.08`, "B" covering `50.60.a1.09` to `50.60.a1.d0`, and "C" covering `50.60.a1.d1` to `50.60.a1.ff`. We would store `50.60.a1.08 => {...A...}`, `50.60.a1.d0 => {...B...}`, and `50.60.a1.ff => {...C...}`. Looking up `50.60.a1.09` would get block B, because `50.60.a1.d0` is lexicographically after it. So would `50.60.a1.d0`; range queries are inclusive on the lower and exclusive on the upper bound, so the row key for block B matches as it should.

As for column keys, it's a tossup based on your access pattern. If you always request full rows, store a single value holding the serialized IP block metadata. If you often want only a subset of fields, store each field into its own column.

Wikipedia: Corpus and Graph

Table 26-5. Wikipedia HBase schema

table	row key	family	qualifier	value

articles	page_id	t	text	
article_versions	page_id	t	text	timestamp: updated_time
article_revisions	page_id-revision_id	v	text, user_id, comment	categories
category-page_id	c		redirects	bad_page_id

Graph Data

Just as we saw with Hadoop, there are two sound choices for storing a graph: as an edge list of `from,into` pairs, or as an adjacency list of all `into` nodes for each `from` node.

Table 26-6. HBase schema for Wikipedia pagelink graph: three reasonable implementations

table	row key	column families	column qualifiers	value	options
page_page	from_page-into_page	l (link)	(none)	(none)	bloom_filter: true
page_links	from_page	l (links)	into_page	(none)	page_links_ro

If we were serving a live wikipedia site, every time a page was updated I'd calculate its adjacency list and store it as a static, serialized value.

For a general graph in HBase, here are some tradeoffs to consider:

- The pagelink graph never has more than a few hundred links for each page, so there are no concerns about having too many columns per row. On the other hand, there are many celebrities on the Twitter "follower" graph with millions of followers or followees. You can shard those cases across multiple rows, or use an edge list instead.
- An edge list gives you fast "are these two nodes connected" lookups, using the bloom filter on misses and read cache for frequent hits.
- If the graph is read-only (eg a product-product similarity graph prepared from server logs), it may make sense to serialize the adjacency list for each node into a single cell. You could also run a regular map/reduce job to roll up the adjacency list into its own column family, and store deltas to that list between rollups.

Refs

- I've drawn heavily on the wisdom of HBase Book
- Thanks to Lars George for many of these design guidelines, and the "Design for Reads" motto.
- HBase Shell Commands

- HBase Advanced Schema Design by Lars George
- *http://www.quora.com/What-are-the-best-tutorials-on-HBase-schema*
- encoding numbers for lexicographic sorting:
  - an insane but interesting scheme: *http://www.zanopha.com/docs/elen.pdf*
  - a Java library for wire-efficient encoding of many datatypes: *https://github.com/mrflip/orderly*
- *http://www.quora.com/How-are-bloom-filters-used-in-HBase*

CHAPTER 27

Appendix

Appendix 1: Acquiring a Hadoop Cluster

- Elastic Map-Reduce
- Brew
- Amazon
- CDH and HortonWorks
- MortarData and TreasureData

Appendix 2: Cheatsheets

…

Appendix 3: Overview of Example Scripts and Datasets

…

Author

Philip (flip) Kromer is cofounder of Infochimps, a big data platform that makes acquiring, storing and analyzing massive data streams transformatively easier. Infochimps became part of Computer Sciences Corporation in 2013, and their big data platform now serves customers such as Cisco, HGST and Infomart. He enjoys Bowling, Scrabble, working on old cars or new wood, and rooting for the Red Sox.

Graduate School, Dept. of Physics - University of Texas at Austin, 2001-2007 Bachelor of Arts, Computer Science - Cornell University, Ithaca NY, 1992-1996

- Cofounder of Infochimps, now Head of Technology and Architecture at Infochimps, a CSC Company.
- Core committer for Storm, a framework for scalable stream processing and analytics
- Core committer for Ironfan, a framework for provisioning complex distributed systems in the cloud or data center
- Original author and core committer of Wukong, the leading Ruby library for Hadoop
- Contributed chapter to *The Definitive Guide to Hadoop* by Tom White

Dieterich Lawson is a recent graduate of Stanford University.

TODO DL: biography

About the Author

Colophon

Writing a book with O'Reilly is magic in so many ways, but none moreso than their Atlas authoring platform. Rather than the XML hellscape that most publishers require, Atlas allows us to write simple text documents with readable markup, do a `git push`, and see a holy-smokes-that-looks-like-a-real-book PDF file moments later.

- Emacs, because a text editor that isn't its own operating system might as well be edlin.
- Visuwords.com give good word when brain not able think word good.
- Posse East Bar and Epoch Coffee House in Austin provided us just the right amount of noise and focus for cranking out content.
- *http://dexy.it* is a visionary tool for writing documentation. With it, we are able to directly use the runnable example scripts as the code samples for the book.

License

TODO: actual license stuff

Text and assets are released under CC-BY-NC-SA (Creative Commons Attribution, Non-commercial, derivatives encouraged but Share Alike)

> This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit *http://creativecommons.org/licenses/by-nc-sa/3.0/* or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Code is Apache licensed unless specifically labeled otherwise.

For access to the source code, visit *http://github.com/infochimps-labs/big_data_for_chimps/*

Open Street Map

Some map images taken from Open Street Map, via Stamen's wonderful "Map → Image" tool.

Glossary

- secondarynn (aka “secondary namenode”) — handles compaction of namenode directory. It is NOT a backup for the namenode.
- support daemon — one of namenode, datanode, jobtracker, tasktracker or secondarynn.
- job process (aka “child process”) — the actual process that executes your code
- tasktracker — interface between jobracker and task processes. It does NOT execute your code, and typically requires a minimal amount of RAM.
- shuffle merge — (aka shuffle and sort)
- shuffle buffer --
- map sort buffer --
- Resident memory (RSS or RES) --
- JVM --
- JVM heap size --
- Old-Gen heap --
- New-Gen heap — portion of JVM ram used for short-lived objects. If too small, transient objects will go into the old-gen, causing fragmentation and an eventual STW garbage collection.
- garbage collection --
- STW garbage collection — "Stop-the-world” garbage collection, a signal that there has been significant fragmentation or heap pressure. Not Good.
- attempt ID --
- task ID --

- job ID --

Questions:

- “task process” or “job process” for child process?

Storm+Trident Glossary

Storm Execution

- Worker
- Daemons
  - — Nimbus
  - — UI
  - — Supervisor
  - — Child Process
- Executor
- Task
- Trident Function

Trident

- Tuple / TridentTuple
- tupletree
- Batch
- Partition
- Group
- Batch (Transaction) ID
- TridentOperation
- TridentFunction
- Aggregator *

Layout

- Topology
- Stream
- Assembly *

Internal

- Master Batch Coordinator
- Spout Coordinator
- TridentSpoutExecutor
- TridentBoltExecutor

Transport

- DisruptorQueue
- Executor Send Queu
- Executor Receive Queue
- Worker Transfer Queue
- (Worker Receive Buffer)

References

Other Hadoop Books

- Hadoop the Definitive Guide, Tom White
- Hadoop Operations, Eric Sammer
- Hadoop In Practice (Alex Holmes)
- Hadoop Streaming FAQ
- Hadoop Configuration defaults — mapred

Unreasonable Effectiveness of Data

- Peter Norvig's Facebook Tech Talk
- Later version of that talk at ?? — I like the
- "On the Unreasonable effectiveness of data"

Source material

- Wikipedia article on Lexington, Texas (CC-BY-SA)
- Installing Hadoop on OSX Lion
- JMX through a ssh tunnel

To Consider

- *http://www.cse.unt.edu/~rada/downloads.html — Texts* semantically annotated with WordNet 1.6 senses (created at Princeton University), and automatically mapped to WordNet 1.7, WordNet 1.7.1, WordNet 2.0, WordNet 2.1, WordNet 3.0

Code Sources

- wp2txt, by Yoichiro Hasebe

the git-scribe toolchain was very useful creating this book. Instructions on how to install the tool and use it for things like editing this book, submitting errata and providing translations can be found at that site.