Big Data for Chimps

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

Big Data for Chimps
by

Revision History for the :

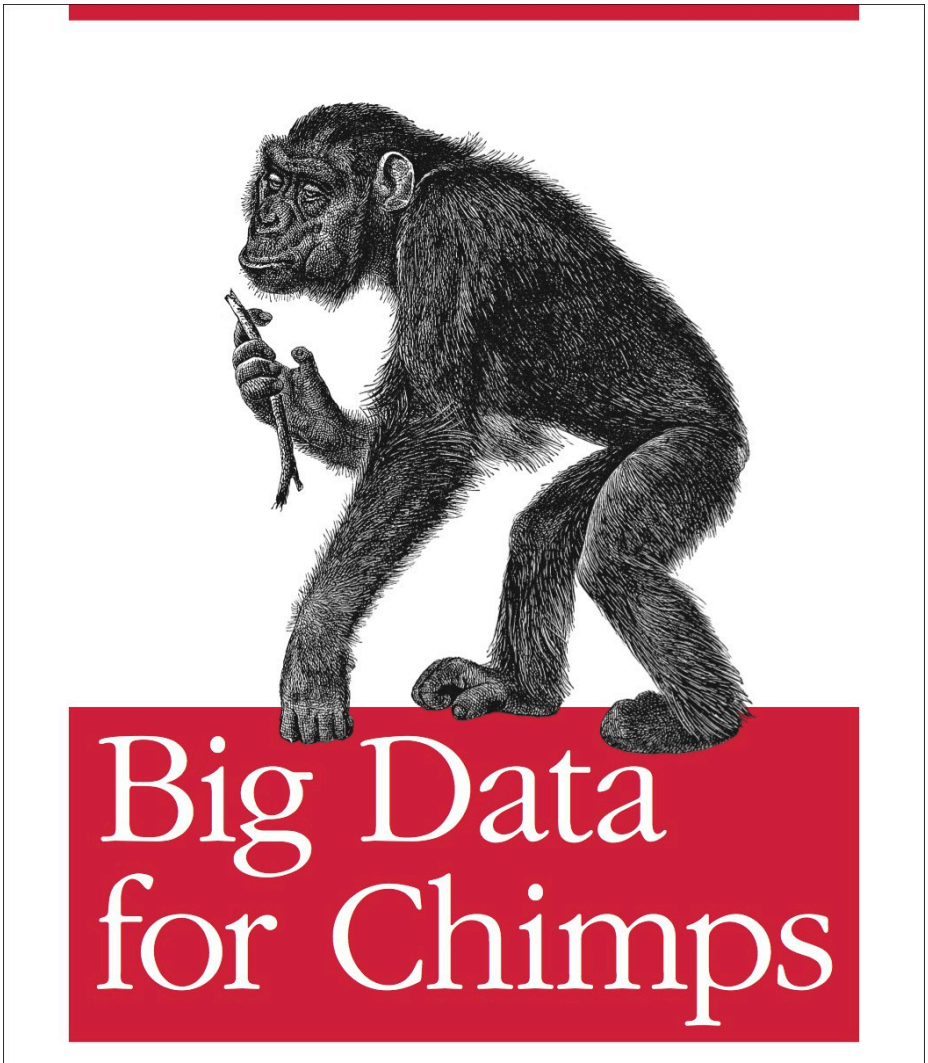
See *http://oreilly.com/catalog/errata.csp?isbn=* for release details.

Table of Contents

Part I. Big Data for Chimps

PART I

Big Data for Chimps



CHAPTER 1

Hello, Early Releasers

Hello and Thanks, Courageous and Farsighted Early Released-To'er! I want to make sure the book delivers value to you now, and rewards your early confidence by becoming the book you're proud to own.

My Questions for You

- The rule of thumb I'm using on introductory material is "If it's well-covered on the internet, leave it out". I hate it when tech books give a topic the bus-tour-of-London ("On your window to the left is the outside of the British Museum!") treatment, but you should never find yourself completely stranded. Please let me know if that's the case.
- Analogies: We'll be accompanied on part of our journey by Chimpanzee and Elephant, whose adventures are surprisingly relevant to understanding the internals of Hadoop. I don't want to waste your time laboriously remapping those adventures back to the problem at hand, but I definitely don't want to get too cute with the analogy. Again, please let me know if I err on either side.

Probable Contents

This is the plan. We'll roll material out over the next few months. Should we find we need to cut things (I hope not to), I've flagged a few chapters as *(bubble)*.

- **First Exploration**: A walkthrough of problem you'd use Hadoop to solve, showing the workflow and thought process. Hadoop asks you to write code poems that compose what we'll call *transforms* (process records independently) and *pivots* (restructure data).
- **Transform-only Job**: Chimpanzee and Elephant are hired to translate the works of Shakespeare to every language; you'll take over the task of translating text to Pig Latin. This is an "embarrasingly parallel" problem, so we can learn the mechanics of launching a job and a coarse understanding of the HDFS without having to think too hard.
  - — Chimpanzee and Elephant start a business
  - — Pig Latin translation
  - — Your first job: test at commandline
  - — Run it on cluster
  - — Input Splits
  - — Why Hadoop I: Simple Parallelism

- **Transform-Pivot Job**: C&E help SantaCorp optimize the Christmas toymaking process, demonstrating the essential problem of data locality (the central challenge of Big Data). We'll follow along with a job requiring map and reduce, and learn a bit more about Wukong (a Ruby-language framework for Hadoop).
  - — Locality: the central challenge of distributed computing
  - — The Hadoop Haiku
- **Server Log Processing**:
  - — Parsing logs and using regular expressions
  - — Histograms and time series of pageviews
  - — Geolocate visitors based on IP
  - — (Ab)Using Hadoop to stress-test your web server
- **Semi-Structured Data**: The dirty art of data munging. It's a sad fact, but too often the bulk of time spent on a data exploration is just getting the data ready. We'll show you street-fighting tactics that lessen the time and pain. Along the way, we'll prepare the datasets to be used throughout the book:
  - — Wikipedia Articles: Every English-language article (12 million) from Wikipedia.
  - — Wikipedia Pageviews: Hour-by-hour counts of pageviews for every Wikipedia article since 2007.
  - — US Commercial Airline Flights: every commercial airline flight since 1987
  - — Hourly Weather Data: a century of weather reports, with hourly global coverage since the 1950s.
  - — "Star Wars Kid" weblogs: large collection of apache webserver logs from a popular internet site (Andy Baio's waxy.org).
- **Data Models & Data Formats**: How to design your data models, transmit their contents, and organize their whole.
- **How to Think**: there are several design patterns for how to pivot your data, like Message Passing (objects send records to meet together); Set Operations (group, distinct, union, etc); Graph Operations (breadth-first search). Taken as a whole, they're equivalent; with some experience under your belt it's worth learning how to fluidly shift among these different models.
- **Geographic Data**:
  - — C&E
- **Why Hadoop**: Why and when is Hadoop called for? What changed about the world that we weren'talways doing it this way?
- **Statistics**:
  - — Sampling responsibly: it's harder and more important than you think

— Statistical aggregates and the danger of large numbers

— Histograms and Percentiles

- **Graph Processing**:
  — Community Extraction: Use the page-to-page links in Wikipedia to identify similar documents
  — Pagerank (centrality): Reconstruct pageview paths from web logs, and use them to identify important pages
  — *(bubble)*
- **Text Processing**: We'll show how to combine powerful existing libraries with hadoop to do effective text handling and Natural Language Processing:
  — Indexing documents
  — Tokenizing documents using Lucene
  — Pointwise Mutual Information
  — *(bubble)* Minhashing to combat a massive feature space
  — *(bubble)* How to cheat with Bloom filters
  — *(bubble)* Topic extraction using (to be determined)
- **Machine Learning for Busy Folks with Things to Do**: We'll combine the record of every commercial flight since 1987 with the hour-by-hour weather data to predict flight delays using
  — Naive Bayes
  — Logistic Regression
  — Random Forest (using Mahout) We'll equip you with a picture of how they work, but won't go into the math of how or why. We will show you how to choose a method, and how to cheat to win.
- **Hadoop Internals**:
  — Tuning for the Wise and Lazy
  — Tuning for the Brave and Foolish
  — The USE Method for understanding performance and diagnosing problems
- **Advanced Pig**:
  — Specialized joins that can dramatically speed up (or make feasible) your data transformations
  — Pig UDFs (User-Defined Functions) to extend Pig's capabilities
- **Cheatsheets**:
  — Regular Expressions

— Sizes of the Universe

— Hadoop Tuning & Configuration Variables

Not Contents

I'm not currently planning to cover Hive — I believe 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".

Other things I don't plan to include:

- Installing or maintaining Hadoop
- HBase (or other databases)
- Other map-reduce-like platforms (disco, spark, etc), or other frameworks (MrJob, Scalding, Cascading)
- Stream processing with Trident. (A likely sequel should this go well?)
- 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.

CHAPTER 2
About

What this book covers

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

It contains

- Detailed example programs applying Hadoop to interesting problems in context
- Advice and best practices for efficient software development
- 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.

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.

This is not a beginner's book. The emphasis on simplicity and fun should make it especially appealing to beginners, but this is not an approach you'll outgrow. The emphasis is on simplicity and fun because it's the most powerful approach, and generates the most value, for creative analytics: humans are important, robots are cheap. The code you see is adapted from programs we write at Infochimps. There are sections describing how and when to integrate custom components or extend the toolkit, but simple high-level transformations meet almost all of our needs.

Most 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

You should be familiar with at least one programming language, but it doesn't have to be Ruby. Ruby is a very readable language, and the code samples provided should map cleanly to languages like Python or Scala. Familiarity with SQL will help a bit, but isn't essential.

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.

All of the code in this book will run unmodified on your laptop computer and on an industrial-strength Hadoop cluster (though you will want to use a reduced data set for the laptop). You do need a Hadoop installation of some sort, even if it's a single machine. While a multi-machine cluster isn't essential, you'll learn best by spending some time on a real environment with real data. Appendix (TODO: ref) describes your options for installing Hadoop.

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.

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.

CHAPTER 3

Hello, Reviewers

I work somehow from the inside out — generate broad content, fill in, fill in, and asymptotically approach coherency. So you will notices sentences that stop in the

I'm endeavoring to leave fewer of these in chapters that hit the preview version, and to fill in the existing ones.

Controversials

I'd love feedback on a few decisions.

Sensible but nonstandard terms: I want to flatten the learning curve for folks who have great hopes of never reading the source code or configuring Hadoop's internals. So where technical hadoop terms are especially misleading, I'm using an isomorphic but non-standard one (introducing it with the technical term, of course). For example, I refer to the "early sort passes" and "last sort pass", rather than the misleading "shuffle" and "sort" terms from the source code.

On the one hand, I know from experience that people go astray with those terms: far more sorting goes on during the shuffle phase than the sort phase. On the other hand, I don't want to leave them stranded with idiosyncratic jargon. Please let me know where I've struck the wrong balance.

- "early sort passes" vs "shuffle phase"
- "last sort pass" vs "sort phase"
- "commit/output" for "commit".

- for configuration options, use the standardized names from wukong (eg `midflight_compress_codec`, `midflight_compress_on` and `output_compress_codec` for `mapred.map.output.compression.codec`, `mapred.compress.map.output` and `mapred.output.compression.codec`).

Vernacular Ruby? or Friendly to non-natives Ruby?: I'm a heavy Ruby user, but I also believe it's the most readable language available. I want to show people the right way to do things, but some of its idioms can be distracting to non-native speakers.

```
# Vernacular                      # Friendly

def bork(xx, yy=nil)              def bork(xx, yy=nil)
  yy ||= xx                         yy = xx if (not yy)
  xx * yy                           return xx * yy
end                               end

items = list.map(&:to_s)          items = list.map{|el| el.to_s }
```

My plan is to use vernacular ruby — with the one exception of providing `return` statements. I'd rather annoy rubyists than visitors, so please let me know what idioms seem opaque, and whether I should explain or eliminate them.

output directories with extensions: If your job outputs tsv files, it will create directory of TSV files with names like `part-00000`. Normally, we hang extensions off the file and never off the directory. However, in Hadoop you don't name those files; and you treat that directory itself as the unit of processing. I've always been on the fence, but now lean towards `/data/results/wikipedia/full/pagelinks.tsv`: you can use the same name in local or hadoop mode; it's advisory; and as mentioned it's the unit of processing.

Style Nits

CHAPTER 4
First Exploration (ch. A)

Hadoop is a remarkably powerful tool for processing data, giving us at long last mastery over massive-scale distributed computing. More than likely, that's how you came to be reading this paragraph.

What you might not yet know is that Hadoop's power comes from *embracing*, not conquering, the constraints of distributed computing. This exposes a core simplicity that makes programming it exceptionally fun.

Hadoop's bargain is this. You must give up fine-grained control over how data is read and sent over the network. Instead, you write a series of short, constrained transformations, a sort of programming Haiku:

```
Data flutters by
Elephants make sturdy piles
Insight from context
```

For any such program, Hadoop's diligent elephants intelligently schedule the tasks across ones or dozens or thousands of machines. They attend to logging, retry and error handling; distribute your data to the workers that process it; handle memory allocation, partitioning and network routing; and attend to myriad other details that otherwise stand between you and insight. Putting these constraints on how you ask your *question* releases constraints that traditional database technology placed on your *data*. Unlocking access to data that is huge, unruly, organic, highly-dimensional and deeply connected unlocks answers to a new deeper set of questions about the large-scale behavior of humanity and our universe. <remark>too much?? pk4</remark>

Where is Barbecue?

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 Chapter 19 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 Texas Memorial Stadium (where the Texas Longhorns football team plays), Snow's BBQ (proclaimed "The Best Texas BBQ in the World") and the TACC (Texas Advanced Computer Center, the largest academic supercomputer to date).

Since the birth of Artificial Intelligence we've wished we could quantify organic concepts like the "regional flavor" of a place — wished we could help a computer understand that Austinites are passionate about Barbeque, Football and Technology — and now we can, by say combining and analyzing the text of every article each city's page either links to or is geographically near.

"That's fine for the robots," says the skeptic, "but I can just phone my cousin Bubba and ask him what people in Austin like. And though I have no friend in Timbuktu, I could learn what's unique about it from the Timbuktu article and all those it links to, using my mouse or my favorite relational database." True, true. This question has what we'll call "easy locality"[1]: the pieces of context we need (the linked-to articles) are a simple mouse click or database JOIN away. But if we turn the question sideways that stops being true.

Instead of the places, let's look at the words. Barbeque is popular all through Texas and the Southeastern US, and as you'll soon be able to prove, the term "Barbeque" is over-represented in articles from that region. You and cousin Bubba would be able to brainstorm a few more terms with strong place affinity, like "beach" (the coasts) or "wine" (France, Napa Valley), and you would guess that terms like "hat" or "couch" will not. But there's certainly no simple way you could do so comprehensively or quantifiably. That's because this question has no easy locality: we'll have to dismantle and reassemble in stages the entire dataset to answer it. This understanding of *locality* is the most important concept in the book, so let's dive in and start to grok it. We'll just look at the step-by-step transformations of the data for now, and leave the actual code for a later chapter.

1. Please discard any geographic context of the word "local": for the rest of the book it will always mean "held in the same computer location"

Where is Barbecue?

So here's our first exploration:

```
For every word in the English language, which of them have a strong geographic flavor, and what ar
```

This may not be a practical question (though I hope you agree it's a fascinating one), but it is a template for a wealth of practical questions. It's a *geospatial analysis* showing how patterns of term usage vary over space; the same approach can instead uncover signs of an epidemic from disease reports, or common browsing behavior among visitors to a website. It's a *linguistic analysis* attaching estimated location to each term; the same approach term can instead quantify document authorship for legal discovery, letting you prove the CEO did authorize his nogoodnik stepson to destroy that orphanage. It's a *statistical analysis* requiring us to summarize and remove noise from a massive pile of term counts; we'll use those methods in almost every exploration we do. It isn't itself a *time-series analysis*, but you'd use this data to form a baseline to detect trending topics on a social network or the anomalous presence of drug-trade related terms on a communication channel.

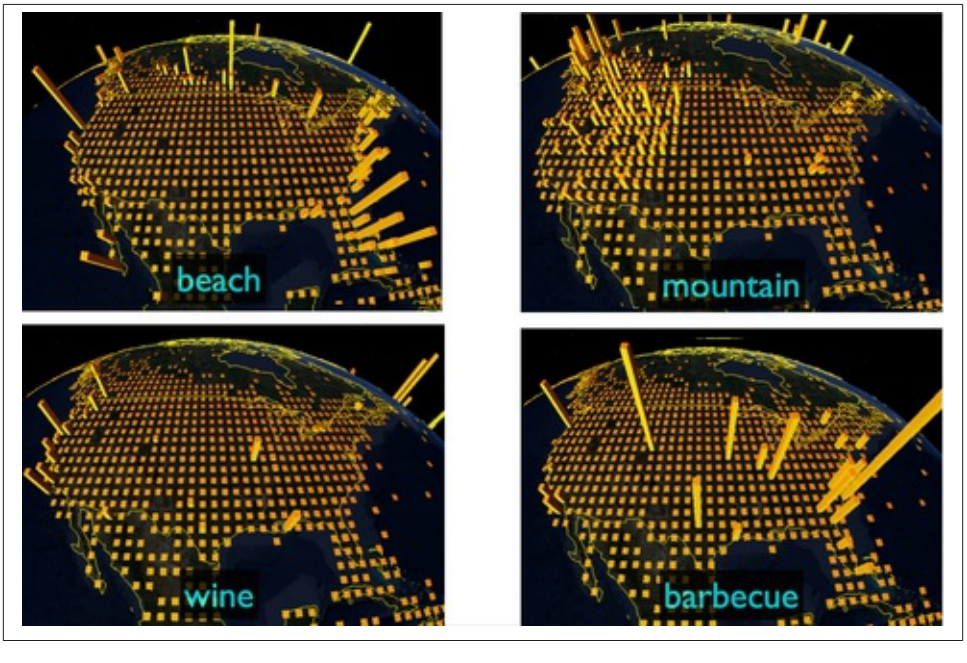


Figure 4-1. Not the actual output, but gives you the picture; TODO insert actual results

Summarize every page on Wikipedia

First, we will summarize each article by preparing its "word bag" — a simple count of the words on its wikipedia page. From the raw article text:

> *Wikipedia article on "Lexington, Texas"*
> Lexington is a town in Lee County, Texas, United States. … Snow's BBQ, which Texas Monthly called "the best barbecue in Texas" and The New Yorker named "the best Texas BBQ in the world" is located in Lexington.

we get the following wordbag:

Wordbag for "Lexington, Texas".

```
Lexington,_Texas {("texas",4)("lexington",2),("best",2),("bbq",2),("barbecue",1), ...}
```

You can do this to each article separately, in any order, and with no reference to any other article. That's important! Among other things, it lets us parallelize the process across as many machines as we care to afford. We'll call this type of step a "transform": it's independent, non-order-dependent, and isolated.

Bin by Location

The article geolocations are kept in a different data file:

Article coordinates.

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

We don't actually need the precise latitude and longitude, because rather than treating article as a point, we want to aggregate by area. Instead, we'll lay a set of grid lines down covering the entire world and assign each article to the grid cell it sits on. That funny-looking number in the fourth column is a *quadkey* [2], a very cleverly-chosen label for the grid cell containing this article's location.

To annotate each wordbag with its grid cell location, we can do a *join* of the two files on the wikipedia ID (the first column). Picture for a moment a tennis meetup, where you'd like to randomly assign the attendees to mixed-doubles (one man and one woman) teams. You can do this by giving each person a team number when they arrive (one pool of numbers for the men, an identical but separate pool of numbers for the women). Have everyone stand in numerical order on the court — men on one side, women on the

2. you will learn all about quadkeys in the "Geographic Data" chapter

other — and walk forward to meet in the middle; people with the same team number will naturally arrive at the same place and form a single team. That is effectively how Hadoop joins the two files: it puts them both in order by page ID, making records with the same page ID arrive at the same locality, and then outputs the combined record:

Wordbag with coordinates.

```
Lexington,_Texas -97.01 30.41 023130130 {("texas",4)("lexington",2),("best",2),("bbq",2),("barbecu
```

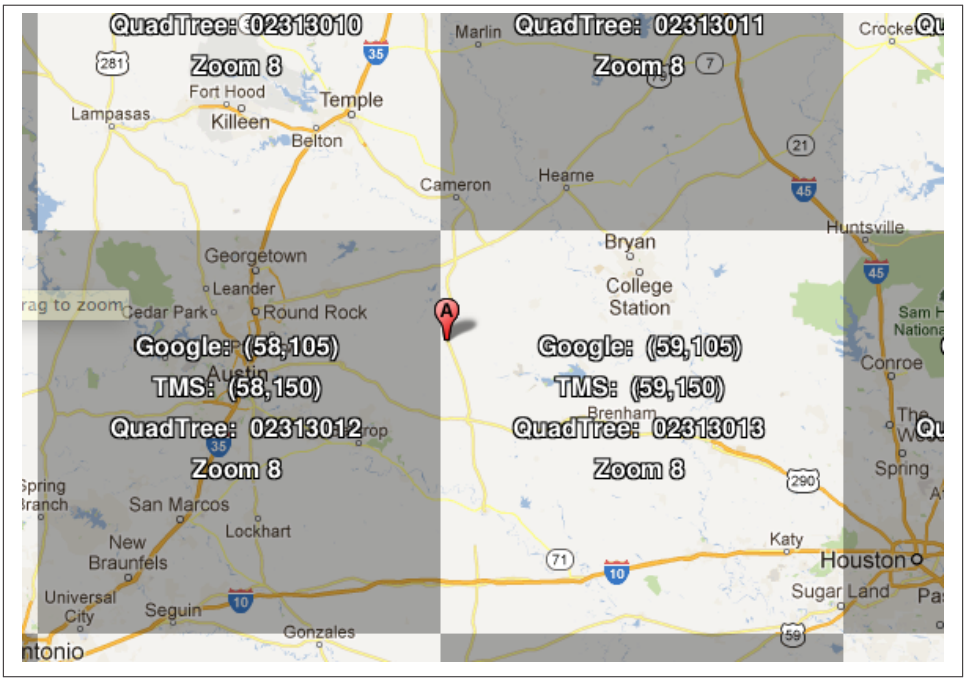


Figure 4-2. Grid Tiles for Central Texas

Gridcell statistics

We have wordbag records labeled by quadkey for each article, but we want combined wordbags for each grid cell. So we'll group the wordbags by quadkey:

```
023130130 {(Lexington,_Texas,(("many", X),...,("texas",X),...,("town",X)...("longhorns",X),...("bb
```

them turn the individual word bags into a combined word bag:

```
023130130 {(("many", X),...,("texas",X),...,("town",X)...("longhorns",X),...("bbq",X),...}
```

A pause, to think

Let's look at the fundamental pattern that we're using. Our steps:

1. transform each article individually into its wordbag
2. augment the wordbags with their geo coordinates by joining on page ID
3. organize the wordbags into groups having the same grid cell;
4. form a single combined wordbag for each grid cell.

It's a sequence of *transforms* (operations on each record in isolation: steps 1 and 4) and *pivots* — operations that combine records, whether from different tables (the join in step 2) or the same dataset (the group in step 3).

In doing so, we've turned articles that have a geolocation into coarse-grained regions that have implied frequencies for words. The particular frequencies arise from this combination of forces:

- *signal*: Terms that describe aspects of the human condition specific to each region, like "longhorns" or "barbecue", and direct references to place names, such as "Austin" or "Texas"
- *background*: The natural frequency of each term — "second" is used more often than "syzygy" — slanted by its frequency in geo-locatable texts (the word "town" occurs far more frequently than its natural rate, simply because towns are geolocatable).
- *noise*: Deviations introduced by the fact that we have a limited sample of text to draw inferences from.

Our next task — the sprint home — is to use a few more transforms and pivots to separate the signal from the background and, as far as possible, from the noise.

Pulling signal from noise

To isolate the signal, we'll pull out a trick called "Pointwise Mutual Information" (PMI). Though it may sound like an insurance holding company, in fact PMI is a simple approach to isolate the noise and background. It compares the following:

- the rate the term *barbecue* is used
- the rate that terms are used on grid cell 023130130
- the rate the term *barbecue* is used on grid cell 023130130

Just as above, we can transform and pivot to get those figures:

- group the data by term; count occurrences

- group the data by tile; count occurrences
- group the data by term and tile; count occurrences
- count total occurrences
- combine those counts into rates, and form the PMI scores.

Rather than step through each operation, I'll wave my hands and pull its output from the oven:

```
023130130 {(("texas",X),...,("longhorns",X),...("bbq",X),...,...}
```

As expected, in Figure 4-1 you see BBQ loom large over Texas and the Southern US; Wine, over the Napa Valley[3].

Takeaway #1: Simplicity

We accomplished an elaborate data exploration, yet at no point did we do anything complex. Instead of writing a big hairy monolithic program, we wrote a series of simple scripts that either *transformed* or *pivoted* the data.

As you'll see later, the scripts are readable and short (none exceed a few dozen lines of code). They run easily against sample data on your desktop, with no Hadoop cluster in sight; and they will then run, unchanged, against the whole of Wikipedia on dozens or hundreds of machines in a Hadoop cluster.

That's the approach we'll follow through this book: develop simple, maintainable transform/pivot scripts by iterating quickly and always keeping the data visible; then confidently transition those scripts to production as the search for a question becomes the rote production of an answer.

The challenge, then, isn't to learn to "program" Hadoop — it's to learn how to think at scale, to choose a workable series of chess moves connecting the data you have to the insight you need. In the first part of the book, after briefly becoming familiar with the basic framework, we'll proceed through a series of examples to help you identify the key locality and thus the transformation each step calls for. In the second part of that book, we'll apply this to a range of interesting problems and so build up a set of reusable tools for asking deep questions in actual practice.

3. This is a simplified version of work by Jason Baldrige, Ben Wing (TODO: rest of authors), who go farther and show how to geolocate texts *based purely on their content*. An article mentioning barbecue and Willie Nelson would be placed near Austin, TX; one mentioning startups and trolleys in San Francisco. See: Baldridge et al (TODO: reference)

CHAPTER 5

The Stream (ch. B)

Chimpanzee and Elephant Start a Business

As you know, chimpanzees love nothing more than sitting at typewriters processing and generating text. Elephants have a prodigious ability to store and recall information, and will carry huge amounts of carge with great determination. The chimpanzees and the elephants realized there was a real business opportunity from combining their strengths, and so they formed the Chimpanzee and Elephant Data Shipping Corporation.

They were soon hired by a publishing firm to translate the works of Shakespeare into every language. In the system they set up, each chimpanzee sits at a typewriter doing exactly one thing well: read a set of passages, and type out the corresponding text in a new language. Each elephant has a pile of books, which she breaks up into "blocks" (a consecutive bundle of pages, tied up with string).

A Simple Streamer

We're hardly as clever as one of these multilingual chimpanzees, but even we can translate text into Pig Latin. For the unfamiliar, you turn standard English into Pig Latin as follows:

- 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".

Pig Latin translator, actual version is a program to do that translation. 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 somehow have a petabyte crying out for pig-latinizing).

Pig Latin 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
```

Pig Latin 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

There's a brief ruby cheatsheet at the end of the book, but

- 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.

- the `yield latinized` line hands the translated string to wukong to output

To test the program on the commandline, run

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

The last line of its output should look like

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

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

Chimpanzee and Elephant: A Day at Work

Each day, the chimpanzee's foreman, a gruff silverback named J.T., hands the day's translation manual to chimps 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.

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.

Running a Hadoop Job

Note: this assumes you have a working Hadoop cluster, however large or small.

As you've surely guessed, Hadoop is organized very much like the Chimpanzee & Elephant team. Let's dive in and see it in action.

First, copy the data onto the cluster:

```
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.

First, let's test on the same tiny little file we used at the commandline. Make sure to notice how much *longer* it takes this elephant to squash a flea than it took to run without hadoop.

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

After outputting a bunch of happy robot-ese to your screen, the job should appear on the jobtracker window within a few seconds. The whole job should complete in far less time than it took to set it up. You can compare its output to the earlier by running

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

Now let's run it on the full Shakespeare corpus. Even this is hardly enough data to make Hadoop break a sweat, but it does show off the power of distributed computing.

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

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.
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

- Sent the job and its assets (code files, etc) to the jobtracker *

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.

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.

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.

Exercises

Exercise 1.1: Running time

It's important to build your intuition about what makes a program fast or slow.

Write the following scripts:

- **null.rb** — emits nothing.
- **identity.rb** — emits every line exactly as it was read in.

Let's run the **reverse.rb** and **piglatin.rb** scripts from this chapter, and the **null.rb** and **identity.rb** scripts from exercise 1.1, against the 30 Million Wikipedia Abstracts dataset.

First, though, write down an educated guess for how much longer each script will take than the `null.rb` script takes (use the table below). So, if you think the `reverse.rb` script will be 10% slower, write *10%*; if you think it will be 10% faster, write *-10%*.

Next, run each script three times, mixing up the order. Write down

- the total time of each run
- the average of those times
- the actual percentage difference in run time between each script and the null.rb script

```
script     | est % incr | run 1 | run 2 | run 3 | avg run time | actual % incr |
null:      |            |       |       |       |              |               |
identity:  |            |       |       |       |              |               |
reverse:   |            |       |       |       |              |               |
pig_latin: |            |       |       |       |              |               |
```

Most people are surprised by the result.

Exercise 1.2: A Petabyte-scale `wc` command

Create a script, `wc.rb`, that emit the length of each line, the count of bytes it occupies, and the number of words it contains.

Notes:

- The `String` methods `chomp`, `length`, `bytesize`, `split` are useful here.
- Do not include the end-of-line characters (`\n` or `\r`) in your count.
- As a reminder — for English text the byte count and length are typically similar, but the funny characters in a string like "Iñtërnâtiônàlizætiøn" require more than one byte each. The character count says how many distinct *letters* the string contains, regardless of how it's stored in the computer. The byte count describes how much space a string occupies, and depends on arcane details of how strings are stored.

CHAPTER 6

Reshape Steps (ch. C)

Locality of Reference

```
Ever since there were

[] [] [] []
[] [] [] [] eight computers and a network,

programmers have wished that

eight computers   [] [] [] [] [] [] [] [] []
solved problems
twice as fast as
four computers    [] [] [] []

The problem comes
      when the computer over here \/
   needs to talk to   [] [] [] [] [] [] [] <- the one over here
                      [] [] [] [] [] [] [] <- or the one here
      and this one -> [] [] [] [] [] [] []
        needs to talk to this one ^^
and so forth.
```

If you've ever had lunch with a very large troop of chimpanzees, hollering across the room to pass the bananas, and grabbing each other's anthill-poking twigs, you can imagine how hard it is to get any work done.

Locality: Examples

This book is fundamentally about just one thing: helping you identify the key locality of a data problem.

- **word count**: Count the occurrences of every word in a large body of text: gather each occurrence into groups by word, then count each group.

- **total sort**: To sort a large number of names, group each name by its first few letters (eg "Aaron" = "aa", "Zoe" = "zo"), making each group small enough to efficiently sort in memory. Reading the output of each group in the order of its label will produce the whole dataset in order.
- **network graph statistics**: Counting the average number of Twitter messages per day in each user's timeline requires two steps. First, take every *A follows B* link and get it into the same locality as the user record for its "B". Next, group all those links by the user record for *A*, and sum their messages-per-day.
- **correlation**: Correlate stock prices with pageview counts of each corporation's Wikipedia pages: bring the stock prices and pageview counts for each stock symbol together, and sort them together by time.

The Hadoop Haiku

You can try to make it efficient for any computer to talk to any other computer. But it requires top-of-the-line hardware, and clever engineers to build it, and a high priesthood to maintain it, and this attracts project managers, which means meetings, and soon everything is quite expensive, so expensive that only nation states and huge institutions can afford to do so. This of course means you can only use these expensive supercomputer for Big Important Problems — so unless you take drastic actions, like joining the NSA or going to grad school, you can't get to play on them.

Instead of being clever, be simple.

Map/Reduce proposes this fair bargain. You must agree to write only one type of program, one that's as simple and constrained as a haiku.

The Map/Reduce Haiku.

```
data flutters by
    elephants make sturdy piles
  insight shuffles forth
```

If you do so, Hadoop will provide near-linear scaling on massive numbers of machines, a framework that hides and optimizes the complexity of distributed computing, a rich ecosystem of tools, and one of the most adorable open-source project mascots to date.

More prosaically,

1. **label** — turn each input record into any number of labelled records
2. **group/sort** — hadoop groups those records uniquely under each label, in a sorted order
3. **reduce** — 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.

Let's join back up with the Chimpanzee and Elephant Shipping Company and see this in practice.

Chimpanzee, Elephant and Elf worked in harmony according to the following workflow:

- Chimpanzee:
  - — turns each child's letter into a set of work orders,
  - — labeled by type of toy and destination,
  - — in file folders partitioned by toy and sorted by destination
- Elephant:
  - — marches each partition of work orders to the specific workbench for that set of toys
- Elf:
  - — as soon as enough elephants stand ready, begins merging the file folders from

CHAPTER 7

Chimpanzee and Elephant Save Christmas (ch. D)

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 this year there was a problem: the world had grown, and the elves just couldn't keep up with the scale of requests. Luckily, their friends at the Elephant & Chimpanzee Data Shipment Company were available to simplify the process.

A Non-scalable approach

To meet the wishes of children from every corner of the earth, each elf is capable of making any kind of toy, from Autobot to Pony to X-box.

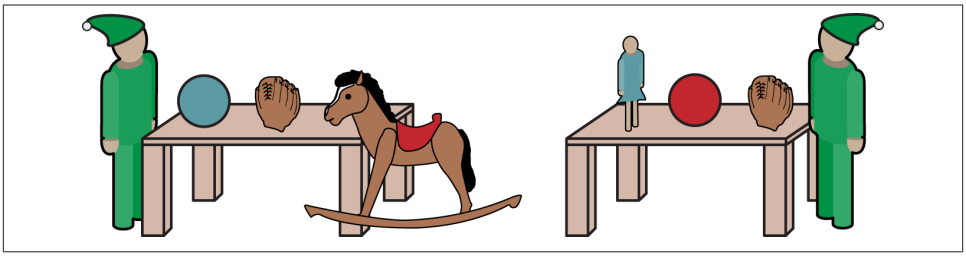


Figure 7-1. The elves' workbenches are meticulous and neat.

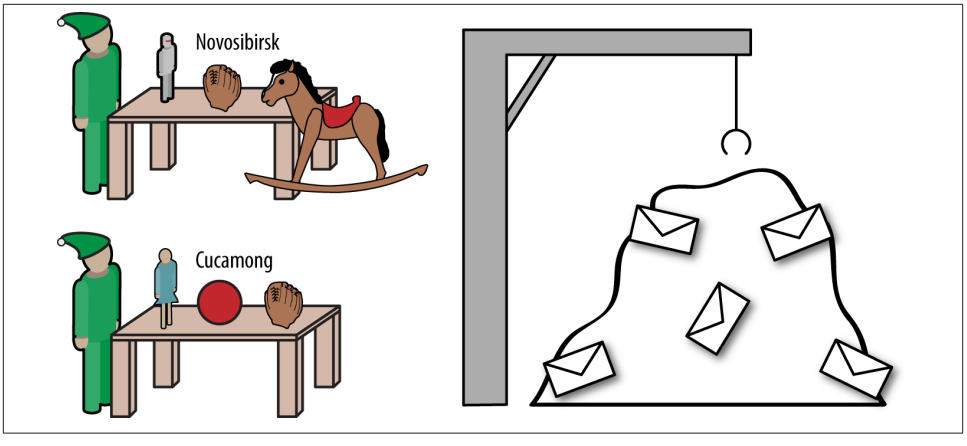


Figure 7-2. Little boys and girls' mail is less so.

As bags of mail arrived from all over, they were hung from the branches of a large Tree (known as the bag tree, or B.Tree for short.) Each time an elf was ready for the next letter from her region, a big claw arm swung out to the right spot on the B.Tree to retrieve it. Without locality of access (the elf covering Novosibirsk might be in line right behind the elf covering Cucamonga), letters could't be pulled from the tree any faster than the crane arm could move.

The hallways were clogged with frazzled elves running back and forth between their workbenches and the B.Tree. It almost seemed as if elves spent as much effort on the mechanics of retrieving letters as they did making toys.

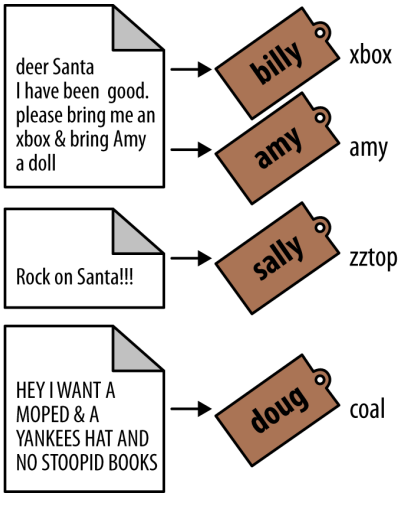
Letters to Toy Requests

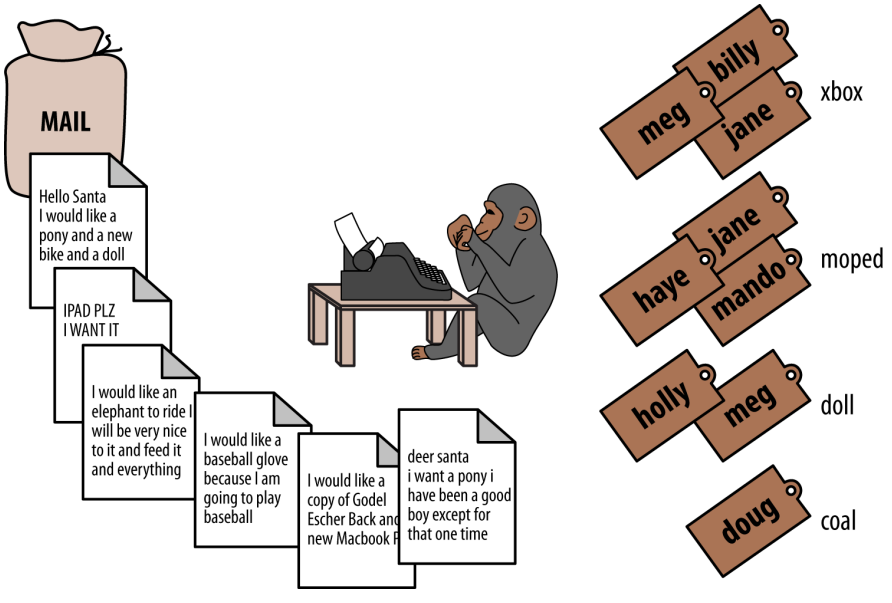
In place of this came the chimps and elephants, singing a rather bawdy version of the Map-Reduce Haiku, who built the following system.

As you might guess, they lined up a finite number of chimpanzees at typewriters to read each letter. Instead of sending the letter on directly, the chimp would fill out a work form for each requested toy, labelled prominently by the type of toy. Some examples:

```
-------------------------------------          # Joe is clearly a good kid, and thoughtful for his

Deer SANTA                                     robot | type="optimus prime" recipient="Joe"
                                               hat   | type="girls small"  recipient="Joe's sister

I wood like an optimus prime robot
and a hat for my sister julia

I have been good this year

love joe

-------------------------------------           # Frank is a jerk. He will get coal.
```

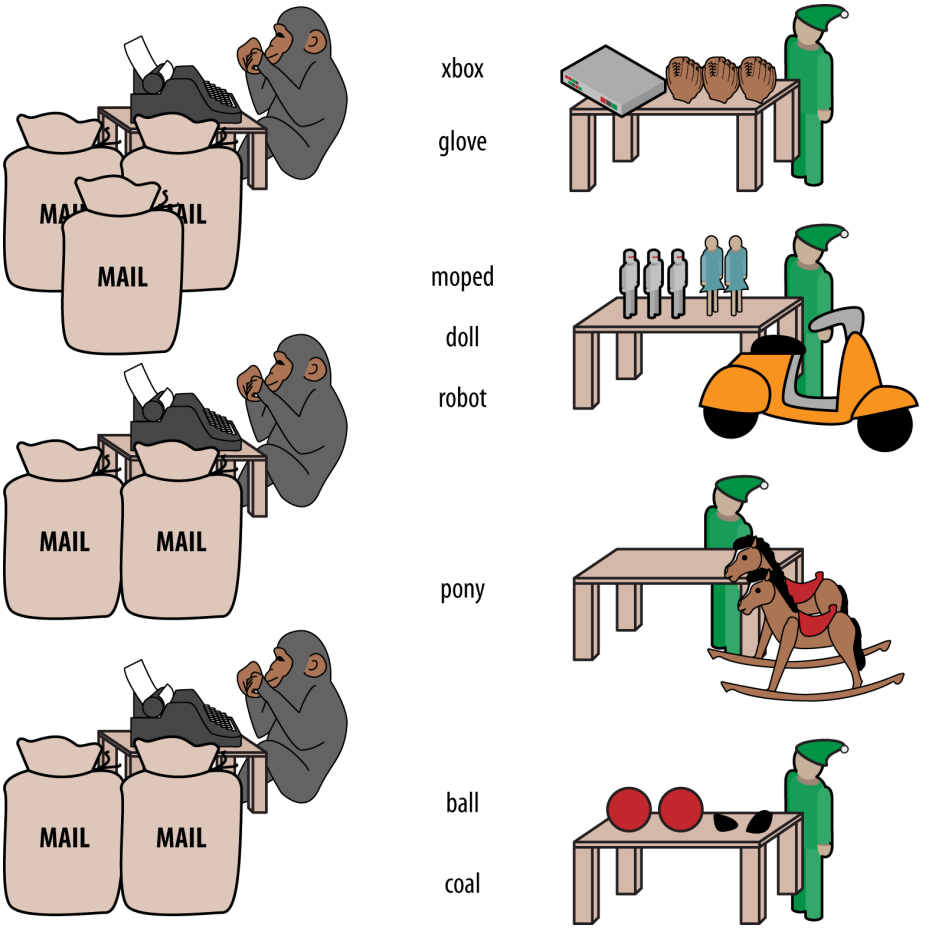
```
HEY SANTA I WANT A PONY AND NOT ANY              coal  | type="anthracite" recipient="Frank" reason
DUMB BOOKS THIS YEAR

FRANK

--------------------------------------           # 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...

--------------------------------------
```



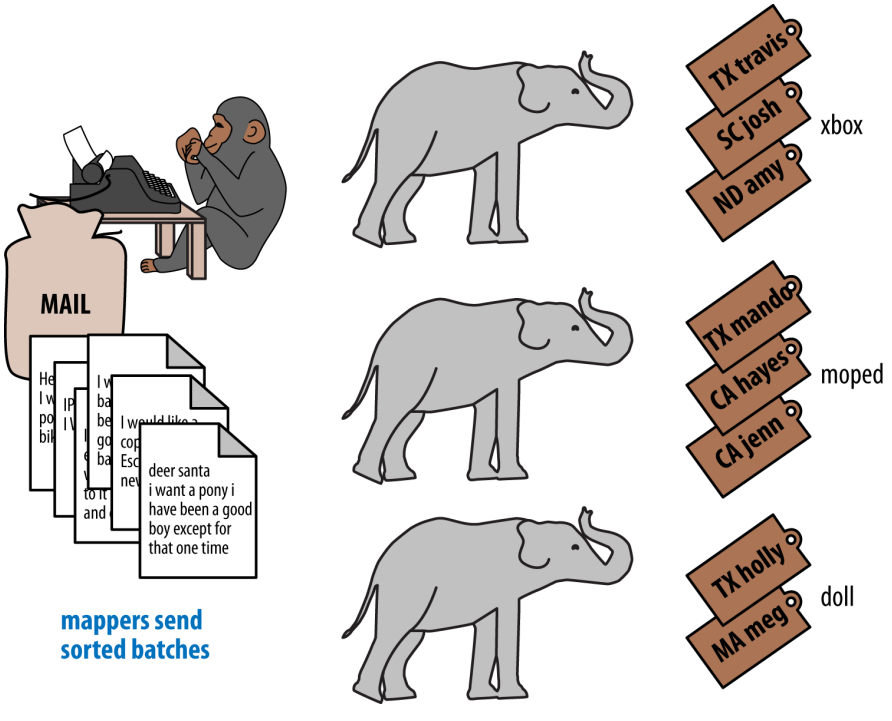


Order Delivery

In the new system, each type of toy request is handled at the one workbench designated for that toy. For example, all robots and model cars might go to workbench A, while ponies, coal and yo-yos went to workbench B.



Next to each chimpanzee stands a line of pygmy elephants, one for each workbench. Once the chimp is done with a mailbag, those elephants march off, each to its addressed workbench. Behind them a new line of elephants trundle into place.



Toy Assembly

A station might handle several types of toys in a factory run, but it always sees them in a continuous batch (they will get all the robots, then all the model cars). This is of great help to the elves, as the set-up for tying ribbons on Ponies is very different from the set-up for gift-wrapping Quidditch Brooms.

```
Doll    Julia          Anchorage       USA
Doll    Madison        Zanesville      USA
Doll    Wei Ju         Shenzen         China
...
Coal    Jim            Mountain View   CA
```

The finished toys go into large sacks for santa to deliver.

Why it's efficient

Now it is still true that each elf workstation has incoming mail from every letter-reader. A constant stream of elephants are constantly dropping off order batches, some light, some heavy.

But the delivery isn't harum-scarum all-at-once, it's orderly and purposeful. If one workstation is slow, the elephants wait patiently — it doesn't slow down the entire operation. And most importantly, all the work of organizing the work requests happens in parallel with reading the letters. It's pretty impressive.

Sorted Batches

Santa delivers presents in order as the holidays arrive, racing the sun from New Zealand, through Asia and Africa and Europe, until the finish in American Samoa.

This is a literal locality: the presents for Auckland must go in a sack together, and Sydney, and Petropavlovsk, and so forth.

Recall that each elephant carries the work orders destined for one workstation. What's more, on the back of each pygmy elephant is a vertical file like you find at a very organized person's desk:

Chimpanzees file each toy request in the order of Santa's path through the world. This is easy, because the files never grow very large and because chimpanzees are very dextrous. So when a pygmy elephant trundles off, all the `puppy` requests are together in order from Auckland to Samoa, and the `robot` requests are together, also in order, and so on:

This makes life very efficient for the elves. They just start pulling work orders from their elephants, always choosing the request that's next in Santa Visit Order:

Elves do not have the prodigious memory that elephants do, but they can easily keep track of the next few dozen work orders each elephant holds. That way there is very little time spent seeking out the next work order. Elves assemble toys as fast as their hammers can fly, and the toys come out in the order Santa needs to make little children happy.

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
  insight shuffles forth
```

More prosaically,

1. **label** — turn each input record into any number of labelled records
2. **group/sort** — hadoop groups those records uniquely under each label, in a sorted order
3. **reduce** — 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 allowed no locality. 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.

Stream steps become mapper-only jobs, but don't conflate a reshape *step* with the reduce *phase* of a job. A reshape step typically becomes at least one mapper phase and reducer phase.

The Reducer Guarantee

It's very important that you understand the guarantee Hadoop provides to each reducer, and what it unlocks.

- Each mapper-output record goes to exactly one reducer, as solely determined by its key.
- All records for a given key go to the same reducer
- If a reducer sees any element of a group, it sees all elements of the group. A reducer may see multiple groups.

- Records are fed to the reducer in order by key.

Partition Key and Sort Key

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"

CHAPTER 8
Geo Data

Spatial data is of course one of the fundamental ways we understand the universe

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,

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)

It's a good jumping-off point for machine learning. Take a tour through some of the sites that curate the best in data visualization, and you'll see a strong over-representation 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 [1]. Geo Data comes out of the

Taking a step back, the fundamental idea this chapter introduced 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. 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.

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 [2], 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
- "Feature" — a generic term for "Point or Path or Region".

1. we dive deeper in the chapter on ??? basics later on
2. in other works you'll see the term Point of Interest ("POI") for a place.

- "Bounding Box" (or `bbox`) — a rectangular bounding region, `[-5.0, 30.0, 5.0, 40.0]` *

Features of Features

There's a slight muddying of the term "feature" — to a geographer, a feature is a generic term for the *thing* being described; later, in the chapter on machine learning, a feature Since I'm writing as a data scientist dabbling in geography (and because that chapter's hairy enough as it is), we'll just say "object" in place of "geographic feature"

Geospatial Information Science ("GIS") is a deep subject, 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. [3]

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.

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.

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

Within each of those, make a Z again:

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

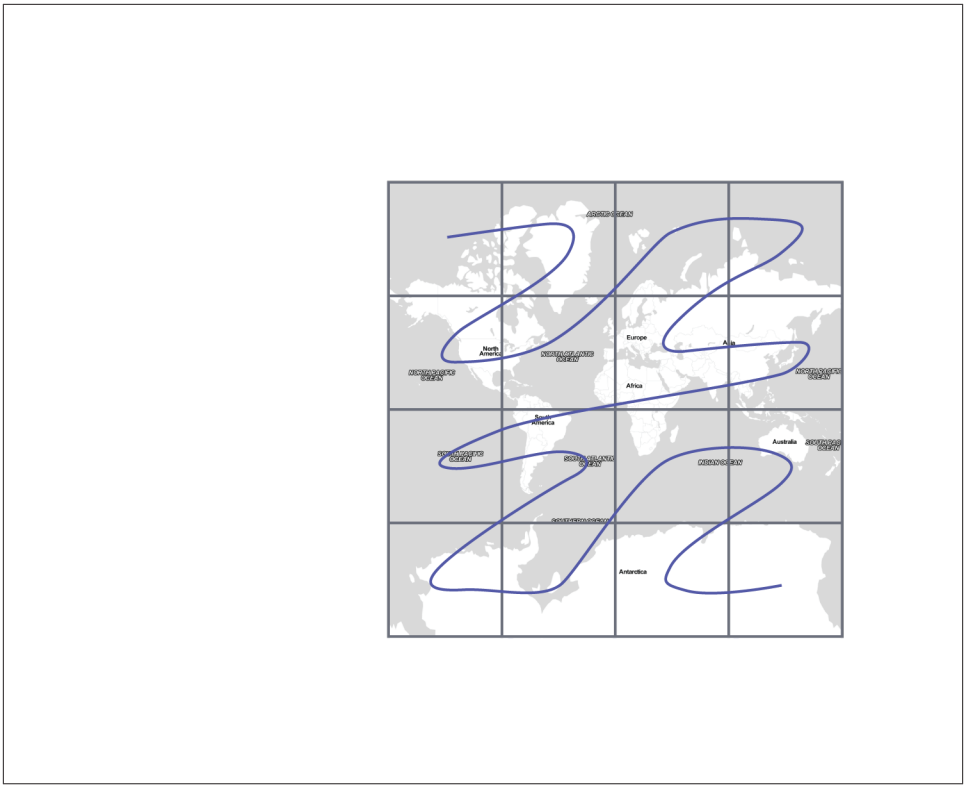


Figure 8-1. Z-path of quadtiles

As you go along, index each tile, as shown in Figure 8-2:

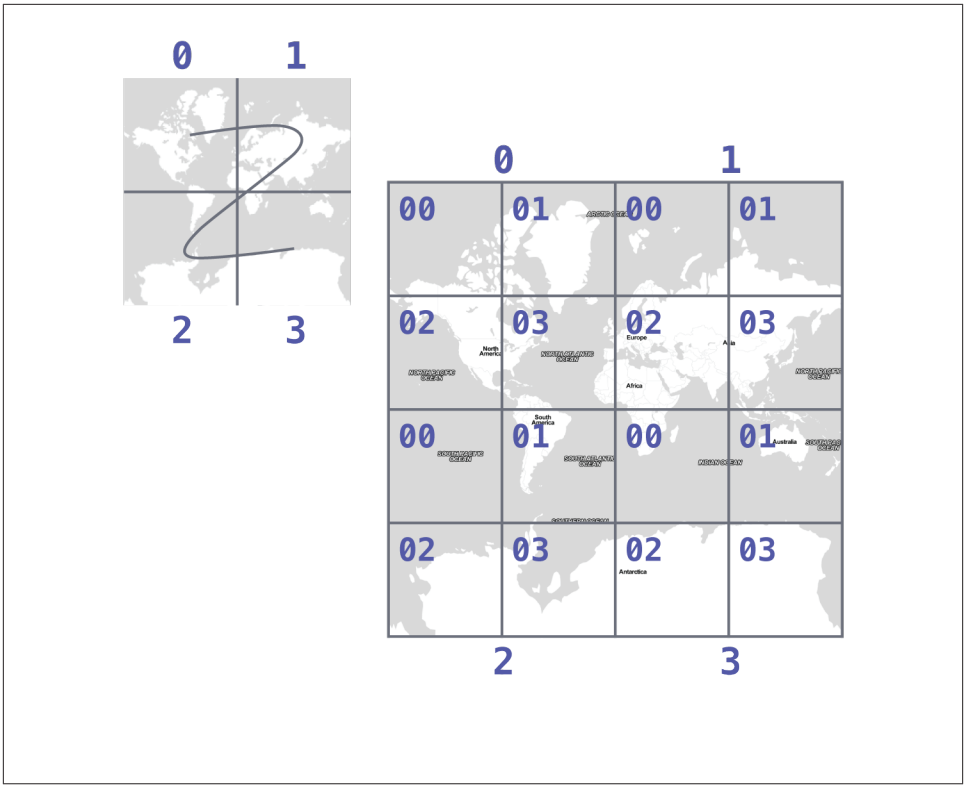


Figure 8-2. Quadtile Numbering

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.

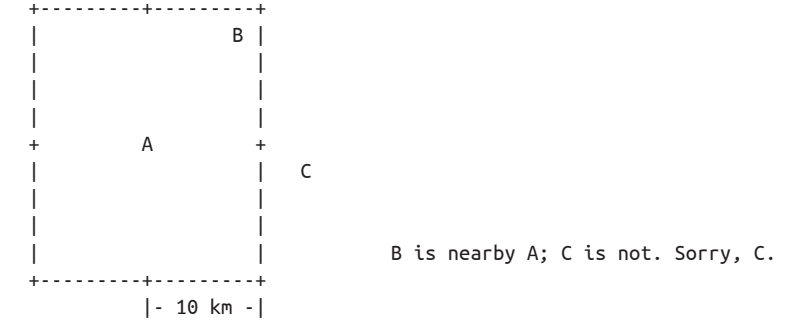
Patterns in UFO Sightings

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

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]`

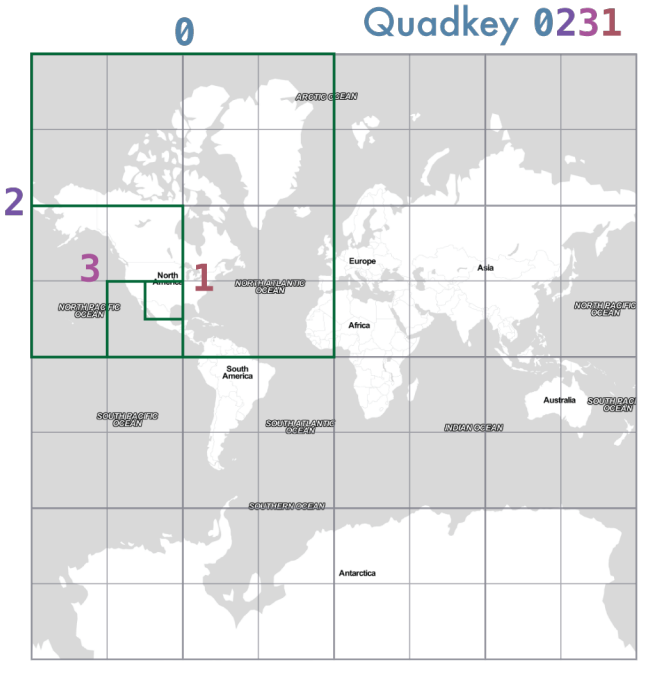
An example hash, taken from the spec:

```
{
  "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] ]
          ]
      }
    }
  ]
}
```

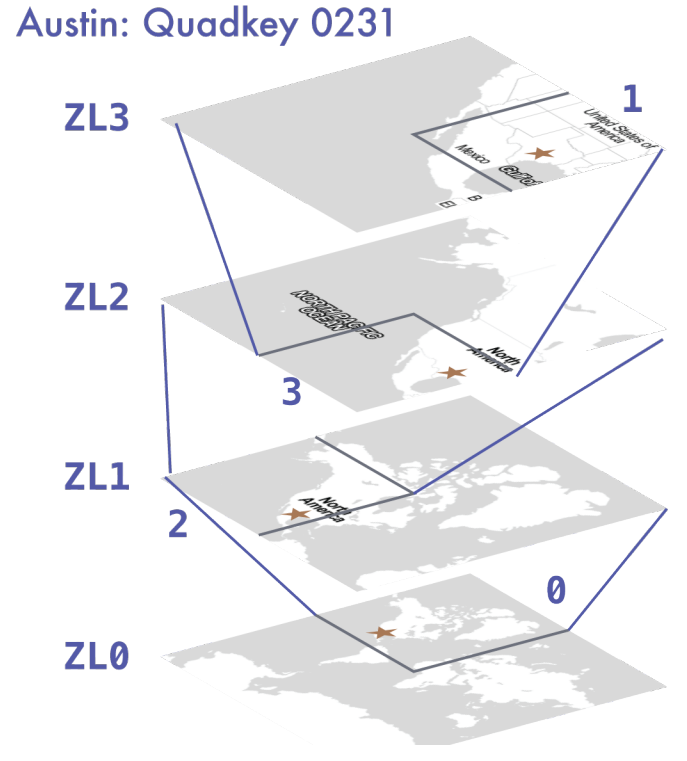
Quadtile Practicalities

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 [4]

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.

4. briefly featured in the Clash's Rock the Casbah Video and where much of this book was written

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.

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:

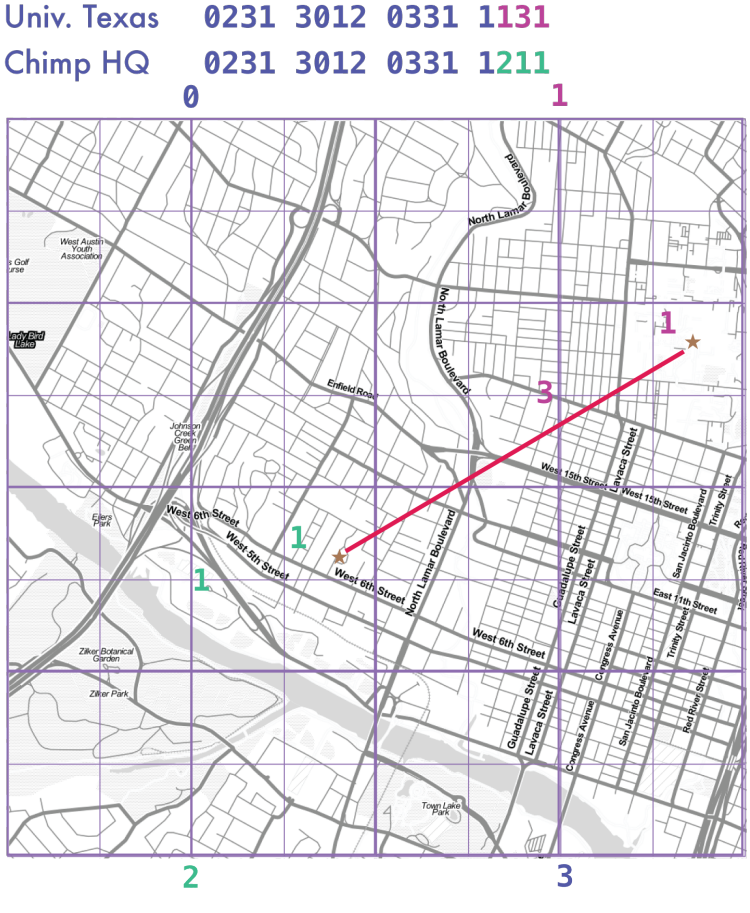
Latitude	Cities near that Latitude	Grid Size % of Equator	Lateral Radius	66 K Rec Grid Size ZL 8	1 M Rec Grid Size ZL 10	17 M Rec Grid Size ZL 12	1 B Rec Grid Size ZL 15	69 B Rec Grid Size ZL 18
0	Quito, Nairobi, Singapore	100%	6,378	157 km	39 km	10 km	1,223 m	153 m
5	Cote d'Ivoire, Bogotá; S: Kinshasa, 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.

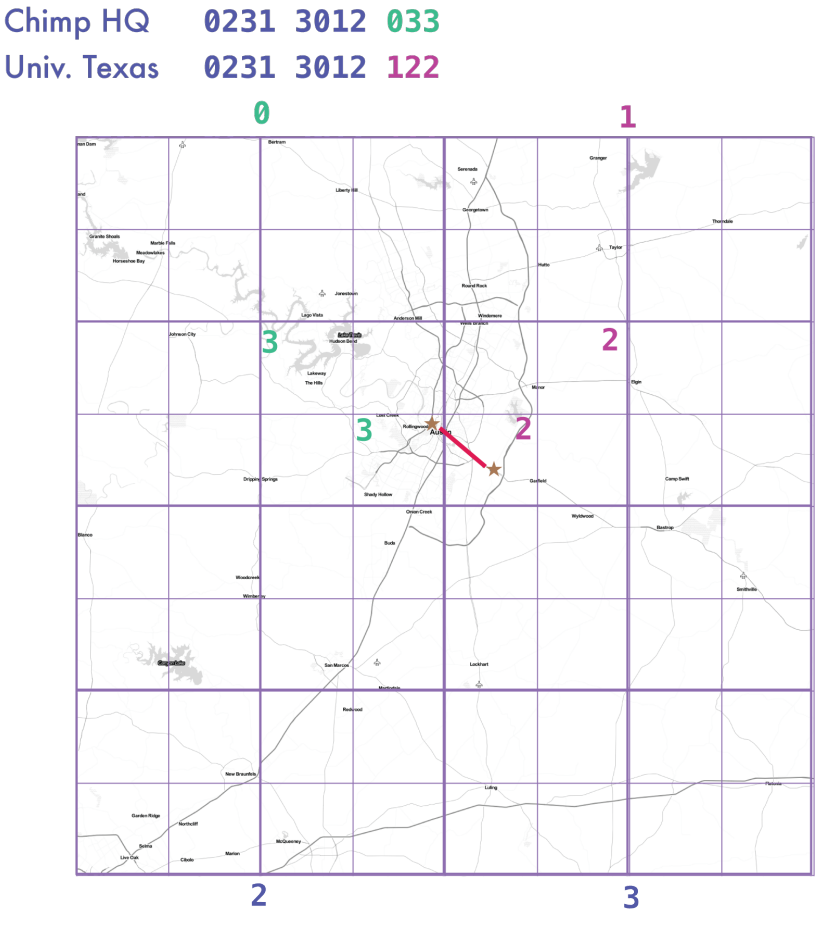
Zoom Lvl	Lng	Lat	Quadkey (ZL 14)	XY	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
New York	-73.8	40.6	03201011120210 02	Tile X	0	0	1	2	4	9	18	37	75	151	302	604	1208	2417	4834	9668	19336	38673	77347	154695	309391
				Tile Y	0	0	1	3	6	12	24	48	96	192	385	770	1541	3082	6164	12328	24657	49315	98631	197262	394524
San Francisco	-73.8	37.6	03201033100212 20	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	02313012122212 13	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	03131313103031 02	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	12330031111212021	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	13300211233021 32	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	13212110302133 01	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	31133300030232 13	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

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[5].

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.

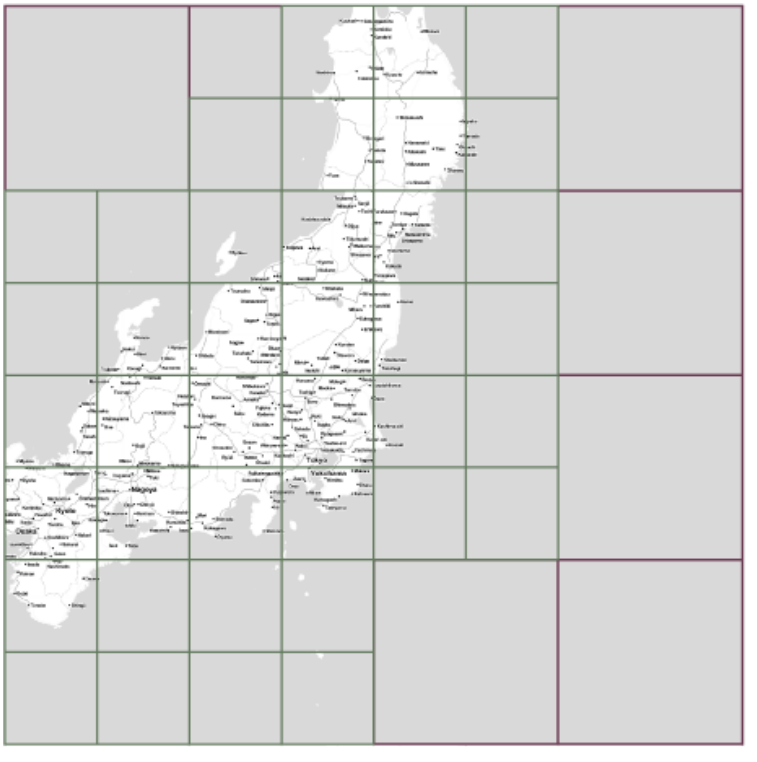
5. 000 001 100 101 202 203 302 and 303

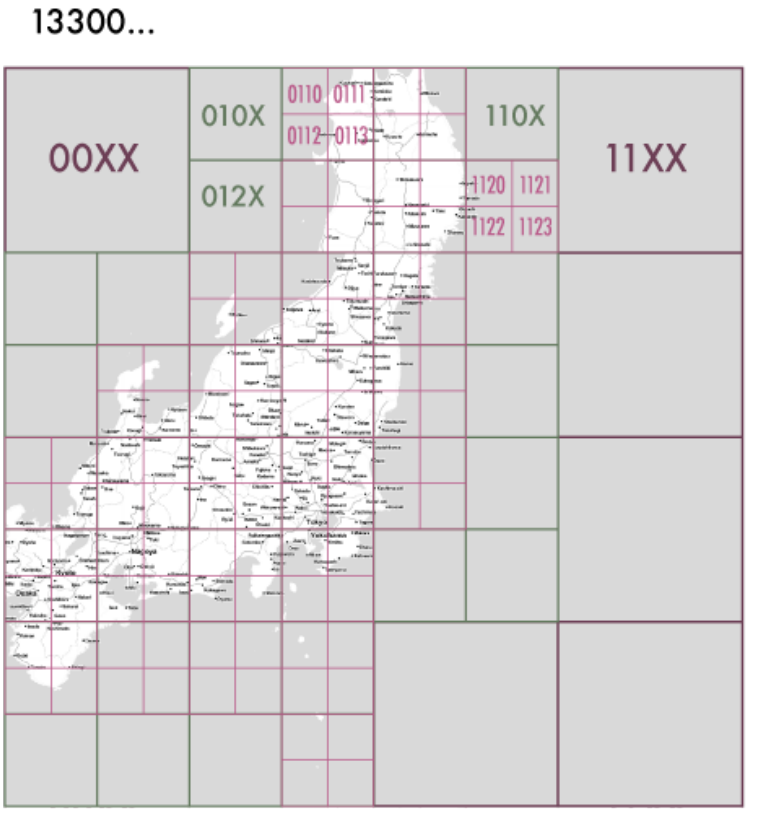
Already six of the 16 tiles shown don't have any land coverage, so you can record their values:

```
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`, ".


Tokyo
Yokohama
Nagoya
Kyushu
Osaka



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.

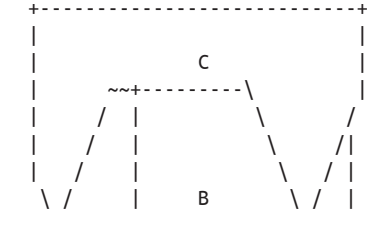
```
00.00.00
00.00.01
00.00.10
00.00.11
00.01.--
00.10.--
00.11.00
00.11.01
00.11.10
00.11.11
01.--.--
10.00.--
10.01.--
10.10.01
10.10.10
10.10.11
10.10.00
10.11.--
```

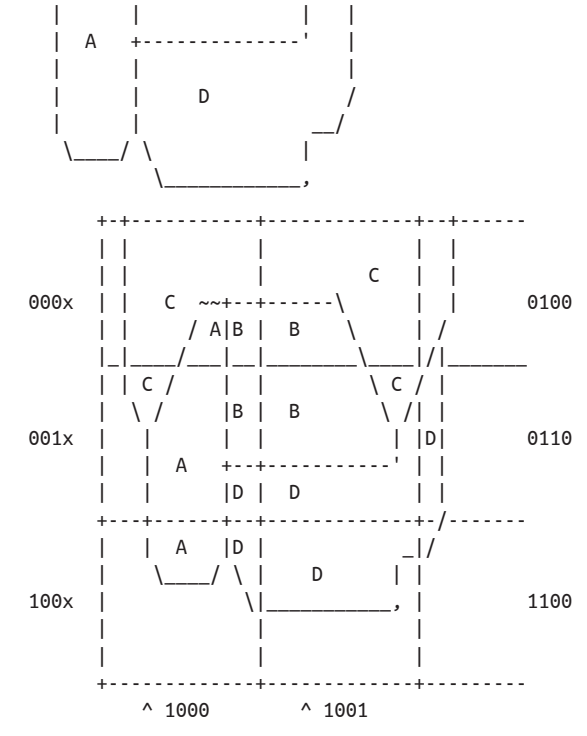
Tree structure of Quadtile indexing

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

These factors rule out any naïve approach to locality, but there's an elegant solution known as a Voronoi diagram [6].

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 [7].

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

6. see Wikipedia entry or (with a Java-enabled browser) this Voronoi Diagram applet

7. 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*

stand at the midpoint of the edge connecting two locations, and walk perpendicular to 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.

K-means clustering to summarize

(TODO: section under construction)

we will describe how to use clustering to form a progressive summary of point-level detail.

there are X million wikipedia topics

at distant zoom levels, storing them in a single record would be foolish

what we can do is summarize their contents — coalesce records into groups based on their natural spatial arrangement. If the points represented foursquare checkins, those clusters would match the population distribution. If they were wind turbine generators, they would cluster near shores and praries.

K-Means Clustering is an effective way to form that summarization.

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

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
```

References

- *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

CHAPTER 9

Log Processing

Data Model

First let's assemble our data model:

```
class Logline
  include Gorillib::Model
  include Gorillib::Model::PositionalFields

  field :ip,            String
  field :junk1,         String
  field :junk2,         String
  #
  field :visit_time,    Time
  field :http_method,   String
  field :path,          String
  field :protocol,      String
  field :response_code, Integer
  field :size,          Integer, blankish: ['', nil, '-']
  field :referer,       String
  field :ua,            String
  field :cruft,         String
end
```

We should also add a method to the model to parse each line. It just breaks up each line into fields and passes them in order to the `Logline` model:

Simple Log Parsing

Now the parse script is simple as can be: take each line, hand it to `Logline` to parse, and emit the model object that returns. We don't need to do anything more, so there's no reducer:

There's a couple more gory details of parsing each log line, but trust me on them for a moment, and let's run this script. First, in local map mode; here's lines 5000-5010 of the logs:

```
head -n 5000 star_wars_kid-parsed.tsv  | tail -n 100 | wu-lign | cutc
193.120.205.131 -       -       2003-04-11 10:33:41 UTC GET     /archive/2003/03/26/hiding_s.shtml
193.120.205.131 -       -       2003-04-11 10:33:52 UTC GET     /
64.68.82.25     -       -       2003-04-11 10:34:52 UTC GET     /mefi/users/?month=06&year=2000
69.10.137.199   -       -       2003-04-11 10:35:00 UTC GET     /watch-info
193.120.205.131 -       -       2003-04-11 10:35:35 UTC GET     /archive/2003/04/03/typo_pop.shtml
66.230.140.66   -       -       2003-04-11 10:36:23 UTC GET     /archive/2002/07/08/gene_kan.shtml
80.58.4.237     -       -       2003-04-11 10:36:28 UTC GET     /archive/2002/09/03/magazine.shtml
80.58.4.237     -       -       2003-04-11 10:36:51 UTC GET     /archive/cat/music/index.shtml
128.54.147.10   -       -       2003-04-11 10:37:09 UTC GET     /archive/2002/06/29/waxy_bac.shtml
207.166.200.201 -       -       2003-04-11 10:37:21 UTC GET     /random/html/uptime.txt
131.107.137.47  -       -       2003-04-11 10:38:06 UTC GET     /archive/2002/11/17/doomsday.shtml
151.100.59.2    -       -       2003-04-11 10:38:14 UTC GET     /archive/2002/06/12/the_onio.shtml
213.122.211.121 -       -       2003-04-11 10:38:36 UTC GET     /archive/2003/04/03/typo_pop.shtml
```

Parser script

I've used the `//x` form of regular expression — this ignores the whitespace and treats `#` as introducing a comment. So at the cost of having to explicitly use `\s` for "space", we get a very readable program:

```
class Logline
  #
  # Regular expression to parse an apache log line.
  #
  # 83.240.154.3 - - [07/Jun/2008:20:37:11 +0000] "GET /faq HTTP/1.1" 200 569 "http://infochimps.c
  #
  LOG_RE = Regexp.compile(%r{\A
               (\S+)           # ip             83.240.154.3
             \s(\S+)           # j1             -
             \s(\S+)           # j2             -
           \s\[(\d+/\w+/\d+    # date part      [07/Jun/2008
               :\d+:\d+:\d+    # time part      :20:37:11
             \s[\+\-]\S*)\]    # timezone       +0000]
         \s\"(?:(\S+)          # http_method    "GET
             \s(\S+)           # path           /faq
    \s(HTTP/[\d\.]+)|-)\"      # protocol       HTTP/1.1"
             \s(\d+)           # response_code  200
             \s(\d+|-)         # size           569
           \s\"([^\"]*)\"      # referer        "http://infochimps.org/search?query=CAC"
           \s\"([^\"]*)\"      # ua             "Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1
         \z}x)
end
```

The second important detail concerns parsing the time.

```
class Logline

  MONTHS = { 'Jan' => 1, 'Feb' => 2, 'Mar' => 3, 'Apr' => 4, 'May' => 5, 'Jun' => 6, 'Jul' => 7, '

  def receive_visit_time(val)
    if %r{(\d+)/(\w+)/(\d+):(\d+):(\d+):(\d+)\s([\+\-]\d\d)(\d\d)} === val
      day, mo, yr, hour, min, sec, tz1, tz2 = [$1, $2, $3, $4, $5, $6, $7, $8]
      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
```

- Don't try to be a hero and get everything done in one regexp. You've gone from gallant to goofus for sure if you start using backreferences. Here, we use the first regexp to just separate the date-time string from the herd, then finish it in a clean, quiet place.
- See wire receiver pattern for why we use `receive_visit_time` rather than override the setter (`visit_time=`) method
  - A receive method must either call `super(val)` to save the value, or call `write_attribute` directly.
- Generic parsing of times can be shockingly slow. The version you see above came after profiling the script verified it was better to
- The main regexp and the one in `receive_visit_time` are good brittle.
  - spells out HTTP, specifies digits `\d` etc
  - anchored at the beginning and end, using multi-line anchors (`^` and `$` match on a newline; `\A` and `\z` match strictly at the beginning and end of a string).

Histograms

When do people visit the site?

We want to group on `day_hr`, so just add a method implementing (and documenting!) that domain knowledge.

```
class Logline
  def day_hr
    [visit_time.year, visit_time.month, visit_time.day, visit_time.hour].join
  end
end
```

This is the advantage of having a model and not just a passive sack of data.

Run it in map mode:

```
$ cat data/swk-100.tsv      | ./histograms.rb --map | wu-lign  | sort
/                                                     200343020
/                                                     200343020
/                                                     200343020
/                                                     200343020
/                                                     200343020
/                                                     200343020
/                                                     200343020
/                                                     200343020
/                                                     200343020
/archive/2003/02/06/the_1250.shtml                    200343020
/archive/2003/02/19/coachell.shtml                    200343020
/archive/2003/02/21/gift_upg.shtml                    200343020
/archive/2003/03/14/dixie_ch.shtml                    200343020
/archive/2003/03/27/open_cdr.shtml                    200343020
```

TODO: digression about `wu-lign`.

Sort and save the map output; then write and debug your reducer.

When things are working, this is what you'll see. Notice that the `.../Star_Wars_Kid.wmv` file already have five times the pageviews as the site root (`/`).

```
$ cat data/swk-100.tsv      | ./histograms.rb --map | sort > data/swk-hist-map.tsv

$ cat data/swk-hist-map.tsv | ./histograms.rb --reduce
/                                                     200343020       10
/archive/2003/02/06/the_1250.shtml                    200343020       1
/archive/2003/02/19/coachell.shtml                    200343020       1
/archive/2003/02/21/gift_upg.shtml                    200343020       1
/archive/2003/03/14/dixie_ch.shtml                    200343020       1
/archive/2003/03/27/open_cdr.shtml                    200343020       1
/archive/2003/04/21/classmat.shtml                    200343020       1
/archive/2003/04/29/star_war.shtml                    200343020       9
/favicon.ico                                          200343020       1
/index.rdf                                            200343020       3
/projects/domains/domains.net.txt                     200343020       1
/random/archive/pooyan.zip                            200343020       1
/random/text/MAD.Magazine.418.june.2002-een.nfo.txt   200343020       1
/random/video                                         200343020       1
/random/video/Star_Wars_Kid.wmv                       200343020       50
/random/video/Star_Wars_Kid_Remix.wmv                 200343020       16
/watch-info                                           200343020       1
```

You're ready to run the script in the cloud! Fire it off and you'll see dozens of workers start processing the data.

```
$ ./histograms.rb --run data/star_wars_kid.tsv data/star_wars_kid-pages_by_hour.tsv
I, [2012-08-15T22:39:43.675553 #61524]  INFO -- :   Launching hadoop!
I, [2012-08-15T22:39:43.676396 #61524]  INFO -- : Running

/usr/local/share/hadoop/bin/hadoop      \
```

```
jar /usr/local/share/hadoop/contrib/streaming/hadoop-*streaming*.jar  \
-D stream.num.map.output.key.fields=3         \
-D mapred.job.name='histograms.rb---data/star_wars_kid.tsv---data/star_wars_kid-pages_by_hour.ts
-mapper  '/Users/flip/.rbenv/versions/1.9.3-p125/bin/ruby histograms.rb --map --log_interval=100
-reducer '/Users/flip/.rbenv/versions/1.9.3-p125/bin/ruby histograms.rb --reduce --log_interval=
-input   'data/star_wars_kid.tsv'     \
-output  'data/star_wars_kid-pages_by_hour.tsv'       \
-file    '/Users/flip/ics/core/wukong_og/examples/server_logs/histograms.rb'  \
-cmdenv 'RUBYLIB=/Users/flip/.rubylib'
```

User Paths through the site ("Sessionizing")

Now let's do some stuff more challenging that you'd try in a normal relational DB.

NOTE:[What are the important locality feature(s) to group on?]

spit out `[ip, day_hr, visit_time, path]`.

run it in map mode:

```
$ cat data/swk-100.tsv      | ./histograms.rb --map | wu-lign
/archive/2003/04/29/star_war.shtml                      200343020
/random/video/Star_Wars_Kid.wmv                         200343020
/random/video/Star_Wars_Kid.wmv                         200343020
/random/video/Star_Wars_Kid.wmv                         200343020
/archive/2003/02/19/coachell.shtml                      200343020
/archive/2003/04/29/star_war.shtml                      200343020
/                                                       200343020
/random/video/Star_Wars_Kid_Remix.wmv                   200343020
/random/video/Star_Wars_Kid_Remix.wmv                   200343020
/random/video/Star_Wars_Kid.wmv                         200343020
/                                                       200343020
/archive/2003/03/27/open_cdr.shtml                      200343020
/random/video/Star_Wars_Kid.wmv                         200343020
/random/video/Star_Wars_Kid.wmv                         200343020
/random/video/Star_Wars_Kid.wmv                         200343020
/random/video/Star_Wars_Kid.wmv                         200343020
/random/video/Star_Wars_Kid_Remix.wmv                   200343020
/random/video/Star_Wars_Kid_Remix.wmv                   200343020
/random/video/Star_Wars_Kid_Remix.wmv                   200343020
```

```
$ cat data/swk-100.tsv      | ./breadcrumbs.rb --map | sort
12.165.53.5     200343020       1051734009      /random/video/Star_Wars_Kid_Remix.wmv
12.165.53.5     200343020       1051734013      /random/video/Star_Wars_Kid_Remix.wmv
12.34.246.7     200343020       1051734043      /archive/2003/04/29/star_war.shtml
12.42.55.174    200343020       1051733934      /random/video/Star_Wars_Kid_Remix.wmv
12.42.55.174    200343020       1051733981      /random/video/Star_Wars_Kid_Remix.wmv
129.8.100.123   200343020       1051734019      /archive/2003/04/29/star_war.shtml
131.212.155.170 200343020       1051734004      /random/video/Star_Wars_Kid.wmv
131.229.113.229 200343020       1051733829      /random/video/Star_Wars_Kid.wmv
133.11.36.26    200343020       1051733907      /archive/2003/03/14/dixie_ch.shtml
```

group on user

```
$ cat data/swk-100.tsv      | ./breadcrumbs.rb --map | sort | ./breadcrumbs.rb --reduce | wu-lign
12.165.53.5     1051734009:/random/video/Star_Wars_Kid_Remix.wmv|1051734013:/random/video/Star_War
12.34.246.7     1051734043:/archive/2003/04/29/star_war.shtml
12.42.55.174    1051733934:/random/video/Star_Wars_Kid_Remix.wmv|1051733981:/random/video/Star_War
129.8.100.123   1051734019:/archive/2003/04/29/star_war.shtml
131.212.155.170 1051734004:/random/video/Star_Wars_Kid.wmv
131.229.113.229 1051733829:/random/video/Star_Wars_Kid.wmv
133.11.36.26    1051733907:/archive/2003/03/14/dixie_ch.shtml
133.11.36.42    1051733922:/archive/2003/02/21/gift_upg.shtml
133.11.36.46    1051734036:/random/text/MAD.Magazine.418.june.2002-een.nfo.txt
133.11.36.50    1051733959:/random/archive/pooyan.zip
137.244.215.2   1051733993:/archive/2003/04/29/star_war.shtml|1051734048:/random/video/Star_Wars_K
141.158.70.90   1051733902:/|1051733927:/random/video/Star_Wars_Kid.wmv|1051733943:/random/video/S
142.179.181.67  1051733947:/random/video/Star_Wars_Kid.wmv
142.205.240.19  1051733940:/random/video/Star_Wars_Kid.wmv
144.13.238.225  1051733894:/archive/2003/04/29/star_war.shtml
150.134.71.174  1051734018:/random/video/Star_Wars_Kid.wmv|1051734027:/random/video/Star_Wars_Kid.
150.134.71.99   1051733847:/|1051734020:/random/video/Star_Wars_Kid.wmv|1051734024:/random/video/S
154.11.61.211   1051733899:/random/video/Star_Wars_Kid.wmv|1051733996:/random/video
154.5.248.92    1051733824:/random/video/Star_Wars_Kid.wmv|1051733852:/random/video/Star_Wars_Kid.
159.124.6.148   1051733931:/random/video/Star_Wars_Kid_Remix.wmv
160.81.221.42   1051733919:/archive/2003/04/29/star_war.shtml|1051733959:/random/video/Star_Wars_K
170.20.11.59    1051733822:/archive/2003/04/29/star_war.shtml|1051733950:/random/video/Star_Wars_K
192.44.136.103  1051733916:/random/video/Star_Wars_Kid_Remix.wmv|1051733929:/random/video/Star_War
195.92.168.170  1051733994:/
198.134.100.10  1051733907:/projects/domains/domains.net.txt
199.181.174.146 1051733848:/random/video/Star_Wars_Kid_Remix.wmv|1051733851:/random/video/Star_War
199.91.33.254   1051733829:/random/video/Star_Wars_Kid.wmv
202.79.214.6    1051733969:/index.rdf
204.50.142.36   1051734031:/random/video/Star_Wars_Kid.wmv
204.92.90.254   1051733977:/random/video/Star_Wars_Kid.wmv|1051733988:/random/video/Star_Wars_Kid.
205.150.14.236  1051733870:/random/video/Star_Wars_Kid.wmv|1051733878:/random/video/Star_Wars_Kid.
206.154.118.2   1051733873:/random/video/Star_Wars_Kid.wmv
207.171.180.101 1051733941:/
209.98.52.113   1051733931:/random/video/Star_Wars_Kid.wmv|1051733938:/random/video/Star_Wars_Kid.
216.201.147.173 1051733887:/random/video/Star_Wars_Kid_Remix.wmv
216.70.35.213   1051733916:/random/video/Star_Wars_Kid.wmv
24.114.8.26     1051733935:/random/video/Star_Wars_Kid.wmv
24.154.8.105    1051733914:/random/video/Star_Wars_Kid.wmv|1051733923:/random/video/Star_Wars_Kid.
24.95.49.63     1051733977:/random/video/Star_Wars_Kid.wmv|1051733986:/random/video/Star_Wars_Kid.
64.173.152.130  1051733838:/archive/2003/02/19/coachell.shtml
64.207.4.108    1051733875:/random/video/Star_Wars_Kid.wmv
64.74.63.8      1051733899:/archive/2003/02/06/the_1250.shtml
65.166.26.146   1051733868:/archive/2003/03/27/open_cdr.shtml|1051733958:/|1051733960:/
65.206.239.163  1051734048:/index.rdf
66.151.218.42   1051733844:/archive/2003/04/29/star_war.shtml|1051733881:/random/video/Star_Wars_K
```

We use the secondary sort so that each visit is in strict order of time within a session.

You might ask why that is necessary — surely each mapper reads the lines in order? Yes, but you have no control over what order the mappers run, or where their input begins and ends.

This script *will* accumulate multiple visits of a page.

TODO: say more about the secondary sort.

Page-Page similarity

What can you do with the sessionized logs? Well, each row lists a user on the left and a bunch of pages on the right.

We've been thinking about that as a table, but it's also a graph!

```
$ cat data/swk-100.tsv      | ./breadcrumbs.rb --map | sort | ./breadcrumb_edges.rb --reduce | wu-
137.244.215.2   /archive/2003/04/29/star_war.shtml      /random/video/Star_Wars_Kid.wmv
141.158.70.90   /                                       /random/video/Star_Wars_Kid.wmv
150.134.71.99   /                                       /random/video/Star_Wars_Kid.wmv
154.11.61.211   /random/video/Star_Wars_Kid.wmv         /random/video
154.5.248.92    /random/video/Star_Wars_Kid.wmv         /
160.81.221.42   /archive/2003/04/29/star_war.shtml      /random/video/Star_Wars_Kid.wmv
170.20.11.59    /archive/2003/04/29/star_war.shtml      /random/video/Star_Wars_Kid.wmv
65.166.26.146   /archive/2003/03/27/open_cdr.shtml      /
66.151.218.42   /archive/2003/04/29/star_war.shtml      /random/video/Star_Wars_Kid_Remix.wmv
67.119.139.98   /archive/2003/04/21/classmat.shtml      /random/video/Star_Wars_Kid_Remix.wmv
67.119.139.98   /archive/2003/04/21/classmat.shtml      /
67.119.139.98   /random/video/Star_Wars_Kid_Remix.wmv    /
68.10.181.55    /archive/2003/04/29/star_war.shtml      /favicon.ico
68.10.181.55    /archive/2003/04/29/star_war.shtml      /
68.10.181.55    /favicon.ico                            /
68.34.219.93    /archive/2003/04/29/star_war.shtml      /random/video/Star_Wars_Kid.wmv
```

You'll learn more about this in the chapter on Processing Graphs, but

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.

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.

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.

```
      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

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.

CHAPTER 10

Why Hadoop Works

- Locality of reference and the speed of light
- Disk is the new tape — Random access on bulk storage is very slow
- Fun — Resilient distributed frameworks have traditionally been very conceptually complex, where by complex I mean "MPI is a soul-sucking hellscape"

Disk is the new tape

Doug Cutting's example comparing speed of searching by index vs. searching by full table scan

see ch05, *the rules of scaling*.

Hadoop is Secretly Fun

Walk into any good Hot Rod shop and you'll see a sign reading "Fast, Good or Cheap, choose any two". Hadoop is the first distributed computing framework that can claim "Simple, Resilient, Scalable, choose all three".

The key, is that simplicity + decoupling + embracing constraint unlocks significant power.

Heaven knows Hadoop has its flaws, and its codebase is long and hairy, but its core is

- speculative execution
- compressed data transport
- memory management of buffers
- selective application of combiners

- fault-tolerance and retry
- distributed counters
- logging
- serialization

Economics:

Say you want to store a billion objects, each 10kb in size. At commodity cloud storage prices in 2012, this will cost roughly [^1]

- $250,000 a month to store in RAM
- $ 25,000 a month to store it in a database with a 1:10 ram-to-storage ratio
- $ 1,500 a month to store it flat on disk

CPU

A 30-machine cluster with 240 CPU cores, 2000 GB total RAM and 50 TB of raw disk [^1]:

- purchase: (→ find out purchase price)
- cloud: about $60/hr; $10,000 to run for 8 hours a day every work day.

By contrast, it costs [^1]

- $ 1,600 a month to hire an intern for 25 hours a week
- $ 10,000 a month to hire an experienced data scientist, if you can find one

In a database world, the dominant cost of an engineering project is infrastructure. In a hadoop world, the dominant cost is engineers.

[^1] I admit these are not apples-to-apples comparisons. But the differences are orders of magnitude: subtly isn't called for

Notes

[1] "Linear" means that increasing your cluster size by a factor of `S` increases the rate of progress by a factor of `S` and thus solves problems in `1/S` the amount of time.

[2] Even if you did find yourself on a supercomputer, Einsten and the speed of light take all the fun out of it. Light travels about a foot per nanosecond, and on a very fast CPU each instruction takes about half a nanosecond, so it's impossible to talk to a machine

more than a hands-breadth away. Even with all that clever hardware you must always be thinking about locality, which is a Hard Problem. The Chimpanzee Way says "Do not solve Hard Problems. Turn a Hard Problem into a Simple Problem and solve that instead"

[3] *http://en.wikipedia.org/wiki/K_computer*

[4] over and over in scalable systems you'll find the result of Simplicity, Decoupling and Embracing Constraint is great power.

[5] you may be saying to yourself, "Self, I seem to recall my teacher writing on the chalkboard that sorting records takes more than linear time — in fact, I recall it is `O(N log N)`". This is true. But in practice you typically buy more computers in proportion to the size of data, so the amount of data you have on each computer remains about the same. This means that the sort stage takes the same amount of time as long as your data is reasonably well-behaved. In fact, because disk speeds are so slow compared to RAM, and because the merge sort algorithm is very elegant, it takes longer to read or process the data than to sort it.

CHAPTER 11
Sampling

- Random sample:
  - fixed size of final sample
  - fixed probability (binomial) for each element
  - spatial sampling
  - with/without replacement
  - weighted
  - by interestingness
  - stratified: partition important features into bins, and sample tastefully to achieve a smooth selection across bins. Think of density of phase space
  - consistent sample: the same sampling parameters on the same population will always return the same sample.
- Algorithms:
  - iterative
  - batch
  - scan
  - reservoir
- graph:
  - sample to preserve connectivity
  - sample to preserve local structure
  - sample to preserve global representation
- random variates

— Ziggurat Algorithm to use a simple lookup table to accelerate generation of complex distributions

We're not going to worry about extracting samples larger than fit on one reducer.

Consistent Random Sampling

The simplest kind of sample is a uniform sample selecting a fraction p of the full dataset.

The naive way take is to generate a random number and select each line if it is less than the probability p. Don't do this.

You want your job to be deterministic. In the large, so that it is predictable and debuggable. in the small, a mapper may be re-tried if the attempt fails, or while doing speculative execution.

What we'll do instead is use a standard digest function (for example, the MD5 hash or murmur hash). A digest function turns any key into a fixed-size number, with the important property that any small change in the input string results in an arbitrarily large change in the output number. It's deterministic (the same input always gives the same output) but effectively washes out all information from the input string.

Then, rather than compare a random number against a fraction, we'll turn the digest into an integer (by treating the lowest 64 bits as an integer) and compare it to that fraction of the largest 64-bit number.

<remark>confirm that it's LSB not MSB we want</remark>

A [*http://github.com/mrflip/wukong/blob/master/examples/sample_records.rb* Ruby example] is available in the wukong examples:

```
#
# Probabilistically emit some fraction of record/lines
#
# Set the sampling fraction at the command line using the
#   --sampling_fraction=
# option: for example, to take a random 1/1000th of the lines in huge_files,
#  ./examples/sample_records.rb --sampling_fraction=0.001 --go huge_files sampled_files
#
class Mapper < Wukong::Streamer::LineStreamer
  include Wukong::Streamer::Filter

def intialize(*)
  super
  get_sampling_threshold
end
```

```
# randomly decide to emit +sampling_fraction+ fraction of lines
def emit? line
   digest_i < sampling_threshold
end

protected

# Uses the sampling_fraction, a real value between 0 and 1 giving the fraction of lines to
# emit.  at sampling_fraction=1 all records are emitted, at 0 none are.
#
# @return [Integer] between 0 and MAX_DIGEST; values below the sampling_threshold should be emitte
def get_sampling_threshold
  if not options[:sampling_fraction] then raise ArgumentError, "Please supply a --sampling_fractio
  @sampling_threshold = (Float(options[:sampling_fraction]) * MAX_DIGEST).to_i
end

  # @return [Integer] the last 64 bits of the record's md5 hash
  def digest_i(record)
    Digest::MD5.digest(record.to_s).unpack('Q*').last
  end
  # One more than the largest possible digest int that digest_i will return.
  MAX_DIGEST = 2 ** 64
end

# Execute the script with nil reducer
Script.new( Mapper, nil ).run
```

References: * See this rapleaf blog post for why randomness is considered harmful.

Random Sampling using strides

Another, often faster, way of doing random sampling is to generate a geometrically-distributed (rather than uniformly-distributed) sampling series For each value `R`, Your mapper skips `R` lines and

see section on statistics for how to get a geometrically-distributed number.

Constant-Memory "Reservoir" Sampling

Want to generate a sample of fixed size N_s — say, 1000 arbitrary records — no matter how large or small the dataset. (Clearly if it is smaller than N_s, you will emit the full dataset).

Suppose you assigned every record an arbitrary sample key, and sorted on that key. Choosing the first N_s records would be a fair way to get our sample. In fact, this is how most card games work: shuffle the records (cards) into an arbitrary order, and draw a fixed-size batch of cards from the collection.

But! of course, a total sort is very expensive. As you may guess, it's unnecessary.

Each mapper creates a "reservoir", of size N_s, for the rows it will select. Add each record to the reservoir, and if there are more than N_s occupants, reject the record with highest sample index. (in practice, you won't even add the record if it would be that highest record). A Fibonacci heap (implementing a priority queue) makes this very efficient

Ruby's stdlib has a `SortedSet` class — a Set that guarantees that it's element are yielded in sorted order (according to the return values of their `#<=>` methods) when iterating over them.

Each mapper outputs the sampling index of each preserved row as the key, and the rest of the row as the value;

It's essential that you keep the sampling index given by the first pass.

References

- Random Sampling from Databases, Frank Olken, 1993
- containers:
  - RBTree for ruby
  - Priority Queue
- Stack Overflow: How to pick random (small) data samples using Map/Reduce?, answer by Bkkbrad

CHAPTER 12

Hadoop Execution in Detail

Launch

When you launch a job (with `pig`, `wukong run`, or `hadoop jar`), it starts a local process that

- prepares a synthesized configuration from config files of the program and the machine (`core-site.xml`, `hdfs-site.xml`, `mapred-site.xml`).
- asks the jobtracker for a job ID
- pushes your program and its assets (jars, script files, distributed cache contents) into the job's directory on the HDFS.
- asks the jobtracker enqueue the job.

After a few seconds you should see the job appear on the jobtracker interface. The jobtracker will begin dispatching the job to workers with free slots, as directed by its scheduler [1]. It knows where all the input blocks are, and will try to launch each task on the same machine as its input ("bring the compute to the data"). The jobtracker will tell you how many map tasks are "local" (launched on a different machine than its input); if it's not harmlessly small, see "Many non-local mappers" (page 102).

The launching process doesn't take many resources, so for a development cluster it's OK to launch a job from a worker machine. Terminating the launch process won't affect the job execution, but its output is useful. To record its output even if you log off, use the `nohup` command:

```
nohup [...normal launch command...] >> /tmp/my_job-`date +%F`.log 2>&1 &
```

1. unless your cluster is heavily used by multiple people, the default scheduler is fine. If fights start breaking out, quickly consult (TODOREF Hadoop Operations) for guidance on the other choices

Run `tail -f /tmp/my_job-*.log` to keep watching the job's progress.

The job draws its default configuration from the *launch* machine's config file. Make sure those defaults doesn't conflict with appropriate values for the workers that will actually execute the job! One great way to screw this up is to launch a job from your dev machine, go to dinner and come back to find it using one reducer and a tiny heap size. Another is to start your job from a master that is provisioned differently from the workers.

Split

Input files are split and assigned to mappers.

Each mapper will receive a chunk bounded by:

- The file size — normally, each mapper handles at most one file (and typically, one part of a very large file). (footnote: Pig will pre-combine small files into single map inputs with the `pig.splitCombination` commandline parameter.)
- Min split size — up to the size of each file, you can force hadoop to make each split larger than `mapred.min.split.size`
- Block size — the natural unit of data to feed each map task is the size of an HDFS file chunk; this is what lets Hadoop "bring the compute to the data".
- Input format — some input formats are non-splittable (by necessity, as for some compression formats; or by choice, when you want to enforce no file splits). [2]

Exercises:

- Create a 2GB file having a 128MB block size on the HDFS. Run `wu-stream cat cat --min_split_mb=1900` on it. How many map tasks will launch? What will the "non-local" cell on the jobtracker report? Try it out for 1900, and also for values of 128, 120, 130, 900 and 1100.

2. Paraphrasing the Hadoop FAQ, to make a *non-splittable* FileInputFormat, your particular input-format should return false for the isSplittable call. If you would like the whole file to be a single record, you must also implement a RecordReader interface to do so — the default is LineRecordReader, which splits the file into separate lines. The other, quick-fix option, is to set mapred.min.split.size to large enough value.

Mappers

Hadoop Streaming (Wukong, MrJob, etc)

If it's a Hadoop "streaming" job (Wukong, MrJob, etc), the child process is a Java jar that itself hosts your script file:

- it forks the script in a new process. The child ulimit applies to this script, but the heap size and other child process configs do not.
- passes all the Hadoop configs as environment variables, changing `.` dots to _ underbars. Some useful examples:
  - `map_input_file` — the file this task is processing
  - `map_input_start` — the offset within that file
  - `mapred_tip_id` — the task ID. This is a useful ingredient in a unique key, or if for some reason you want each mapper's output to go to a distinct reducer partition.
- directs its input to the script's `STDIN`. Not all input formats are streaming-friendly.
- anything the script sends to its `STDOUT` becomes the jar's output.

forks yet another

Once the maps start, it's normal for them to seemingly sit at 0% progress for a little while: they don't report back until a certain abount of data has passed through. Annoyingly, jobs with gzipp'ed input will remain mute until they are finished (and then go instantly from 0 to 100%).

exercise: Write a mapper that ignores its input but emits a configurable number of bytes, with a configurable number of bytes per line. Run it with one mapper and one reducer. Compare what happens when the output is just below, and just above, each of these thresholds: - the HDFS block size - the mapper sortbuf spill threshold - the mapper sortbuf data threshold - the mapper sortbuf total threshold

Speculative Execution ======

For exploratory work, it's worth

Choosing a file size

Jobs with Map and Reduce

For jobs that have a reducer, the total size of the output dataset divided by the number of reducers implies the size of your output files [3]. Of course your working dataset is less than a few hundred MB this doesn't matter.

If your working set is large enough to care and less than about 10 TB, size your reduce set for files of about 1 to 2 GB.

- *Number of mappers*: by default, Hadoop will launch one mapper per HDFS block; it won't assign more than one file to each mapper [4]. More than a few thousand
- *Reducer efficiency*: as explained later (TODO: ref reducer_size), your reducers are most efficient at 0.5 to 2 GB.
- *HDFS block size*: >= 1-2 GB — a typically-seen hadoop block size is 128 MB; as you'll see later, there's a good case for even larger block sizes. You'd like each file to hold 4 or more blocks.
- *your network connection* (< 4GB): a mid-level US internet connection will download a 4 GB file segment in about 10 minutes, upload it in about 2 hours.
- *a DVD*: < 4 GB — A DVD holds about 4GB. I don't know if you use DVDs still, but it's a data point.
- *Cloud file stores*: < 5 GB — The Amazon S3 system now allows files greater than 5 GB, but it requires a special multi-part upload transfer.
- *Browsability*: a 1 GB file has about a million 1kB records.

Even if you don't find any of those compelling enough to hang your hat on, I'll just say that files of 2 GB are large enough to be efficient and small enough to be manageable; they also avoid those upper limits even with natural variance in reduce sizes.

If your dataset is

Mapper-only jobs

There's a tradeoff:

If you set your min-split-size larger than your block size, you'll get non-local map tasks, which puts a load on your network.

3. Large variance in counts of reduce keys not only drives up reducer run times, it causes variance in output sizes; but that's just insult added to injury. Worry about that before you worry about the target file size.
4. Pig has a special option to roll up small files

However, if you let it launch one job per block, you'll have two problems. First, one mapper per HDFS block can cause a large number of tasks: a 1 TB input dataset of 128 MB HDFS blocks requires 8,000 map tasks. Make sure your map task runtimes aren't swamped by job startup times and that your jobtracker heap size has been configured to handle that job count. Secondly, if your job is ever-so-slightly expansive — if it turns a 128 MB input block into a 130 MB output file — then you will double the block count of the dataset. It takes twice the actual size to store on disk and implies twice the count of mappers in subsequent stages.

My recommendation: (TODO: need to re-confirm with numbers; current readers please take with a grain of salt.)

To learn more, see the

CHAPTER 13

Pathology of Tuning (aka "when you should touch that dial")

Mapper

A few map tasks take noticably longer than all the rest

Typically, the only good reason for a map task to run much longer than its peers is if it's processing a lot more data.

The jobtracker assigns blocks in decreasing order of size to help prevent the whole job waiting on one last mapper. If your input format is non-splitable (eg it's `.bz2` compressed), and some files are huge, those files will take proportionally longer. If these are so imbalanced as to cause trouble, I can't offer much more than a) don't write imbalance files, or b) switch to a splitable format.

If some map tasks are very slow, but aren't processing proportionally more data, look for the following:

- Check the logs of the slow tasks — did they hit a patch of bad records and spend all their time rescuing a ton of exceptions?
- Some records take far longer to process than others — for example a complex regexp (regular expression) that requires a lot of backtracking.
- If the slow tasks always occur on the same machine(s), they might be failing.
- Check the logs for both the child processes and the daemons — are there timeouts or other errrors?

Tons of tiny little mappers

For jobs with a reduce step, the number of map tasks times the heap size of each mapper should be about twice the size of the input data.

CombineFileInputFormat

…TODO…

Many non-local mappers

- A prior stage used one (or very few) reducers
- You recently enlarged your cluster
- HDFS is nearly full
- Overloaded data nodes

Map tasks "spill" multiple times

Each mapper is responsible for sorting the individual records it emits, allowing the reducers to just do a merge sort. It does so by filing records into a fixed-size sort buffer (the analog of the inbox sorter on the back of each elephant). Each time the sort buffer overflows, the mapper will "spill" it to disk; once finished, it merge sorts those spills to produce its final output. Those merge passes are not cheap — the full map output is re-read and re-written. If the midflight data size is several times your allocable heap, then those extra merge passes are necessary: you should smile at how well Hadoop is leveraging what RAM you have.

However, if a mapper's midflight size even slightly exceeds the sort buffer size, it will trigger an extra spill, causing a 20-40% performance hit.

On the jobtracker, check that the

Job output files that are each slightly larger than an HDFS block

If your mapper task is slightly expansive (outputs more data than it reads), you may end up with an output file that for every input block emits one full block and almost-empty block. For example, a task whose output is about 120% of its input will have an output block ratio of 60% — 40% of the disk space is wasted, and downstream map tasks will be inefficient.

1. Check this by comparing (TODO: grab actual terms) HDFS bytes read with mapper output size.

You can check the block ratio of your output dataset with `hadoop fsck -blocks`

(TODO: capture output)

If your mapper task is expansive and the ratio is less than aobut 60%, you may want to set a min split size of about

Alternatively, turn up the min split size on the *next* stage, sized so that it

Reducer

Tons of data to a few reducers (high skew)

- Did you set a partition key?
- Can you use a finer partition key?
- Can you use a combiner?
- Are there lots of records with a NULL key?
  - Here's a great way to get null keys: `j = JOIN a BY akey LEFT OUTER, b by bkey; res = FOREACH j GENERATE bkey AS key, a::aval, b::bval; grouped = GROUP res BY key;`. Whenever there's a key in `a` with no match in `b`, `bkey` will be null and the final `GROUP` statement will be painful. You probably meant to say `... GENERATE akey AS key .....`
- Does your data have intrinsically high skew? — If records follow a long-tail distribution,
- Do you have an "LA and New York" problem? If you're unlucky, the two largest keys might be hashed to the same reducer. Try running with one fewer reducers — it should jostle the keys onto different machines.

If you have irrevocably high skew, Pig offers a `skewed join` operator.

Reducer merge (sort+shuffle) is longer than Reducer processing

Output Commit phase is longer than Reducer processing

Way more total data to reducers than cumulative cluster RAM

If you are generating a huge amount of midflight data for a merely-large amount of reducer output data, you might be a candidate for a better algorithm.

In the graph analytics chapter, we talk about "counting triangles": how many of your friends-of-friends are also direct friends? More than a million people follow Britney Spears and Justin Bieber on Twitter. If they follow each other (TODO: verify that they

do), the "obvious" way of counting shared friends will result in trillions (`10^12`) of candidates — but only millions if results. This is an example of "multiplying the short tails" of two distributions. The graph analytics chapter shows you one pattern for handling this.

If you can't use a better algorithm, then as they said in Jaws: "you're going to need a bigger boat".

System

Excessive Swapping

- don't use swap — deprovision it.
- if you're going to use swap, set swappiness and overcommit

Otherwise, treat the presence of *any* significant swap activity as if the system were "Out of Memory / No C+B reserve" (page 104).

Out of Memory / No C+B reserve

Your system is out of memory if any of the following occurs:

- there is no remaining reserve for system caches+buffers
- significant swap activity
- OOM killer (the operating system's Out Of Memory handler) kills processes

For Hadoop's purposes, if the OS has no available space for caches+buffers, it has already run out of system RAM — even if it is not yet swapping or OOMing

- check overcommit

You may have to reduce slots, or reduce heap per slot.

Stop-the-world (STW) Garbage collections

STW garbage collection on a too-large heap can lead to socket timeouts, increasing the load and memory pressure on other machines, leading to a cascading degradation of the full cluster.

Checklist

- Unless your map task is CPU-intensive, mapper task throughput should be comparable to baseline throughput.

- The number of non-local map tasks is small.
- Map tasks take more than a minute or so.
- Either *Spilled bytes* and *mapper output bytes* are nearly equal, or *Spilled bytes* is three or more times *mapper output bytes*.
- The size of each output file is not close-to-but-above the HDFS block size

Other

Basic Checks

- enough disk space
- hadoop native libraries are discovered (`org.apache.hadoop.util.NativeCodeLoader: Loaded the native-hadoop library` appears in the logs).
- midstream data uses snappy compression (`org.apache.hadoop.io.compress.snappy.LoadSnappy: Snappy native library is available` appears in the logs).

CHAPTER 14

Hadoop Metrics

The USE Method applied to Hadoop

There's an excellent methodology for performance analysis known as the "USE Method": For every resource, check Utilization, Saturation and Errors" [1]:

- utilization — How close to capacity is the resource? (ex: CPU usage, % disk used)
- saturation — How often is work waiting? (ex: network drops or timeouts)
- errors — Are there error events?

For example, USE metrics for a supermarket cashier would include:

- checkout utilization: flow of items across the belt; constantly stopping to look up the price of tomatoes or check coupons will harm utilization.
- checkout saturation: number of customers waiting in line
- checkout errors: calling for the manager

The cashier is an I/O resource; there are also capacity resources, such as available stock of turkeys:

- utilization: amount of remaining stock (zero remaining would be 100% utilization)
- saturation: in the US, you may need to sign up for turkeys near the Thanksgiving holiday.
- errors: spoiled or damaged product

1. developed by Brendan Gregg for system performance tuning, modified here for Hadoop

As you can see, it's possible to have high utilization and low saturation (steady stream of customers but no line, or no turkeys in stock but people happily buying ham), low utilization and high saturation (a cashier in training with a long line), or any other combination.

Why the USE method is useful

It may not be obvious why the USE method is novel — "Hey, it's good to record metrics" isn't new advice.

What's novel is its negative space. "Hey, it's good *enough* to record this *limited* set of metrics" is quite surprising. Here's "USE method, the extended remix": *For each resource, check Utilization, Saturation and Errors. This is the necessary and sufficient foundation of a system performance analysis, and advanced effort is only justifiable once you have done so*.

There are further benefits:

- An elaborate solution space becomes a finite, parallelizable, delegatable list of activities.
- Blind spots become obvious. We once had a client issue where datanodes would go dark in difficult-to-reproduce circumstances. After much work, we found that persistent connections from Flume and chatter around numerous small files created by Hive were consuming all available datanode handler threads. We learned by pain what we could have learned by making a list — there was no visibility for the number of available handler threads.
- Saturation metrics are often non-obvious, and high saturation can have non-obvious consequences. (For example, the hard lower bound of TCP throughput)
- It's known that Teddy Bears make superb level-1 support techs, because being forced to deliver a clear, uninhibited problem statement often suggests its solution. To some extent any framework this clear and simple will carry benefits, simply by forcing an organized problem description.

<remark>http://en.wikipedia.org/wiki/Rubber_duck_debugging</remark>

Look for the Bounding Resource

The USE metrics described below help you to identify the limiting resource of a job; to diagnose a faulty or misconfigured system; or to guide tuning and provisioning of the base system.

Improve / Understand Job Performance

Hadoop is designed to drive max utilization for its *bounding resource* by coordinating manageable saturation_ of the resources in front of it.

The "bounding resource" is the fundamental limit on performance — you can't process a terabyte of data from disk faster than you can read a terabyte of data from disk. k

- disk read throughput
- disk write throughput
- job process CPU
- child process RAM, with efficient utilization of internal buffers
- If you don't have the ability to specify hardware, you may need to accept "network read/write throughput" as a bounding resource during replication.

At each step of a job, what you'd like to see is very high utilization of exactly *one* bounding resource from that list, with reasonable headroom and managed saturation for everything else. What's "reasonable"? As a rule of thumb, utilization above 70% in a non-bounding resource deserves a closer look.

Diagnose Flaws

Balanced Configuration/Provisioning of base system

Resource List

Please see the ??? for definitions of terms. I've borrowed many of the system-level metrics from Brendan Gregg's Linux Checklist; visit there for a more-detailed list.

Table 14-1. USE Method Checklist

resource	type	metric	instrument
CPU-like concerns			
CPU	utilization	system CPU	`top`/`htop`: CPU %, overall
	utilization	**job process CPU**	`top`/`htop`: CPU %, each child process
	saturation	max user processes	`ulimit -u` ("noproc" in `/etc/security/limits.d/...`)
mapper slots	utilization	mapper slots used	jobtracker console; impacted by `mapred.tasktracker.map.tasks.maximum`
mapper slots	saturation	mapper tasks waiting	jobtracker console; impacted by scheduler and by speculative execution settings
	saturation	task startup overhead	???

resource	type	metric	instrument
	saturation	combiner activity	jobtracker console (TODO table cell name)
reducer slots	utilization	reducer slots used	jobtracker console; impacted by `mapred.tasktracker.reduce.tasks.maximum`
reducer slots	saturation	reducer tasks waiting	jobtracker console; impacted by scheduler and by speculative execution and slowstart settings
Memory concerns	_		_
memory capacity	utilization	total non-OS RAM	`free`, `htop`; you want the total excluding caches+buffers.
	utilization	child process RAM	`free`, `htop`: "RSS"; impacted by `mapred.map.child.java.opts` and `mapred.reduce.child.java.opts`
	utilization	JVM old-gen used	JMX
	utilization	JVM new-gen used	JMX
memory capacity	saturation	swap activity	`vmstat 1` - look for "r" > CPU count.
	saturation	old-gen gc count	JMX, gc logs (must be specially enabled)
	saturation	old-gen gc pause time	JMX, gc logs (must be specially enabled)
	saturation	new-gen gc pause time	JMX, gc logs (must be specially enabled)
mapper sort buffer	utilization	record size limit	announced in job process logs; controlled indirectly by `io.sort.record.percent`, spill percent tunables
	utilization	record count limit	announced in job process logs; controlled indirectly by `io.sort.record.percent`, spill percent tunables
mapper sort buffer	saturation	spill count	spill counters (jobtracker console)
	saturation	sort streams	io sort factor tunable (`io.sort.factor`)
shuffle buffers	utilization	buffer size	child process logs
	utilization	buffer %used	child process logs
shuffle buffers	saturation	spill count	spill counters (jobtracker console)
	saturation	sort streams	merge parallel copies tunable `mapred.reduce.parallel.copies` (TODO: also `io.sort.factor`?)
OS caches/buffers	utilization	total c+b	`free`, `htop`
disk concerns	_		_
system disk I/O	utilization	req/s, read	`iostat -xz 1` (system-wide); `iotop` (per process); `/proc/{PID}/sched` "se.statistics.iowait_sum"
	utilization	req/s, write	`iostat -xz 1` (system-wide); `iotop` (per process); `/proc/{PID}/sched` "se.statistics.iowait_sum"
	utilization	MB/s, read	`iostat -xz 1` (system-wide); `iotop` (per process); `/proc/{PID}/sched` "se.statistics.iowait_sum"

resource	type	metric	instrument
	utilization	MB/s, write	`iostat -xz 1` (system-wide); `iotop` (per process); `/proc/{PID}/sched` "se.statistics.iowait_sum"
system disk I/O	saturation	queued requests	`iostat -xnz 1`; look for "avgqu-sz" > 1, or high "await".
system disk I/O	errors		`/sys/devices/.../ioerr_cnt`; `smartctl`, `/var/log/messages`
network concerns	_		_
network I/O	utilization		`netstat`; `ip -s {link}`; `/proc/net/{dev}` — RX/TX throughput as fraction of max bandwidth
network I/O	saturation		`ifconfig` ("overruns", "dropped"); `netstat -s` ("segments retransmited"); `/proc/net/dev` (RX/TX "drop")
network I/O	errors	interface-level	`ifconfig` ("errors", "dropped"); `netstat -i` ("RX-ERR"/"TX-ERR"); `/proc/net/dev` ("errs", "drop")
		request timeouts	daemon and child process logs
handler threads	utilization	nn handlers	(TODO: how to measure) vs `dfs.namenode.handler.count`
	utilization	jt handlers	(TODO: how to measure) vs
	utilization	dn handlers	(TODO: how to measure) vs `dfs.datanode.handler.count`
	utilization	dn xceivers	(TODO: how to measure) vs `dfs.datanode.max.xcievers
framework concerns	_		_
disk capacity	utilization	system disk used	`df -bM`
	utilization	HDFS directories	`du -smc /path/to/mapred_scratch_dirs` (for all directories in `dfs.data.dir`, `dfs.name.dir`, `fs.checkpoint.dir`)
	utilization	mapred scratch space	`du -smc /path/to/mapred_scratch_dirs` (TODO scratch dir tunable)
	utilization	total HDFS free	namenode console
	utilization	open file handles	`ulimit -n` ("nofile" in `/etc/security/limits.d/...`)
job process	errors		stderr log
	errors		stdout log
	errors		counters
datanode	errors		
namenode	errors		
secondarynn	errors		
tasktracker	errors		
jobtracker	errors		

Metrics in bold are critical resources — you would like to have exactly one of these at its full sustainable level

Ignore past here please

Ignore past here please

Ignore past here please

See What's Happening

JMX (Java Monitoring Extensions)

Whenever folks are having "my programming language is better than yours", the Java afficianado can wait until somebody's scored a lot of points and smugly play the "Yeah, but Java has JMX" card. It's simply amazing.

Deep Explanation of JMX

VisualVM is a client for examining JMX metrics.

If you're running remotely (as you would be on a real cluster), here are instructions for [2]

There is also an open-source version, MX4J.

- You need a file called `jmxremote_optional.jar`, from Oracle Java's "Remote API Reference Implementation"; download from Oracle. They like to gaslight web links, so who knows if that will remain stable.
- Add it to the classpath of

(on a mac, in `/usr/bin/jvisualvm`)

- Run visualvm `visualvm -cp:a /path/to/jmxremote_optional.jar` (on a mac, add a *j* in front: `/usr/bin/jvisualvm ...`).

You will need to install several plugins; I use `VisualVM-Extensions`, `VisualVM-MBeans`, `Visual GC`, `Threads Inspector`, `Tracer-{Jvmstat,JVM,Monitpr} Probes`.

Poor-man's profiling

To find the most CPU-intensive java threads:

```
ps -e HO ppid,lwp,%cpu --sort %cpu | grep java | tail; sudo killall -QUIT java
```

2. Thanks to Mark Feeney for the writeup

The -QUIT sends the SIGQUIT signal to Elasticsearch, but QUIT doesn't actually make the JVM quit. It starts a Javadump, which will write information about all of the currently running threads to standard out.

Other tools:

- Ganglia
- BTrace

Rough notes

Metrics:

- number of spills
- disk {read,write} {req/s,MB/s}
- CPU % {by process}
- GC
  - — heap used (total, %)
  - — new gen pause
  - — old gen pause
  - — old gen rate
  - — STW count
- system memory
  - — resident ram {by process}
  - — paging
- network interface
  - — throughput {read, write}
  - — queue
- handler threads
  - — handlers
  - — xceivers *
- mapper task CPU
- mapper taks Network interface — throughput Storage devices — throughput, capacity Controllers — storage, network cards Interconnects — CPUs, memory, throughput
- disk throughput

- handler threads
- garbage collection events

Exchanges:

* * shuffle buffers — memory for disk * gc options — CPU for memory

If at all possible, use a remote monitoring framework like Ganglia, Zabbix or Nagios. However clusterssh or its OSX port along with the following commands will help

Exercises

Exercise 1: start an intensive job (eg <remark>TODO: name one of the example jobs</remark>) that will saturate but not overload the cluster. Record all of the above metrics during each of the following lifecycle steps:

- map step, before reducer job processes start (data read, mapper processing, combiners, spill)
- near the end of the map step, so that mapper processing and reducer merge are proceeding simultaneously
- reducer process step (post-merge; reducer processing, writing and replication proceeding)

Exercise 2: For each of the utilization and saturation metrics listed above, describe job or tunable adjustments that would drive it to an extreme. For example, the obvious way to drive shuffle saturation (number of merge passes after mapper completion) is to bring a ton of data down on one reducer — but excessive map tasks or a `reduce_slow start_pct` of 100% will do so as well.

CHAPTER 15

Data Formats and Schemata

there's only three data serialization formats you should use: TSV, JSON, or Avro.

TESTING

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 (`
` 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

HBase Data Model

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. 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.

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.

- What question do you want to ask that *must* happen in less than 30 milliseconds?
- There are a set of related questions — which would you like to be fast?
- Given that — here's the rest of the questions you can ask?

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 day-care 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.

Keep it Stupidly Simple

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 [1]. 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.

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 16-1. 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 [2], 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.

1. First, join on the pagerank table (see TODO: ref) to attach a "prominence" to each page. Next, write a map-reduce job: 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.

2. 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.

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 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 [3].

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.

3. many relational databases accomplish the same end wtih "vertical partitioning".

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.

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`)

use packed integer quadkeys — space efficient

When you are using this "Rows as Columns" technique, make sure you set "scanner caching" on. Scanner caching [4] creates a read buffer 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.

4. confusing name: it's "Caching of rows found by scanner", not "Caching of scanner objects"

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 16-2. 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

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 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* [5]. 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.

5. 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*?)

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.

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

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?) [6].

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".

6. The TTL will only work if you're playing honest with the timestamps — you can't use it with the most-frequent URL table

Table 16-3. 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)	-		
ip_tbs	`ip-timebucket`				

IP Address Geolocation

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 16-4. 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 16-5. 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 16-6. 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.

Review of HBase options

- column families — use only one, unless you need both full-row *and* partial-row access. Even still, high-performance tables shouldn't use more than a few colum families.
- `BLOOMFILTER` — `false` except for a high-impact table with many misses. Monitor the memory usage and performance with and without, and take some time to understand the interaction with the blocksize.
- `VERSIONS` — set to 1 unless you know why you need more. You must always specify, because the default is 3.
- `COMPRESSION` — set to "snappy" until you can test performance with/without compression
- `TTL` — `-1`, unless you need expiration
- `BLOCKCACHE` — `true` (the default)
- `IN_MEMORY` — `false` (the default)
- `BLOCKSIZE` — `65536` (the default)

definition of a table for incrementers (from)

```
{NAME => 'timelesstest', DEFERRED_LOG_FLUSH => 'true', FAMILIES => [{NAME => 'family', BLOOMFILTER
```

DRAFT

DRAFT — ignore below

DRAFT

Vertical Partitioning (Column Families)

Suppose that after releasing the autocomplete API, we find that a sizeable minority of developers want to consume pre-baked HTML rather than the existing (and still-popular) JSON response. No request returns both HTML and JSON, Instead, we'll store each response type in its own *column family* in the autocomplete table. The pattern of access and data size are similar for each, but It might even be reasonable to put them in different tables.

Feature Set review

- **TTL
- Atomic counters: accumulate a numeric value, guaranteed consistent even if multiple clients simultaneously update it

- TTL ("Time to Live"): an optional amount of time, after which values are expired.
- Versioning by timestamp
- Column Families
- read caching
- Bloom filters fast rejection of missing rows
- Block-level compression

The "Snappy" algorithm gives a great balance of compression factor vs speed, and is easy to install.

- query filters: impose server load,
- and a kind of stored procedures/stored triggers called coprocessors). Here's a partial list of things you do *not* get:

From Hbase Def Guide:

Optimal loading of row keys: When performing a table scan where only the row keys are needed (no families, qualifiers, values, or timestamps), add a FilterList with a MUST_PASS_ALL operator to the scanner using setFilter(). The filter list should include both a First KeyOnlyFilter and a KeyOnlyFilter instance, as explained in Dedicated Filters on page 147. Using this filter combination will cause the region server to only load the row key of the first KeyValue (i.e., from the first column) found and return it to the client, resulting in minimized network traffic.

"Design for Reads"

HBase stores data in cells, scoped like this:

- Table — a hard partition of data. Tables are stored, partitioned and optimized in isolation.
- Row Key — the primary key for a record. Row contents are stored together, sorted by row key.
- Column Key — indexed elements of a row, in the form `column_family:column_qualifier` (the qualifier is optional).
  - Column Family — coarse-grained sub-partition of a row. You must declare the column family in advance. There are several options (like number of versions) you can set independently per column family.
  - Column Qualifier — the arbitrary remainder of a column key;
- Value — the contents you'd like to store, anything or nothing.

Table names and column familty names must be defined in advance, and their names may only contain printable characters (I recommend only using `[a-z_][a-z0-9_]*`). Everything else is bytes in / bytes out, exactly as issued.

- Avoid having more than a handful of column families on any high-performance table, especially if their patterns of write access are distinct.
- Avoid having more than a few million columns per row.
- Column families
  - — always specify the `versions`: by default it's 3, and you almost always want 1 or a value you've thought very carefully about
  - — Don't use more than two or three column families for a high-impact table; all of them have to keep pace with the most-heavily-used one.
- Use short row and column names. *Every* cell is stored with its row, column, timestamp and value, every time. (trust the HBase folks: this is the Right Thing).
  - — even still, fat row names (larger than their contents) often make sense. If so, increase the block size so that table indexes don't eat all your RAM.
- Keys should be space-efficient. Use *very* short names for column families (*u*, not *url*). Don't be profligate with size of column keys and row keys on huge tables: a binary-packed SHA digest of a URL is more efficient than its hex-encoded representation, which is likely more efficient than the URL itself. However, if that bare URL will let you efficiently index on sub-paths, use a bare URL. For another example, we gladly waste 6 bits of every byte in a quadkey, because it lets us do multi-scale queries.
- Keys should be properly encoded and sanitized
  - — HBase stores and returns arbitrary binary data, unmolested.
- All sorting is *lexicographic*: beware the "derp sort". Given row keys 1, 2, 7, 12, and 119, HBase stores them in the order 1, 119, 12, 2, 7: it sorts by the most significant (leftmost) byte first.
  - — zero-pad decimal numbers, and null-pad binary packet numbers. Suppose a certain key ranged from 0 to 60,000; you would zero-pad the number 69 as `00069` (5 bytes); the null-padded version would have bytes `00 45` (2 bytes).
  - — annoyingly, `+` sorts less than `-`, so `+45` precedes `-45`. However, `
  - — reverse timestamp
- Timestamps let HBase skip HStores
- Always set timestamps on fundamental objects. Server log lines, tweets, blog posts, and airline flight departures all have an intrinsic timestamp of occurrence, and they are all "fundamental" objects, not assertions derived from something else. In such

cases, always set a timestamp. In contrast, the "May 2012 Archive" page of a blog, containing many posts, is not fundamental; neither is an hourly cached count of server errors. These are *observations*, correct at the time they're made — so that observation time, not the intrinsic timestamp

- make sure you set the VERSIONS when you create the table+column family

Composite Keys. NOTE notation — HBase makes heavy use of composite keys (several values combined into a single string). We'll describe them using * quote marks (`"literal"`) to mean "that literal string" * braces `{field}` mean "substitute value of that field, removing the braces" * and separators, commonly `:`, `|` or `-`, to mean "that character, and make damn sure it's not used anywhere in the field value".

HBase is a database for storing "billions of rows and millions of columns"

References

- 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 17

Semi-Structured Data

Wikipedia Metadata

Wikipedia Pageview Stats (importing TSV)

This dataset is as easy as it gets. Well, that is, until you hit the brick wall of having to work around strings with broken encodings.

It is a `.tsv` file with columns for `lang_and_project`, `page_id`, `request_count`, `transferred_bytesize`. Since it's a tsv, parsing is as easy as defining the model and calling `from_tuple`:

```
class Wikipedia::RawPageview < Wikipedia::Base
  field :lang_and_project,     String
  field :id,                   String
  field :request_count,        Integer
  field :transferred_bytesize, Integer
end
mapper do
  input > from_tsv >
    ->(vals   ){ Wikipedia::RawPageview.from_tuple(vals) } >
    to_json > output
end
```

We're going to make the following changes:

- split the `lang_and_project` attribute into `in_language` and `wp_project`. They're different properties, and there's no good reason to leave them combined.
- add the numeric id of the article
- add the numeric id of the redirect-resolved article: this will make it easy to group page views under their topic

Assembling the namespace join table

- Take the pages metadata table,
  — get just the distinct pairs
  — verify
- 101 is “Book" — at least in English it is; in XX wikipedia it’s XX, and in XX it’s XX
  — pull in header from top of XML file
  — FIXME: is there a simpler script

Getting file metadata in a Wukong (or any Hadoop streaming) Script

TODO:

Translation

The translation is light, and the original model is of little interest, so we can just put the translation code into the raw model. First, we’ll add the `in_language` and `wp_project` properties as virtual accessors:

```
class Wikipedia::RawPageview < Wikipedia::Base
  # ... (cont) ...

  def in_language() lang_and_project.split('.')[0]       ; end
  def wp_project() lang_and_project.split('.')[1] || 'a' ; end

  def to_wikipedia_pageview
    Wikipedia::WikipediaPageview.receive(
      id: id, request_count: request_count, transferred_bytesize: transferred_bytesize,
      in_language: in_language, wp_project: wp_project)
  end
end

mapper do
  input > from_tsv >
    ->(vals   ){ Wikipedia::RawPageview.from_tuple(vals) } >
    ->(raw_rec){ raw_rec.to_wikipedia_pageview }           >
    to_json > output
end
```

Wikipedia Article Metadata (importing a SQL Dump)

Necessary Bullcrap #76: Bad encoding

Encoding errors (TODO: `LC_ALL`).

Scrub illegal urf-8 contenf from contents.

Wikipedia Page Graph

The wikipedia raw dumps have a pagelinks

```
INSERT INTO `pagelinks` VALUES (11049,0,'Rugby_union'),(11049,0,'Russia'),(11049,0,'Scottish_Footb
```

The Wikipedia datasets a bug that is unfortunately common and appallingly difficult to remediate: the

Perhaps, people from the future, Wikipedia will have a `cut: stdin: Illegal byte sequence -e:1:in <main>: invalid byte sequence in UTF-8 (ArgumentError)`

```
[source, ruby]
class Wikipedia::Pagelink < Wikipedia::Base
  field     :name, String
end
```

- SQL parser to tsv
  - — no need to assemble source domain model (yet) so don't
  - — emits `from_page_id into_namespace into_title`
- pig script to tack on page name for the from, page id for the into
  - — emits `from_page_id into_page_id from_namespace from_title into_name space into_title`

Target Domain Models

First step is to give some thought to the target domain model. There's a clear match to the Schema.org `Article` type, itself a subclass of `CreativeWork`, so we'll use the property names and descriptions:

```
class Wikipedia::WpArticle
  field      :id,                    String,    doc: "Unique identifier for the article; it forms
  field      :wp_page_id,            Integer,   doc: "Numeric serial ID for the page (as opposed
  field      :name,                  String,    doc: "Topic name (human-readable)"
  field      :description,           String,    doc: "Short abstract of the content"
  field      :keywords,    Array, of: String,    doc: "List of freeform tags for the topic"
  field      :article_body,          String     doc: "Contents of the article"
  field      :coordinates, Geo::Coordinates,     doc: "Primary location for the topic"
  field      :content_location, Geo::Place,      doc: "The location of the topic"
  field      :in_language,           String,    doc: "Language identifier"
  collection :same_as_ids,           String,    doc: "Articles that redirect to this one"
  collection :wp_links,              Hyperlink, doc: "Links to Wikipedia Articles from this artic
  collection :external_links,        Hyperlink, doc: "Links to external sites from this article"
  field      :wp_project,            String,    doc: "Wikipedia project identifier; main wikipedi
  field      :extended_properties,   Hash,      doc: "Interesting properties for this topic, extr
end
```

XML Data (Wikipedia Corpus)

The Wikipedia corpus is a good introduction to handling XML input. It lacks some of the really hairy aspects of XML handling foonote:[FIXME: make into references see the section on XML for a list of drawbacks], which will let us concentrate on the basics.

The raw data is a single XML file, 8 GB compressed and about 40 GB uncompressed. After a brief irrelevant header, it's simply several million records that look like this:

```
    <page>
      <title>Abraham Lincoln</title>
      <id>307</id>
      <revision>
        <id>454480143</id>
        <timestamp>2011-10-08T01:36:34Z</timestamp>
        <contributor>
          <username>AnomieBOT</username>
          <id>7611264</id>
        </contributor>
        <minor />
        <comment>Dating maintenance tags: Page needed</comment>
        <text xml:space="preserve">...(contents omitted; they
are XML-encoded (that&apos;s helpful)
  with spacing      preserved
including newlines)...</text>
      </revision>
    </page>
```

I've omitted the article contents, which are cleanly XML encoded and not in a CDATA block, so there's no risk of a spurious XML tag in an inappropriate place; avoid CDATA blocks if you can. The contents preserve all the whitespace of the original body, so we'll need to ensure that our XML parser does so as well.

From this, we'd like to extract the title, numeric page id, timestamp, and text, and re-emit each record in one of our favorite formats.

Now we meet our first two XML-induced complexities: *splitting* the file among mappers, so that you don't send the first half of an article to one task and the rest to a different one; and *recordizing* the file from a stream of lines into discrete XML fragments, each describing one article.

FIXME: we used `crack` not plain text the whole way

The law of small numbers

The law of small numbers: given millions of things, your one-in-a-million occurrences become commonplace.

Out of 12,389,353 records, almost all look like `<text xml:space="preserve">...stuff</text>` — but 446 records have an empty body, `<text xml:space="preserve" />`. Needless to say, we found this out not while developing locally, but rather some hundreds of thousands of records in while running on the cluster.

This crashing is a *good feature* of our script: it wasn't clear that an empty article body is permissible.

Custom Splitter / InputFormat

At 40GB uncompressed, the articles file will occupy about 320 HDFS blocks (assuming 128MB blocks), each destined for its own mapper. However, the division points among blocks is arbitrary: it might occur in the middle of a word in the middle of a record with no regard for your feelings about the matter. However, if you do it the courtesy of pointing to the first point within a block that a split *should* have occurred, Hadoop will handle the details of patching it onto the trailing end of the preceding block. Pretty cool.

You need to ensure that Hadoop splits the file at a record boundary: after `</page>`, before the next `<page>` tag.

If you're

Writing an input format and splitter is only as hard as your input format makes it, but it's the kind of pesky detail that lies right at the "do it right" vs "do it (stupid/simpl)ly" decision point. Luckily there's a third option, which is to steal somebody else's code [1]. Oliver Grisel (@ogrisel) has written an Wikipedia XML reader as a raw Java API reader in the Mahout project, and as a Pig loader in his pignlproc project. Mahout's XmlInputFormat (src)

Brute Force

If all you need to do is yank the data out of it's ill-starred format, or if the data format's complexity demands the agility of a high-level language, you can use Hadoop Streaming as a brute-force solution. In this case, we'll still be reading the data as a stream of lines, and use native libraries to do the XML parsing. We only need to ensure that the splits are correct, and the `StreamXmlRecordReader` (doc / source); that ships with Hadoop is sufficient.

```
class Wikipedia::RawArticle
  field :title,     Integer
  field :id,        Integer
  field :revision,  Wikipedia::RawArticleRevision
end
class Wikipedia::RawArticleRevision
```

1. see Hadoop the Definitive Guide, chapter FIXME: XX for details of building your own splitter

```
    field :id,        Integer
    field :timestamp, Time
    field :text,      String
  end
```

To explore data, you need data, in a form you can use, and that means engaging in the necessary evils of data munging: code to turn the data you have into the data you'd like to use. Whether it's public or commercial data data_commons, data from a legacy database, or managing unruly human-entered data, you will need to effectively transform the data into a format that can be used and understood.

Your goal is to take data from the *source domain* (the shape, structure and cognitive model of the raw data) to the *target domain* (the shape, structure and cognitive model you will work with). If you separate

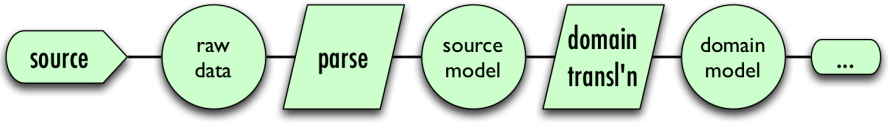
- Figure out the data model you'd like to target and build a lightweight class to represent it.
- Extract: the *syntactic* transformation from raw blobs to passive structured records
- Transform: the *semantic* transformation of structured data in the source domain to active models in the target domain
- Normalize
  - — clean up errors and missing fields
  - — augment with data from other tables
- Canonicalization:
  - — choose exemplars and mountweazels
  - — pull summary statistics
  - — sort by the most likely join key
  - — pull out your best guess as to a subuniverse (later)
- Land
  - — the data in its long-term home(s)

(later):

- fix your idea of subuniverse

DL TODO: (fix style (document pig style); align terminology with above `wikipedia/pageviews_extract_a.rb`

Extract, Translate, Canonicalize



- Raw data, as it comes from the source. This could be unstructured (lines of text, JOEMAN:HS
- Source domain models:
- Target domain models:
- Transformer:

You'll be tempted to move code too far to the right — to put transformation code into JOEMAN:HS Resist the urge. At the beginning of the project you're thinking about the details of extracting the data, and possibly still puzzling out the shape of the target domain models. Make the source domain models match the raw data as closely as reasonable, doing only the minimum effective work to make the data active.

Separate the model's *properties* (fundamental intrinsic data), its *metadata* (details of processing), and its *derived variables* (derived values).

In an enterprise context, this process is "ETL" — extraction, transformation and loading. In data at scale, rather than centralizing access in a single data store, you'll more often syndicate data along a documented provenance chain. So we'll change this to "Extract, Transform and Land".

Solitary, poor, nasty, brutish, and short

Thomas Hobbes wrote that the natural life of man is "solitary, poor, nasty, brutish, and short". and so should your data munging scripts be:

- Solitary: write discrete, single concern scripts. It's much easier to validate the data when your transformations are simple to reason about.

- Poor: spend wisely. It's especially important to optimize for programmer time, not cluster efficiency: you will probably spend more in opportunity cost to write the code than you ever will running it. Even if it will be a production ETL [2] job, get it running well and then make it run quickly. The hard parts are not where you expect them to be.
- Nasty: as you'll see, the real work in data munging are the special cases and domain incompatibilities. Ugly transformations will lead to necessarily ugly code, but even still there are ways to handle it. There's also a certain type of brittleness that is **good** — a script that quietly introduces corrupt data JOEMAN:HS
- Brutish: be effective, not elegant. Fight the ever-strong temptation to automate the in-automatable. Here are some street-fighting techniques we'll use below:
  - — Hand-curating a 200-row join table is the kind of thing you learn computer programing to avoid, but it's often the smart move.
  - — Faced with a few bad records that don't fit any particular pattern, you can write a script to recognize and repair them; or, if it's of modern size, just hand-fix them and save a diff. You've documented the change concisely and given future-you a chance to just re-apply the patch.
  - — To be fully-compliant, most XML libraries introduce a necessary complexity that complexifies **up** the work to deal with simple XML structures. It's often more sensible to handle simple as plain text.
- Short: use a high-level language. You'll invest much more in using the data than in writing the munging code. Two weeks later, when you discover that 2% of the documents used an alternate text encoding (or whatever card Loki had dealt) you'll be glad for brief readable scripts and a rich utility ecosystem.

FIXME: the analogy will line up better with the source if I can make the point that *your data munging scripts are to civilize data from the state of nature*. FIXME: the stuff above I like; need to chop out some of the over-purplish stuff from the rest of the chapter so the technical parts don't feel too ranty.

Canonical Data

- **sort** the data along its most-likely join field (sometimes but not always its primary key). This often enables a (merge_join TODO: ref), with tremendous speedup.
- **choose exemplars or add mountweazels**. Choose a few familiar records, and put their full contents close at hand to use for testing and provenance. You may also wish to add a mountweazel, a purposefully-bogus record. Why? First, so you have

2. ETL = Extraction, Transformation and Loading — what you call data munging when it runs often and well enough to deserve an enterprise-y name

something that fully exercises the data pipeline — all the fields are present, text holds non-keyboard and escape characters, and so forth. Second, for production smoke testing: it's a record you can send through the full data pipeline, including writing into the database, without concern that you will clobber an actual value. Lastly, since exemplars are real records, they may change over time; you can hold your mountweazel's fields constant. Make sure it's unmissably bogus and unlikely to collide: "John Doe" is a *terrible* name for a mountweazel, if there's any way user-contributed data could find its way into your database. The best-chosen bogus names appeal to the worst parts of a 5th-grader's sense of humor.

- **sample** a coherent subuniverse. Ensure this includes, or go back and add, your exemplar records.

Domain Models

Here are the models for Airport, Airline and Flight:

{{ d['code/munging/airline_flights/airport.rb|idio']['airport_model'] }}
Here's a snippet of the raw data:

```
{{ d['data/airline_flights/dataexpo_airports-raw-sample.csv|snippet'] }}
```

And here's what it looks like, transformed from raw to target model:

```
{{ d['tmp/airline_flights/dataexpo_airports-parsed-sample.tsv|snippet|wulign'] }}
{{ d['tmp/airline_flights/openflights_airports-parsed-sample.tsv|snippet|wulign'] }}
{{ d['tmp/airline_flights/airport_identifiers-sample.tsv|snippet|wulign'] }}
```

Airplane Models.

Airline Models.

Flight Models.

Data Extraction

Most of the extraction is straightforward. For reasons explained below, we have two sources of airline and airplane data, plus a gazette recon

Recovering Time Zone

So far, the airline data is fairly straightforward to import. However, Loki the Trickster rarely stays clear when it comes to adapting datasets across domains. The flight data has *local* actual/scheduled times, and it has airports, and it has the date — but it has neither the *absolute* time nor the time zone.

So, you need a map from airports to time zones. Good news: Openflights.org has that data. In fact, it's more comprehensive and adds some other interesting columns. Bad news: its data is somewhat messier and its identifiers don't cleanly reconcile against the `airline_flights` table. So, you need a *gazette*: a unified table listing the IATA, ICAO and FAA id of each airport. Wikipedia has a table indexing airports by IATA (and some, but not all, pairings with ICAO and FAA); and a table indexing airports by ICAO (and some, but not all, pairings with IATA and FAA) — and they mostly agree, but with a couple hundred (about 2% of nearly 10,000 airports) in conflict.

If you're keeping track: for want of a time zone, we need an airport-TZ map; for want of common identifiers, we need an ICAO-IATA-FAA identifier gazette; for want of clean resolution anywhere we end up reconciling two datasets against two different gazettes and hand-curating the *identifiable* errors in the mapping [3]. I won't go into the boring details of reconciling the airports: they're boring, and detailed in the code (see `munging/airline_flights/reconcile*.rb`).

But it's important to share that there is no royal road to clean data. It's easy to account for the work required to correct the obvious, surface messiness in the data. The *common case* is that correcting the 2% of outliers, reconciling conflicting assertions, dealing with ill-formatted records or broken encodings, and the rest of the "chimpanzee" work takes more time than anything else involved in semi-structured data extraction. Most importantly, that work is not a programming problem — it requires you to understand obscure details from the source domain not otherwise needed to solve your problem at hand

4

note:[Note that I both converted the altitude figure to meters and rounded it to one decimal place.]

3. "Yak Shaving": a recursively unbound descent into the sunk-cost fallacy

4. I now know far more about the pecadilloes of international airport identifier schemes than I ever wished to know. Airports may have an IATA id, an ICAO id, and (in the US and its territories) an FAA id. In the *continental* US, the ICAO is always the FAA id preceded by a "K": Austin-Bergstrom airport has FAA id `AUS`, IATA id `AUS` and ICAO id `KAUS`. However: * Not all airports have ICAO ids, and not all airports have IATA ids. * sThe FAA id often, but not always, matches the IATA id; this is the primary source of errors in the airport metadata, as people blithely assign the FAA id to an IATA-id-less airport. There is yet another identifier, the METAR id, used to identify the weather station at an airport; it was once the same as the ICAO id but they are now maintained independently. Yet, somehow, all those planes typically land at the right place.

Foundational Data

Exemplars & Mountweazels

For exemplar airports, I'll chose AUS (Austin), SFO (San Francisco), and BWI (Baltimore-Washington) because I'm most familiar with them; YYZ (Toronto), HNL (Honolulu), ANC (Anchorage) and SJU (San Juan, Puerto Rico) for geographic spread, and PHX (Phoenix) because Arizona (like the preceding three) has an odd time zones. That set is reasonably diverse, so there's no need for a mountweazel.

Helpful Sort

Standard Sample

I also selected a core set of 50 airports that cover most of the top US metropolitan areas (and includes the exemplars above)

sample, so we could construct the following reduced datasets:

- `airports-sampled.tsv`
- `flights-sampled.tsv` — flights between core airports

Daily Weather

Foundation data

exemplars and mountweazels:

final sort: There are two plausible choices. The first guess would be weather station ID, and our first encounters with the

Pitfalls

It's easy to

- Mangled unicode
- Timezones
- Incomplete joins *

When you have billions of something, one-in-a-million events stop being uncommon.

Drifting Identifiers

Another case where too much truth can be incredibly distruptive.

In mechanical engineering, you learn there are cases where *adding* a beam can weaken the part:

> \[the beam\] becomes a redundant support that limits angular deflection. But it may well add stiffness far out of proportion to its strength. Cracks may appear in or near the welded joint, thereby elimitating the added stiffness. Furthermor, the cracks so formed may propogate through the main angle iron. If so, the addition of the triangular "reinforcement" would actually *weaken* the part. Sometimes failures in a complicated structural member (as a casting) can be corrected by *removing* stiff but weak portions.

REFERENCE: Fundamentals of machine component design, Juvinall & Marshek — p 58

Too many identifiers provide just such a *redundant support*. When they line up with everything, they're useless. When they don't agree, the system becomes overconstrained.

My favorite example: user records in the Twitter API have **four** effective primary keys, none strictly reconcilable:

- a serial integer primary-key `id` — fairly straightforward [5].
- a legible `screen_name` (eg `@mrflip`); used in `@replies`, in primary user interaction, and in the primary URL.
  - — The mapping from `screen_name` key to the numeric `user_id` key is **almost** always stable, but at some few parts per million names and ids have been reassigned.
  - — Early on, the twitter API allowed screen names with arbitrary text; even now there are active users with non-ascii characters (©), spaces, and special characters (like `&`,`*`) in their screen_name.
  - — Even worse, it also allowed purely numeric screen names. For much of its history, the twitter API allowed the numeric id **or** screen name in the URL slug. If there were a user with screen name `1234` then the URL `http://twitter.com/users/show/1234.xml` would be ambiguous.
- The web-facing URL, a string like `http://twitter.com/mrflip` in the early days, now annoyingly redirected to `http://twitter.com/!#/mrflip`

5. as opposed to the Tweet ID, which had to undergo a managed transition from 32-bit to 64-bit before the 2 billion'th tweet could occur. They presumably look forward to doing the same for user ids at some point

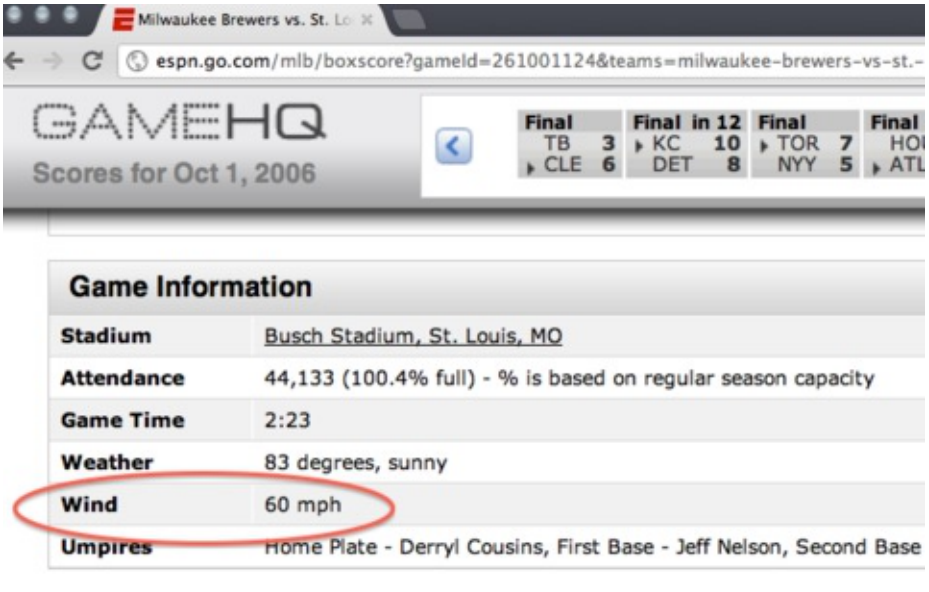
- Since its technology came through a corporate acquisition, Twitter's search API uses a completely incompatible numeric ID. For years, the search API never referenced the primary numeric ID (only its ID and screen name), while rest of the API never referenced the search ID.

When identifiers conflict, several bad things happen. Once you catch it, you have to do something to reconcile the incompatible assertions — too often, that something is hand-inspection of the bad records. (Your author spent a boring Sunday with the raw airport table, correcting cases where multiple airports were listed for the same identifier).

If the error is infrequent enough, you might not notice. A join which retains only one record can give different results for different joins, depending on which conflicting record is dropped. If the join isn't explicitly unique, you'll have a creeping introduction of extra records. Suppose some of the 38 towns named Springfield in the US listed the same airport code. Join the flight graph with the airport metadata, and each flight will be emitted that multiplicity of times. There are now extra records and subtly-inconsistent assertions (the wrong time zone could produce a flight that travelled back in time). Even worse, you've now coupled two different parts of the graph. This is yet another reason to sanity-check your record counts.

Authority and Consensus

Most dangerously, inconsistent assertions can spread. Wikipedia is incredibly valuable for disambiguating truth; but it can also be a vector for circular false authority: suppose Last.fm, Wikipedia and AllMusicGuide make the same claim about



The Baseball Reference website reports that on October 1, 2006, 44,133 fans watched the Milwaukee Brewers beat the St Louis Cardinals, 5-3, on a day that was "83° F (28 C), Wind 60mph (97 km/h) out to left field, Sunny." How bucolic! A warm sunny day, 100% attendance, two hours of good sport — and storm-force winds. Winds of that speed cause 30 foot (10 m) waves at sea; trees are broken off or uprooted, saplings bent and deformed, and structural damage to buildings becomes likely. Visit ESPN.com and MLB's official website and you'll see the same thing . They all use data from retrosheet.org, who have meticulously compiled the box score of every documented baseball game back to 1871. You won't win many bar bets by contesting the joint authority of baseball's notably meticulous encyclopedists, ESPN, and Major-League Baseball itself on the subject of a baseball game, but in this case it's worth probing further.

What do you do? One option is to eliminate such outliers: a 60mph wind is clearly erroneous. Do you discard the full record, or just that cell? In some of the outlier cases, the air temperature and wind speed are identical, suggesting a data entry error; in such cases it's probably safe to discard that cell and not the record. Where do you stop, though? If a game in Boston in October lists wind speeds of 24mph and a temperature of 24F, may you assume that end-of-season scheduling hassles caused them to play through bitter cold *and* bitter wind?

Your best bet is to bring independent authority to bear. In this case, though MLB and its encyclopedists are entrusted with the truth about baseball games, the NOAA is entrusted with the truth about the weater; we can consult the weather data to find the actual wind conditions on that day. (TODO: consult the weather data to find the actual wind conditions on that day) [6]

What's nice about this is that it's not the brittle kind of redundant support introduced by conflicting identifiers.

The abnormality of Normal in Humanity-scale data

Almost all of our tools for dealing with numerical error and uncertainty are oriented around normally-distributed data. Humanity-scale data drives artifacts such as sampling error to the edge (in fact we are not even sampling). Artifacts like a 60 mph wind do *not* follow a normal (see also the section in statistics).

In the section on statistics, we'll take advantage of it: Correlations tighten up — a 40F game temperature is an outlier in Atalanta in July, but not in Boston in September. There's

6. reconciling this inconsistency spurred an extended yak-shaving expedition to combine the weather data with the baseball data. Discovering there was nowhere to share the cleaned-up data led me to start Infochimps.

a natural distribution for wind speeds in general, and the subset found to accompany baseball games. Since we know these in their effective entirety, those distributions become very crisp. This helps you set reasonable heuristics for the "Where do you stop?" question --

REFERENCE: *http://www.srh.noaa.gov/images/bmx/outreach/EMA_wx_school/2009/DamageReporting.pdf* REFERENCE: *http://en.wikipedia.org/wiki/Beaufort_scale http://vizsage.com/blog/2007/10/retrosheet-eventfile-inconsistencies-ii.html*

The Abnormal

Given a billion of anything, one-in-a-million errors become commonplace. Given a billion records and a normal distribution of values, you can expect multiple 6-sigma outliers and hundreds of 5-sigma outliers.

A nice rule of thumb — and an insidious fallacy, the kind that makes airplanes crash and economies crumble. You may have a billion records **modelable** by a normal distribution, but you don't have a normal distribution of values.

First of all, even as a statistical process, a successful model for the bulk is suspect at the extreme tails. Furthermore, it neglects the "Black Swan" principle, that something unforeseen and unpredictable based on prior data could show up.

Unusual records are unusual

The central limit theorem lets you reason in the bulk about data of sufficient scale. It doesn't mean that your data **follows** the normal distribution, especially not at the tails.

ddd

Baseball scores from 1973-current are *modelable* by a normal distribution. You can use a normal distriubtion to reason forward, from a sample of baseball scores, to the properties of every baseball score. And you can use the normal distribution in combination with a statistical model to make predictions about future scores. But the scores do not follow the normal distribution: they follow the baseball-score distribution. We have an exact measurement of every score of every official game since 1973. There is no uncertainty and there are no outliers. (NOTE: I think this paragraph's point is interesting, but maybe it is not. Advice from readers welcome.)

Correlated risk, Model risk, and other Black Swans

```
"If you store 10,000 objects with us, on average we may lose one of them every 10 million years or
```

Amazon's engineering prowess is phenomenal, and this quote is meant to clarify, engineer-to-engineer, how much they have mitigated the risk from certain types of statistically-modelable error:

- disk failures
- cosmic rays causing bit flips
- an earthquake or power outage disabling two data separate centers

However, read broadly it's dangerous. 10 million years is long enough to let me joke about the Zombie Apocalypse or phase transitions in the fundamental physical constants. But even a 100-year timeline exposes plausible risks to your data from

- sabotage by disgruntled employees
- an act of war causing simultaneous destruction of their datacenters
- firmware failure simultaneously destroying every hard drive from one vendor
- a software update with an errant exclamation point causing the system to invalidate good blocks and preserve bad blocks
- Geomagnetic reversal of the earth's magnetic field causing unmitigated spike in cosmic-ray error rate

The above are examples of Black Swans. **Correlated risk**: Statistical models assume independent events (correlated samples) a model based on the observed default rate of consumer mortgages will fail if it neglects

Model risk: your predictions are plausible for the system you modeled — but the system you are modeling fundamentally changes.

For some time, it was easy for "black-hat" (adversarial) companies to create bogus links that would increase the standing of their website in Google search results. Google found a model that could successfully expose such cheaters, and in a major algorithm update began punishing linkbait-assisted sites. What happened? Black-hat companies began creating bogus links to their *competitors*, so they would be downranked instead. The model still successfully identified linkspam-assisted sites, but the system was no longer one in which a site that was linkspam-assisted meant a site that was cheating. The introduction of a successful model destabilized the system.

Even more interestingly, algorithms can **stably** modify their system. For some time, when the actress Anne Hathaway received positive reviews for her films, the stock price of the firm Berkshire Hathaway trended up — news-reading "algorithmic trading" robots correctly graded the sentiment but not its target. It's fair to call this a flaw in the model, because Anne Hathaway's pretty smile doesn't correspond to the financial health of a diversified insurance company. An "algorithmic trading robot" algorithmic trading robot can thus bet that Berkshire Hathaway results will regress to their former value if they spike following an Anne Hathaway film. Those adversarial trades **change the sys-**

tem, from one in which Berkshire Hathaway's stock price followed its financial health, to a system where Berkshire Hathaway's stock price followed its financial health, Anne Hathaway's acting career, and a coupling constant governed by the current state of the art in predictive analytics.

Coupling risk: you hedge your financial model with an insurance contract, but the insurance counterparty goes bankrupt,

Other Parsing Strategies

Stateful Parsing

s-Expression parsing

Ruby stdlib's StringScanner provides for lexical scanning operations on a String.

CHAPTER 18

Hadoop Execution in Detail

Launch

When you launch a job (with `pig`, `wukong run`, or `hadoop jar`), it starts a local process that

- prepares a synthesized configuration from config files of the program and the machine (`core-site.xml`, `hdfs-site.xml`, `mapred-site.xml`).
- asks the jobtracker for a job ID
- pushes your program and its assets (jars, script files, distributed cache contents) into the job's directory on the HDFS.
- asks the jobtracker enqueue the job.

After a few seconds you should see the job appear on the jobtracker interface. The jobtracker will begin dispatching the job to workers with free slots, as directed by its scheduler [1]. It knows where all the input blocks are, and will try to launch each task on the same machine as its input ("bring the compute to the data"). The jobtracker will tell you how many map tasks are "local" (launched on a different machine than its input); if it's not harmlessly small, see "Many non-local mappers" (page 102).

The launching process doesn't take many resources, so for a development cluster it's OK to launch a job from a worker machine. Terminating the launch process won't affect the job execution, but its output is useful. To record its output even if you log off, use the `nohup` command:

```
nohup [...normal launch command...] >> /tmp/my_job-`date +%F`.log 2>&1 &
```

1. unless your cluster is heavily used by multiple people, the default scheduler is fine. If fights start breaking out, quickly consult (TODOREF Hadoop Operations) for guidance on the other choices

Run `tail -f /tmp/my_job-*.log` to keep watching the job's progress.

The job draws its default configuration from the *launch* machine's config file. Make sure those defaults doesn't conflict with appropriate values for the workers that will actually execute the job! One great way to screw this up is to launch a job from your dev machine, go to dinner and come back to find it using one reducer and a tiny heap size. Another is to start your job from a master that is provisioned differently from the workers.

Split

Input files are split and assigned to mappers.

Each mapper will receive a chunk bounded by:

- The file size — normally, each mapper handles at most one file (and typically, one part of a very large file). (footnote: Pig will pre-combine small files into single map inputs with the `pig.splitCombination` commandline parameter.)
- Min split size — up to the size of each file, you can force hadoop to make each split larger than `mapred.min.split.size`
- Block size — the natural unit of data to feed each map task is the size of an HDFS file chunk; this is what lets Hadoop "bring the compute to the data".
- Input format — some input formats are non-splittable (by necessity, as for some compression formats; or by choice, when you want to enforce no file splits). [2]

Exercises:

- Create a 2GB file having a 128MB block size on the HDFS. Run `wu-stream cat cat --min_split_mb=1900` on it. How many map tasks will launch? What will the "non-local" cell on the jobtracker report? Try it out for 1900, and also for values of 128, 120, 130, 900 and 1100.

2. Paraphrasing the Hadoop FAQ, to make a *non-splittable* FileInputFormat, your particular input-format should return false for the isSplittable call. If you would like the whole file to be a single record, you must also implement a RecordReader interface to do so — the default is LineRecordReader, which splits the file into separate lines. The other, quick-fix option, is to set mapred.min.split.size to large enough value.

Mappers

Hadoop Streaming (Wukong, MrJob, etc)

If it's a Hadoop "streaming" job (Wukong, MrJob, etc), the child process is a Java jar that itself hosts your script file:

- it forks the script in a new process. The child ulimit applies to this script, but the heap size and other child process configs do not.
- passes all the Hadoop configs as environment variables, changing `.` dots to _ underbars. Some useful examples:
  - `map_input_file` — the file this task is processing
  - `map_input_start` — the offset within that file
  - `mapred_tip_id` — the task ID. This is a useful ingredient in a unique key, or if for some reason you want each mapper's output to go to a distinct reducer partition.
- directs its input to the script's `STDIN`. Not all input formats are streaming-friendly.
- anything the script sends to its `STDOUT` becomes the jar's output.

forks yet another

Once the maps start, it's normal for them to seemingly sit at 0% progress for a little while: they don't report back until a certain abount of data has passed through. Annoyingly, jobs with gzipp'ed input will remain mute until they are finished (and then go instantly from 0 to 100%).

exercise: Write a mapper that ignores its input but emits a configurable number of bytes, with a configurable number of bytes per line. Run it with one mapper and one reducer. Compare what happens when the output is just below, and just above, each of these thresholds: - the HDFS block size - the mapper sortbuf spill threshold - the mapper sortbuf data threshold - the mapper sortbuf total threshold

Speculative Execution ======

For exploratory work, it's worth

Choosing a file size

Jobs with Map and Reduce

For jobs that have a reducer, the total size of the output dataset divided by the number of reducers implies the size of your output files [3]. Of course your working dataset is less than a few hundred MB this doesn't matter.

If your working set is large enough to care and less than about 10 TB, size your reduce set for files of about 1 to 2 GB.

- *Number of mappers*: by default, Hadoop will launch one mapper per HDFS block; it won't assign more than one file to each mapper [4]. More than a few thousand
- *Reducer efficiency*: as explained later (TODO: ref reducer_size), your reducers are most efficient at 0.5 to 2 GB.
- *HDFS block size*: >= 1-2 GB — a typically-seen hadoop block size is 128 MB; as you'll see later, there's a good case for even larger block sizes. You'd like each file to hold 4 or more blocks.
- *your network connection* (< 4GB): a mid-level US internet connection will download a 4 GB file segment in about 10 minutes, upload it in about 2 hours.
- *a DVD*: < 4 GB — A DVD holds about 4GB. I don't know if you use DVDs still, but it's a data point.
- *Cloud file stores*: < 5 GB — The Amazon S3 system now allows files greater than 5 GB, but it requires a special multi-part upload transfer.
- *Browsability*: a 1 GB file has about a million 1kB records.

Even if you don't find any of those compelling enough to hang your hat on, I'll just say that files of 2 GB are large enough to be efficient and small enough to be manageable; they also avoid those upper limits even with natural variance in reduce sizes.

If your dataset is

Mapper-only jobs

There's a tradeoff:

If you set your min-split-size larger than your block size, you'll get non-local map tasks, which puts a load on your network.

3. Large variance in counts of reduce keys not only drives up reducer run times, it causes variance in output sizes; but that's just insult added to injury. Worry about that before you worry about the target file size.
4. Pig has a special option to roll up small files

However, if you let it launch one job per block, you'll have two problems. First, one mapper per HDFS block can cause a large number of tasks: a 1 TB input dataset of 128 MB HDFS blocks requires 8,000 map tasks. Make sure your map task runtimes aren't swamped by job startup times and that your jobtracker heap size has been configured to handle that job count. Secondly, if your job is ever-so-slightly expansive — if it turns a 128 MB input block into a 130 MB output file — then you will double the block count of the dataset. It takes twice the actual size to store on disk and implies twice the count of mappers in subsequent stages.

My recommendation: (TODO: need to re-confirm with numbers; current readers please take with a grain of salt.)

To learn more, see the

Reduce Logs

TODO: need one that does reduce spills

```
2012-12-17 02:11:58,555 WARN org.apache.hadoop.conf.Configuration: /mnt/hadoop/mapred/local/taskTr
...
2012-12-17 02:11:58,580 WARN org.apache.hadoop.conf.Configuration: /mnt/hadoop/mapred/local/taskTr
```

These are harmless: this job has read the system .xml files into its jobconf (as it will if none are explicitly specified); some non-adjustable parameters came along for the ride

```
2012-12-17 02:11:58,776 INFO org.apache.hadoop.util.NativeCodeLoader: Loaded the native-hadoop lib
```

That's good news: the native libraries are much faster.

```
...
2012-12-17 02:12:00,300 WARN org.apache.hadoop.io.compress.snappy.LoadSnappy: Snappy native librar
2012-12-17 02:12:00,300 INFO org.apache.hadoop.io.compress.snappy.LoadSnappy: Snappy native librar
```

Also good news: compressing midstream data is almost always a win, and the Snappy codec is a good balance of CPU and bandwidth.

```
2012-12-17 02:12:00,389 INFO org.apache.hadoop.mapred.ReduceTask: ShuffleRamManager: MemoryLimit=2
```

This is crucial.

- The `MemoryLimit` figure should be `mapred.job.shuffle.input.buffer.percent * 2147483647` if you have more than 2GB of heap set aside for the reducer.
- The `MaxSingleShuffleLimit` should be 25% of that.

The memory used to store map outputs during shuffle is given by `(shuffle_heap_frac * [reduce_heap_mb, 2GB].min)` If you have more than 2GB of reducer heap size, consider increasing this value. It only applies during the shuffle, and so does not compete with your reducer for heap space.

default[:hadoop][:shuffle_heap_frac] = 0.70

The memory used to store map outputs during reduce is given by # (reduce_heap_frac * [reduce_heap_mb, 2GB].min) # These buffers compete with your reducer code for heap space; however, many # reducers simply stream data through and have no real memory burden once the # sort/group is complete. If that is the case, or if your reducer heap size is # well in excess of 2GB, consider adjusting this value. # Tradeoffs — Too high: crash on excess java heap. Too low: modest performance # hit on reduce default[:hadoop][:reduce_heap_frac] = 0.00

```
...
2012-12-17 02:12:00,758 INFO org.apache.hadoop.mapred.ReduceTask: attempt_201212070519_0013_r_0000
2012-12-17 02:12:00,758 INFO org.apache.hadoop.mapred.ReduceTask: attempt_201212070519_0013_r_0000
2012-12-17 02:12:00,767 INFO org.apache.hadoop.mapred.ReduceTask: attempt_201212070519_0013_r_0000
```

The reducer has fully started, and is ready to receive data from each mapper.

```
...
2012-12-17 02:12:05,759 INFO org.apache.hadoop.mapred.ReduceTask: attempt_201212070519_0013_r_0000
...
2012-12-17 02:13:03,350 INFO org.apache.hadoop.mapred.ReduceTask: attempt_201212070519_0013_r_0000
```

These roll in as each map task finishes

```
2012-12-17 02:15:53,686 INFO org.apache.hadoop.mapred.ReduceTask: Ignoring obsolete output of KILL
...
2012-12-17 02:16:20,713 INFO org.apache.hadoop.mapred.ReduceTask: Ignoring obsolete output of KILL
2012-12-17 02:16:22,696 INFO org.apache.hadoop.mapred.ReduceTask: attempt_201212070519_0013_r_0000
```

The `obsolete output of KILLED map-task` lines may look dire, but they're harmless. The jobtracker specified several map tasks for speculative execution (TODO: ref), and then killed the attempts that didn't finish first. The reducers are smart enough to ignore the output of failed an killed jobs, and only proceed to the reduce when exactly one copy of the data has arrived.

TODO: get a job that has multiple spills

```
2012-12-17 02:16:23,715 INFO org.apache.hadoop.mapred.ReduceTask: GetMapEventsThread exiting
2012-12-17 02:16:23,715 INFO org.apache.hadoop.mapred.ReduceTask: getMapsEventsThread joined.
```

The map tasks have all arrived, and so the final merge passes may begin.

```
2012-12-17 02:16:23,717 INFO org.apache.hadoop.mapred.ReduceTask: Closed ram manager
2012-12-17 02:16:23,742 INFO org.apache.hadoop.mapred.ReduceTask: Interleaved on-disk merge comple
2012-12-17 02:16:23,743 INFO org.apache.hadoop.mapred.ReduceTask: In-memory merge complete: 161 fi
2012-12-17 02:16:23,744 INFO org.apache.hadoop.mapred.ReduceTask: Merging 0 files, 0 bytes from di
2012-12-17 02:16:23,747 INFO org.apache.hadoop.mapred.ReduceTask: Merging 161 segments, 922060579
2012-12-17 02:16:23,750 INFO org.apache.hadoop.mapred.Merger: Merging 161 sorted segments
2012-12-17 02:16:23,751 INFO org.apache.hadoop.mapred.Merger: Down to the last merge-pass, with 14
2012-12-17 02:16:23,762 INFO org.apache.hadoop.streaming.PipeMapRed: PipeMapRed exec [/usr/bin/rub
2012-12-17 02:16:23,808 INFO org.apache.hadoop.streaming.PipeMapRed: R/W/S=1/0/0 in:NA [rec/s] out
2012-12-17 02:16:23,809 INFO org.apache.hadoop.streaming.PipeMapRed: R/W/S=10/0/0 in:NA [rec/s] ou
2012-12-17 02:16:23,814 INFO org.apache.hadoop.streaming.PipeMapRed: R/W/S=100/0/0 in:NA [rec/s] o
```

```
2012-12-17 02:16:23,834 INFO org.apache.hadoop.streaming.PipeMapRed: R/W/S=1000/0/0 in:NA [rec/s]
2012-12-17 02:16:24,213 INFO org.apache.hadoop.streaming.PipeMapRed: Records R/W=1746/1
2012-12-17 02:16:25,243 INFO org.apache.hadoop.streaming.PipeMapRed: R/W/S=10000/355/0 in:10000=10
2012-12-17 02:16:34,232 INFO org.apache.hadoop.streaming.PipeMapRed: Records R/W=82322/4975
2012-12-17 02:16:36,151 INFO org.apache.hadoop.streaming.PipeMapRed: R/W/S=100000/5472/0 in:8333=1
2012-12-17 02:16:44,241 INFO org.apache.hadoop.streaming.PipeMapRed: Records R/W=154675/6964
2012-12-17 02:33:40,532 INFO org.apache.hadoop.streaming.PipeMapRed: Records R/W=7456136/352941
2012-12-17 02:33:44,878 INFO org.apache.hadoop.streaming.PipeMapRed: R/W/S=7500000/354750/0 in:720
```

CHAPTER 19
Overview of Datasets

The examples in this book use the "Chimpmark" datasets: a set of freely-redistributable datasets, converted to simple standard formats, with traceable provenance and documented schema. They are the same datasets as used in the upcoming Chimpmark Challenge big-data benchmark. The datasets are:

- Wikipedia English-language Article Corpus (`wikipedia_corpus`; 38 GB, 619 million records, 4 billion tokens): the full text of every English-language wikipedia article.
- Wikipedia Pagelink Graph (`wikipedia_pagelinks`;): every page-to-page hyperlink in wikipedia.
- Wikipedia Pageview Stats (`wikipedia_pageviews`; 2.3 TB, about 250 billion records (FIXME: verify num records)): hour-by-hour pageviews for all of Wikipedia
- ASA SC/SG Data Expo Airline Flights (`airline_flights`; 12 GB, 120 million records): every US airline flight from 1987-2008, with information on arrival/depature times and delay causes, and accompanying data on airlines, airports and airplanes.
- NCDC Hourly Global Weather Measurements, 1929-2009 (`ncdc_weather_hourly`; 59 GB, XX billion records): hour-by-hour weather from the National Climate Data Center for the entire globe, with reasonably-dense spatial coverage back to the 1950s and in some case coverage back to 1929.
- 1998 World Cup access logs (`access_logs/ita_world_cup_apachelogs`; 123 GB, 1.3 billion records): every request made to the 1998 World Cup Web site between April 30, 1998 and July 26, 1998, in apache log format.
- 60,000 UFO Sightings
- Retrosheet Game Logs — every recorded baseball game back to the 1890s

- Gutenberg Corpus

Wikipedia Page Traffic Statistic V3

```
-
- a 150 GB sample of the data used to power trendingtopics.org. It includes a full 3 months of hou
```

- Twilio/Wigle.net Street Vector Data Set — — geo — Twilio/Wigle.net database of mapped US street names and address ranges.
- 2008 TIGER/Line Shapefiles — 125 GB — geo — This data set is a complete set of Census 2000 and Current shapefiles for American states, counties, subdivisions, districts, places, and areas. The data is available as shapefiles suitable for use in GIS, along with their associated metadata. The official source of this data is the US Census Bureau, Geography Division.

ASA SC/SG Data Expo Airline Flights

This data set is from the [ASA Statistical Computing / Statistical Graphics](*http://statcomputing.org/dataexpo/2009/the-data.html*) section 2009 contest, "Airline Flight Status — Airline On-Time Statistics and Delay Causes". The documentation below is largely adapted from that site.

The U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics (BTS) tracks the on-time performance of domestic flights operated by large air carriers. Summary information on the number of on-time, delayed, canceled and diverted flights appears in DOT's monthly Air Travel Consumer Report, published about 30 days after the month's end, as well as in summary tables posted on this website. BTS began collecting details on the causes of flight delays in June 2003. Summary statistics and raw data are made available to the public at the time the Air Travel Consumer Report is released.

The data consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. This is a large dataset: there are nearly 120 million records in total, and takes up 1.6 gigabytes of space compressed and 12 gigabytes when uncompressed.

The data comes originally from the DOT's [Research and Innovative Technology Administration (RITA)](*http://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp*) group, where it is [described in detail](*http://www.transtats.bts.gov/Fields.asp?Table_ID=236*). You can download the original data there. The files here have derivable variables removed, are packaged in yearly chunks and have been more heavily compressed than the originals.

Here are a few ideas to get you started exploring the data:

- When is the best time of day/day of week/time of year to fly to minimise delays?
- Do older planes suffer more delays?
- How does the number of people flying between different locations change over time?
- How well does weather predict plane delays?
- Can you detect cascading failures as delays in one airport create delays in others? Are there critical links in the system?

Support data

- **Openflights.org** (ODBL-licensed): user-generated datasets on the world of air flight.
  - `openflights_airports.tsv` (original) — info on about 7000 airports.
  - `openflights_airlines.tsv` (original) — info on about 6000 airline carriers
  - `openflights_routes.tsv` (original) — info on about 60_000 routes between 3000 airports on 531 airlines.
- **Dataexpo** (Public domain): The core airline flights database includes
  - `dataexpo_airports.tsv` (original) — info on about 3400 US airlines; slightly cleaner but less comprehensive than the Openflights.org data.
  - `dataexpo_airplanes.tsv` (original) — info on about 5030 US commercial airplanes by tail number.
  - `dataexpo_airlines.tsv` (original) — info on about 1500 US airline carriers; slightly cleaner but less comprehensive than the Openflights.org data.
- **Wikipedia.org** (CC-BY-SA license): Airport identifiers
  - `wikipedia_airports_iata.tsv` (original) — user-generated dataset pairing airports with their IATA (and often ICAO and FAA) identifiers.
  - `wikipedia_airports_icao.tsv`(original) — user-generated dataset pairing airports with their ICAO (and often IATA and FAA) identifiers.

The airport datasets contain errors and conflicts; we've done some hand-curation and verification to reconcile them. The file `wikipedia_conflicting.tsv` shows where my patience wore out.

ITA World Cup Apache Logs

- 1998 World Cup access logs (`access_logs/ita_world_cup_apachelogs`; 123 GB, 1.3 billion records): every request made to the 1998 World Cup Web site between April 30, 1998 and July 26, 1998, in apache log format.

Daily Global Weather Measurements, 1929-2009 (NCDC, GSOD)

- 20 GB
- geo, stats
- Old Weather project --
- *http://philip.brohan.org.transfer.s3.amazonaws.com/oW_imma_20120801.tgz*

Retrosheet

```
25          Retrosheet: MLB play-by-play, high detail, 1840-2011    ripd/www.retrosheet.org-2007/b
25          Retrosheet: MLB box scores, 1871-2011                   ripd/www.retrosheet.org-2007/b
```

Gutenberg corpus

The main collection is about 650GB (as of October 2011) with nearly 2 million files, 60 languages, and dozens of different file formats.

- Gutenberg collection catalog as RDF
- rsync, repeatable: Mirroring How-To
- wget, zip files: Gutenberg corpus download instructions. From a helpful Stack Overflow thread, you can run `wget -r -w 2 -m 'http://www.gutenberg.org/robot/harvest?filetypes[]=txt&langs[]=en'`

  — see also ZIP file input format

  ```
  cd /mnt/gutenberg/ ; mkdir -p logs/gutenberg
  nohup rsync -aviHS --max-size=10M --delete --delete-after --exclude=\*.{m4a,m4b,ogg,spx,tei,
  ```

- Complete works of William Shakespeare

CHAPTER 20
Cheatsheets

Terminal Commands

Table 20-1. Hadoop Filesystem Commands

action	command
list files	`hadoop fs -ls`
list files' disk usage	`hadoop fs -du`
total HDFS usage/available	visit namenode console
copy local → HDFS	
copy HDFS → local	
copy HDFS → remote HDFS	
make a directory	`hadoop fs -mkdir ${DIR}`
move/rename	`hadoop fs -mv ${FILE}`
dump file to console	`hadoop fs -cat ${FILE} | cut -c 10000 | head -n 10000`
remove a file	
remove a directory tree	
remove a file, skipping Trash	
empty the trash NOW	
health check of HDFS	
report block usage of files	
decommission nodes	
list running jobs	
kill a job	

action	command
kill a task attempt	
CPU usage by process	`htop`, or `top` if that's not installed
Disk activity	
Network activity	
	`grep -e '[regexp]'`
	`head`, `tail`
	`wc`
	`uniq -c`
	`sort -n -k2`

Table 20-2. UNIX commandline tricks

action	command	Flags
Sort data	`sort`	reverse the sort: `-r`; sort numerically: `-n`; sort on a field: `-t [delimiter] -k [index]`
Sort large amount of data	`sort --parallel=4 -S 500M`	use four cores and a 500 megabyte sort buffer
Cut delimited field	`cut -f 1,3-7 -d ','`	emit comma-separated fields one and three through seven
Cut range of characters	`cut -c 1,3-7`	emit characters one and three through seven
Split on spaces	`	ruby -ne *puts $_.split(/\\s+/).join("\t")*`
split on continuous runs of whitespace, re-emit as tab-separated	Distinct fields	`
sort	uniq`	only dupes: `-d`
Quickie histogram	`	sort
uniq -c`	TODO: check the rendering for backslash	Per-process usage
`htop`	Installed	Running system usage

For example: `cat * | cut -c 1-4 | sort | uniq -c` cuts the first 4-character

Not all commands available on all platforms; OSX users should use Homebrew, Windows users should use Cygwin.

Regular Expressions

Table 20-3. Regular Expression Cheatsheet

character	meaning

character	meaning
TODO	
`.`	any character
`\w`	any word character: a-z, A-Z, 0-9 or _ underscore. Use `[:word:]` to match extended alphanumeric characters (accented characters and so forth)
`\s`	any whitespace, whether space, tab (`\t`), newline (`\n`) or carriage return (`\r`).
`\d`	
`\x42` (or any number)	the character with that hexadecimal encoding.
`\b`	word boundary (zero-width)
`^`	start of line; use `\\A` for start of string (disregarding newlines). (zero-width)
`$`	end of line; use `\\z` for end of string (disregarding newlines). (zero-width)
`[^a-mA-M]`	match character in set
`[a-mA-M]`	reject characters in set
`a|b|c`	a or b or c
`(...)`	group
`(?:...)`	non-capturing group
`(?<varname>...)`	named group
`*`, `+`	zero or more, one or more. greedy (captures the longest possible match)
`*?`, `+?`	non-greedy zero-or-more, non-greedy one-or-more
`{n,m}`	repeats `n` or more, but `m` or fewer times

These Table 20-4 are for practical extraction, not validation — they may let nitpicks through that oughtn't (eg, a time zone of `-0000` is illegal by the spec, but will pass the date regexp given below). As always, modify them in your actual code to be as brittle (restrictive) as reasonable.

Table 20-4. Example Regular Expressions

intent	Regular Expression	Comment
Double-quoted string	`%r{"((?:\\.	[^\"])*)"}`
all backslash-escaped character, or non-quotes, up to first quote	Decimal number with sign	`%r{([\-\+\d]+\.\d+)}`
optional sign; digits-dot-digits	Floating-point number	`%r{([\+\-]?\d+\.\d+(?:[eE][\+\-]?\d+)?)}`
optional sign; digits-dot-digits; optional exponent	ISO date	`%r{b(\d\d\d\d)-(\d\d)-(\d\d)T(\d\d):(\d\d):(\d\d)([\+\-]\d\d:?\d\d
[\+\-]\d\d	Z)\b}`	groups give year, month, day, hour, minute, second and time zone respectively.

Ascii table:

```
"\x00"  \c
"\x01"  \c
"\x02"  \c
"\x03"  \c
"\x04"  \c
"\x05"  \c
"\x06"  \c
"\a"    \c
"\b"    \c
"\t"    \c              \s
"\n"    \c              \s
"\v"    \c              \s
"\f"    \c              \s
"\r"    \c              \s
"\x0E"  \c
"\x0F"  \c
"\x10"  \c
"\x11"  \c
"\x12"  \c
"\x13"  \c
"\x14"  \c
"\x15"  \c
"\x16"  \c
"\x17"  \c
"\x18"  \c
"\x19"  \c
"\x1A"  \c
"\e"    \c
"\x1C"  \c
"\x1D"  \c
"\x1E"  \c
"\x1F"  \c
" "                     \s
"!"
"\""
"#"
"$"
"%"
"&"
"'"
"("
")"
"*"
"+"
","
"-"
"."
"/"
"0"             \w
"1"             \w
"2"             \w
"3"             \w
```

```
"4"             \w
"5"             \w
"6"             \w
"7"             \w
"8"             \w
"9"             \w
":"
";"
"<"
"="
">"
"?"
"@"
"A"             \w
"B"             \w
"C"             \w
"D"             \w
"E"             \w
"F"             \w
"G"             \w
"H"             \w
"I"             \w
"J"             \w
"K"             \w
"L"             \w
"M"             \w
"N"             \w
"O"             \w
"P"             \w
"Q"             \w
"R"             \w
"S"             \w
"T"             \w
"U"             \w
"V"             \w
"W"             \w
"X"             \w
"Y"             \w
"Z"             \w
"["
"\\"
"]"
"^"
"_"             \w
"`"
"a"             \w
"b"             \w
"c"             \w
"d"             \w
"e"             \w
"f"             \w
"g"             \w
```

```
"h"             \w
"i"             \w
"j"             \w
"k"             \w
"l"             \w
"m"             \w
"n"             \w
"o"             \w
"p"             \w
"q"             \w
"r"             \w
"s"             \w
"t"             \w
"u"             \w
"v"             \w
"w"             \w
"x"             \w
"y"             \w
"z"             \w
"{"
"|"
"}"
"~"
"\x7F"  \c
"\x80"  \c
```

Pig Operators

Table 20-5. Pig Operator Cheatsheet

action	operator
	JOIN
	FILTER

CHAPTER 21

Book Metadata

Author

Philip (flip) Kromer is the founder and CTO at Infochimps.com, a big data platform that makes acquiring, storing and analyzing massive data streams transformatively easier. I enjoy 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

- Core committer for Wukong, the leading ruby library for Hadoop
- Core committer for Ironfan, a framework for provisioning complex distributed systems in the cloud or data center.
- Wrote the most widely-used cookbook for deploying hadoop clusters using Chef
- Contributed chapter to *The Definitive Guide to Hadoop* by Tom White

This book is a guide to data science in practice

- practical
- simple
- how to make hard problems simple
- real data, real problems
- developer friendly

terabytes not petabytes cloud not fixed exploratory no production

Hadoop is a remarkably powerful tool for processing data, giving us at long last mastery over massive-scale distributed computing. More than likely, that's how you came to be reading this sentence.

What you might not yet know is that Hadoop's power comes from *embracing*, not conquering, the constraints of distributed computing; and in doing so, exposes a core simplicity that makes programming it exceptionally fun.

Hadoop's bargain is thus: you must agree to write all your programs according to single certain form, which we'll call the "Map / Reduce Haiku":

```
Data flutters by
Elephants make sturdy piles
Number becomes thought
```

For any such program, Hadoop's diligent elephants will intelligently schedule the tasks across ones or dozens or thousands of machines; attend to logging, retry and error handling; distribute your data to the workers that process it; handle memory allocation, partitioning and network routing; and a myriad other details that would otherwise stand between you and insight.

Here's an example. (we'll skip for now many of the details, so that you can get a high-level sense of how simple and powerful Hadoop can be.)

Oct 23-25

PRE-RELEASE DESCRIPTION: Big Data for Chimps

Short description:

Working with big data for the first time? This unique guide shows you how to use simple, fun, and elegant tools working with Apache Hadoop. You'll learn how to break problems into efficient data transformations to meet most of your analysis needs. It's an approach that not only works well for programmers just beginning to tackle big data, but for anyone using Hadoop.

Long description:

This unique guide shows you how to use simple, fun, and elegant tools leveraging Apache Hadoop to answer big data questions. You'll learn how to break problems into efficient data transformations to meet most of your analysis needs. Its developer-friendly approach works well for anyone using Hadoop, and flattens the learning curve for those working with big data for the first time.

Written by Philip Kromer, founder and CTO at Infochimps, this book uses real data and real problems to illustrate patterns found across knowledge domains. It equips you with

a fundamental toolkit for performing statistical summaries, text mining, spatial and time-series analysis, and light machine learning. For those working in an elastic cloud environment, you'll learn superpowers that make exploratory analytics especially efficient.

- Learn from detailed example programs that apply Hadoop to interesting problems in context
- Gain advice and best practices for efficient software development
- Discover how to think at scale by understanding how data must flow through the cluster to effect transformations
- Identify the tuning knobs that matter, and rules-of-thumb to know when they're needed. Learn how and when to tune your cluster to the job and ///

/// e

- Humans are important, robots are cheap: you'll learn how to recognize which tuning knobs, and *