Big Data for Chimps

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

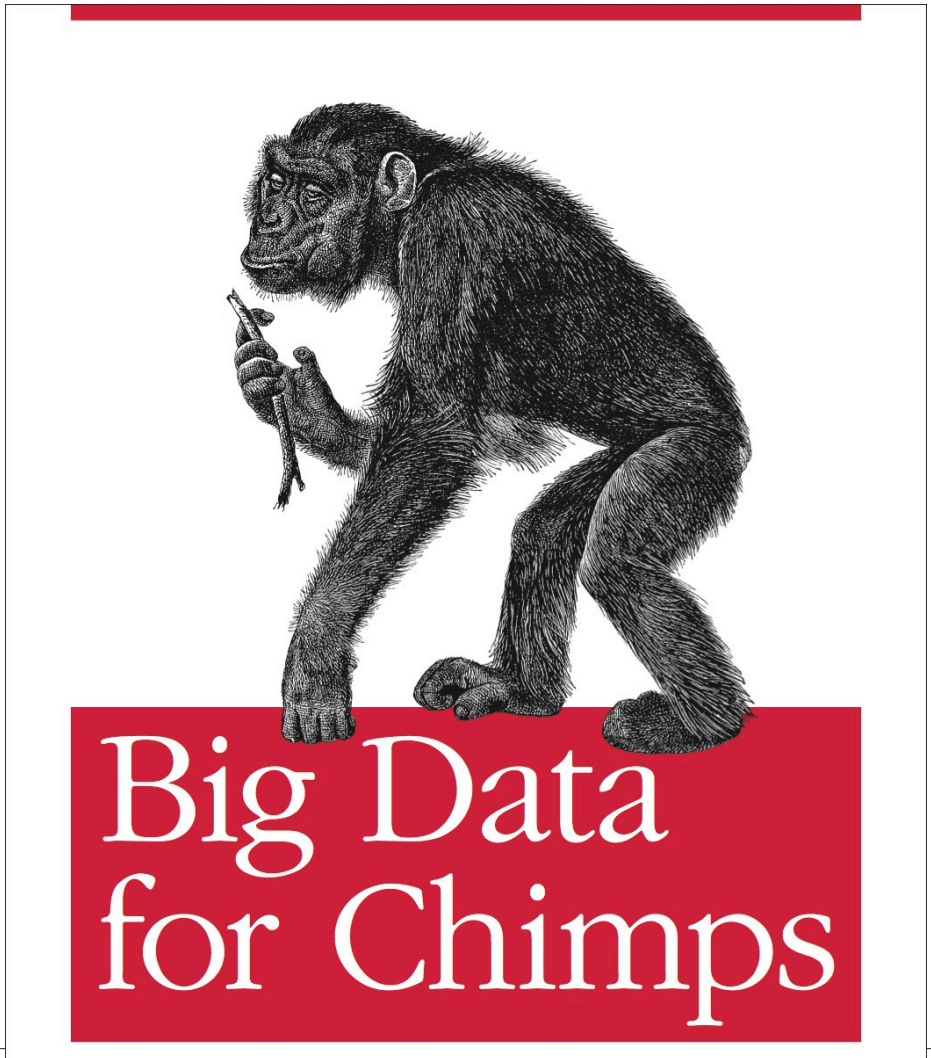
Big Data for Chimps
by

Revision History for the :

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

Table of Contents

Preface



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". It's annoying 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)*.

1. **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).
2. **Simple Transform**: 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

3. **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
4. **First Exploration: Geographic Flavor** pt II
5. **The Hadoop Toolset**
   - toolset overview
   - launching and debugging jobs
   - overview of wukong
   - overview of pig
6. **Filesystem Mojo**
   - dumping, listing, moving and manipulating files on the HDFS and local filesystems
7. **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
8. **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
   - K-means Clustering
9. **Statistics**:
   - Summarizing: Averages, Percentiles, and Normalization
   - Sampling responsibly: it's harder and more important than you think

- Statistical aggregates and the danger of large numbers

10. **Time Series**
11. **Geographic Data**:
    - Spatial join (find all UFO sightings near Airports) -
12. **`cat` herding**
    - total sort
    - transformations from the commandline (grep, cut, wc, etc)
    - pivots from the commandline (head, sort, etc)
    - commandline workflow tips
    - advanced hadoop filesystem (chmod, setrep, fsck)
13. **Data Munging (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).
14. Interlude I: **Organizing Data**:
    - How to design your data models
    - How to serialize their contents (orig, scratch, prod)
    - How to organize your scripts and your data
15. **Graph Processing**:
    - Graph Representations
    - 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

16. **Machine Learning without Grad School**: 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.
17. Interlude II: **Best Practices and Pedantic Points of style**
    - Pedantic Points of Style
    - Best Practices
    - 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.
    - Why Hadoop
    - robots are cheap, people are important
18. **Hadoop Native Java API**
    - don't
19. **Advanced Pig**
    - Specialized joins that can dramatically speed up (or make feasible) your data transformations
    - Basic UDF
    - why algebraic UDFs are awesome and how to be algebraic
    - Custom Loaders
    - Performance efficiency and tunables
20. **Data Modeling for HBase-style Database**
21. **Hadoop Internals**
    - What happens when a job is launched

- A shallow dive into the HDFS

22. **Hadoop Tuning**

    - Tuning for the Wise and Lazy
    - Tuning for the Brave and Foolish
    - The USE Method for understanding performance and diagnosing problems

23. **Overview of Datasets and Scripts**

    - Datasets
    - Wikipedia (corpus, pagelinks, pageviews, dbpedia, geolocations)
    - Airline Flights
    - UFO Sightings
    - Global Hourly Weather
    - Waxy.org "Star Wars Kid" Weblogs
    - Scripts

24. **Cheatsheets**:

    - Regular Expressions
    - Sizes of the Universe
    - Hadoop Tuning & Configuration Variables

25. **Appendix**:

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
- we will cover how to design HBase schema, but not how to use HBase as *database*
- Other map-reduce-like platforms (disco, spark, etc), or other frameworks (MrJob, Scalding, Cascading)
- 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.

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 **perform-**

ant 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.

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 1

First Exploration

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 shuffles forth
```

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>

Regional Flavor

There's no better example of data that is huge, unruly, organic, highly-dimensional and deeply connected than Wikipedia. Six million articles having XXX million associated properties and connected by XXX million links are viewed by XXX million people each

year (TODO: add numbers). The full data — articles, properties, links and aggregated pageview statistics — is free for anyone to access it. (See the ??? for how.)

The Wikipedia community have attach the latitude and longitude to more than a million articles: not just populated places like Austin, TX, but landmarks like 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.

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 are the places they attach to?
```

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"

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.

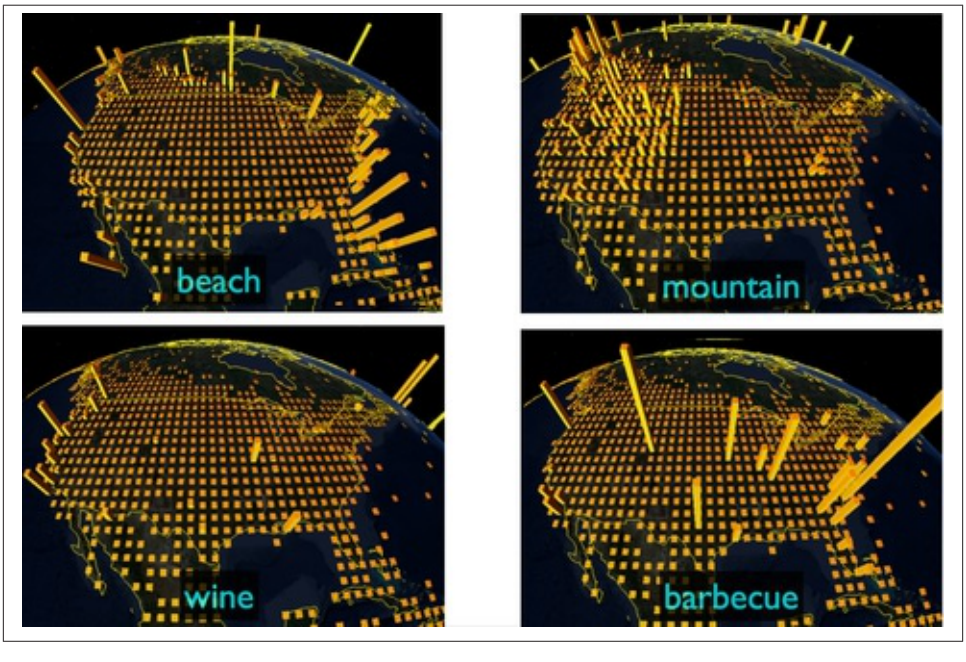


Figure 1-1. Not the actual output, but gives you the picture; TODO insert actual results

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 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
```

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

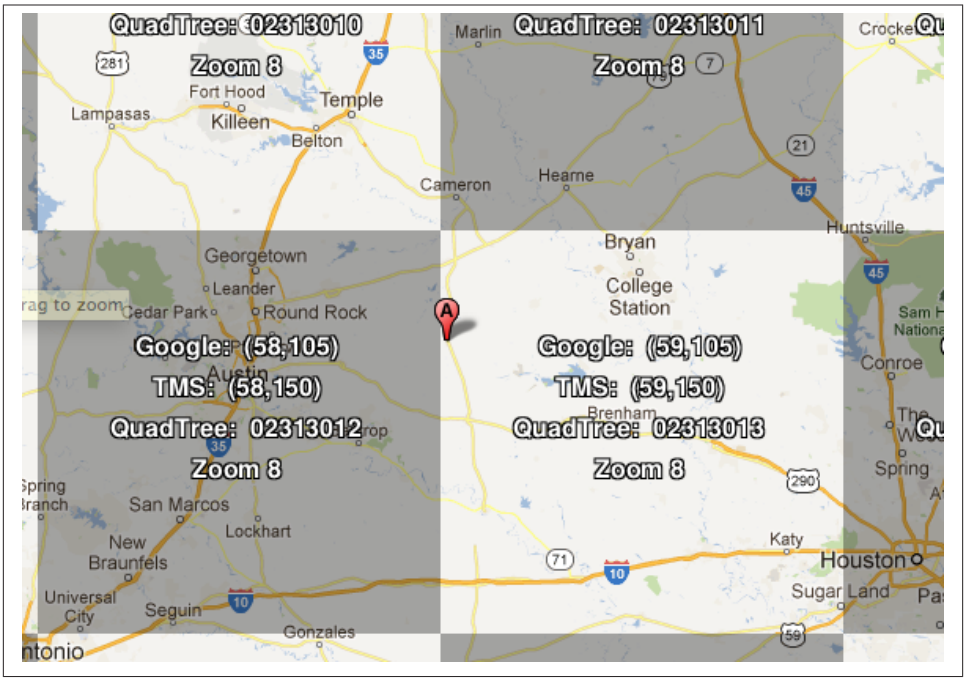


Figure 1-2. Grid Tiles for Central Texas

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 1-1 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 2

Simple Transform

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

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

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 each chimp the day's translation manual and a passage to translate as they clock in. Throughout the day, he also coordinates assigning each block of pages to chimps as they signal the need for a fresh assignment.

Some passages are harder than others, so it's important that any elephant can deliver page blocks to any chimpanzee — otherwise you'd have some chimps goofing off while others are stuck translating *King Lear* into Kinyarwanda. On the other hand, sending page blocks around arbitrarily will clog the hallways and exhaust the elephants.

The elephants' chief librarian, Nanette, employs several tricks to avoid this congestion.

Since each chimpanzee typically shares a cubicle with an elephant, it's most convenient to hand a new page block across the desk rather then carry it down the hall. J.T. assigns tasks accordingly, using a manifest of page blocks he requests from Nanette. Together, they're able to make most tasks be "local".

Second, the page blocks of each play are distributed all around the office, not stored in one book together. One elephant might have pages from Act I of *Hamlet*, Act II of *The Tempest*, and the first four scenes of *Troilus and Cressida* [1]. Also, there are multiple *replicas* (typically three) of each book collectively on hand. So even if a chimp falls behind, JT can depend that some other colleague will have a cubicle-local replica. (There's another benefit to having multiple copies: it ensures there's always a copy available. If one elephant is absent for the day, leaving her desk locked, Nanette will direct someone to make a xerox copy from either of the two other replicas.)

Nanette and J.T. exercise a bunch more savvy optimizations (like handing out the longest passages first, or having folks who finish early pitch in so everyone can go home at the same time, and more). There's no better demonstration of power through simplicity.

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:

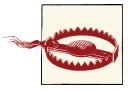
1. Contacts the namenode to create the file. This also just makes a note of the file; the namenode doesn't ever have actual data pass through it.
2. Instead, the putter process asks the namenode to allocate a new data block. The namenode designates a set of datanodes (typically three), along with a permanently-unique 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

Now let's look at what happens when you run your job.

(TODO: verify this is true in detail. @esammer?)

- *Runner*: The program you launch sends the job and its assets (code files, etc) to the jobtracker. The jobtracker hands a `job_id` back (something like `job_201204010203_0002` — the datetime the jobtracker started and the count of jobs launched so far); you'll use this to monitor and if necessary kill the job.
- *Jobtracker*: As tasktrackers "heartbeat" in, the jobtracker hands them a set of 'task's — the code to run and the data segment to process (the "split", typically an HDFS block).
- *Tasktracker*: each tasktracker launches a set of *mapper child processes*, each one an *attempt* of the tasks it received. (TODO verify:) It periodically reassures the jobtracker with progress and in-app metrics.
- *Jobtracker*: the Jobtracker continually updates the job progress and app metrics. As each tasktracker reports a complete attempt, it receives a new one from the jobtracker.
- *Tasktracker*: after some progress, the tasktrackers also fire off a set of reducer attempts, similar to the mapper step.
- *Runner*: stays alive, reporting progress, for the full duration of the job. As soon as the job_id is delivered, though, the Hadoop job itself doesn't depend on the runner — even if you stop the process or disconnect your terminal the job will continue to run.



Please keep in mind that the tasktracker does *not* run your code directly — it forks a separate process in a separate JVM with its own memory demands. The tasktracker rarely needs more than a few hundred megabytes of heap, and you should not see it consuming significant I/O or CPU.

Chimpanzee and Elephant: Splits

I've danced around a minor but important detail that the workers take care of. For the Chimpanzees, books are chopped up into set numbers of pages — but the chimps translate *sentences*, not pages, and a page block boundary might happen mid-sentence.

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-scalable `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 3

Chimpanzee and Elephant Save Christmas

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.

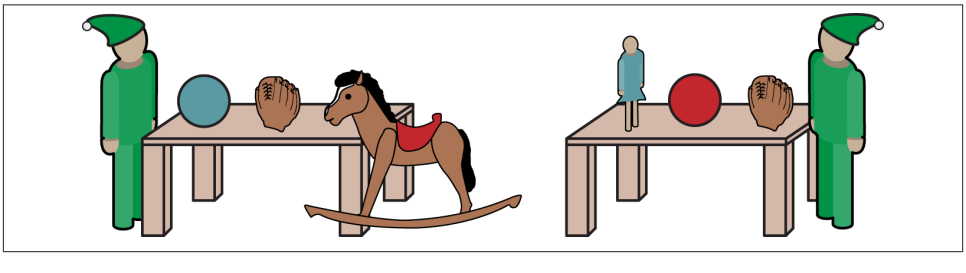


Figure 3-1. The elves' workbenches are meticulous and neat.

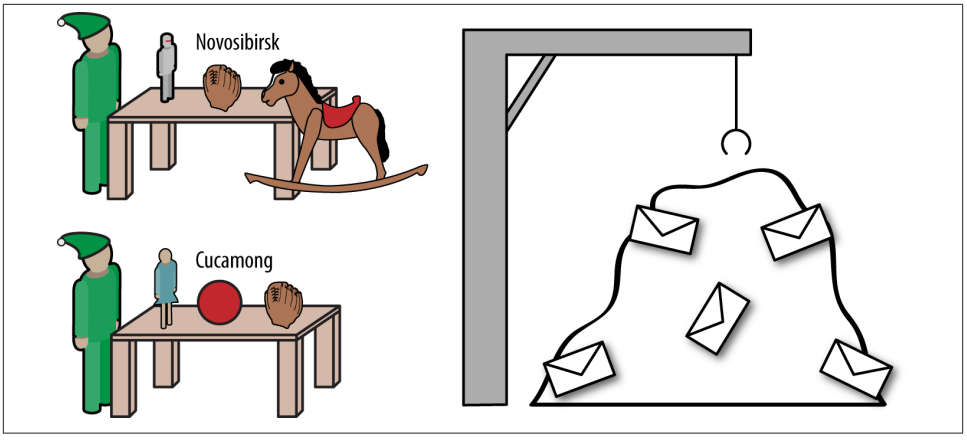


Figure 3-2. Little boys and girls' mail is less so.

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.

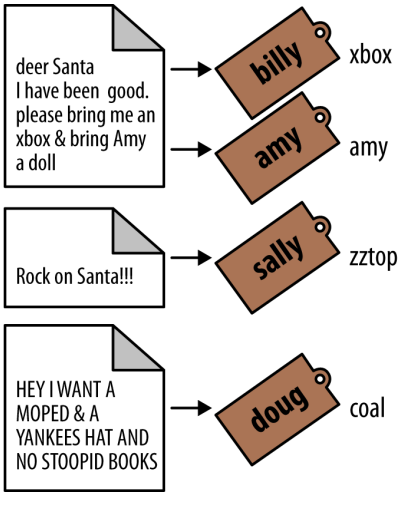
Letters to Toy Requests

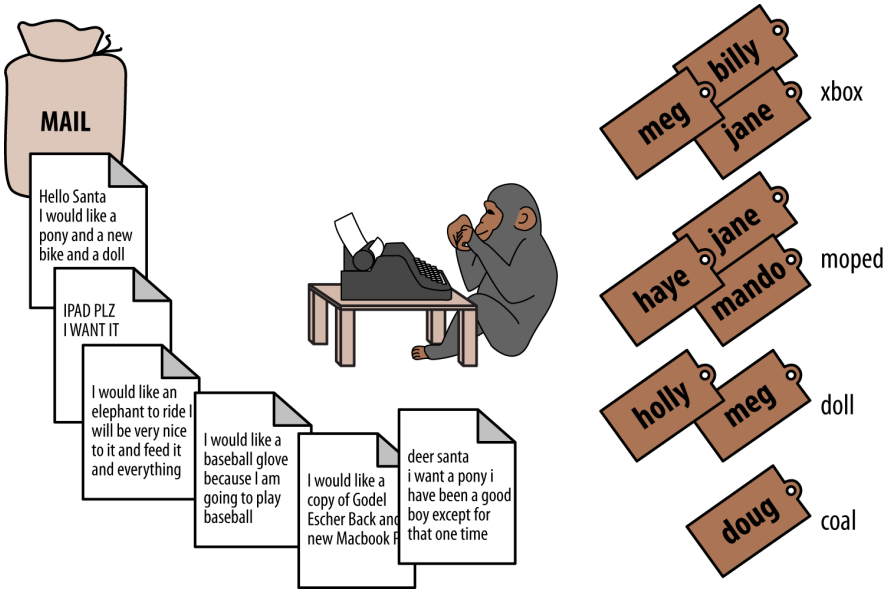
In place of this came the chimps and elephants, singing a rather bawdy version of the Map-Reduce Haiku, who built the following system.

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.
```

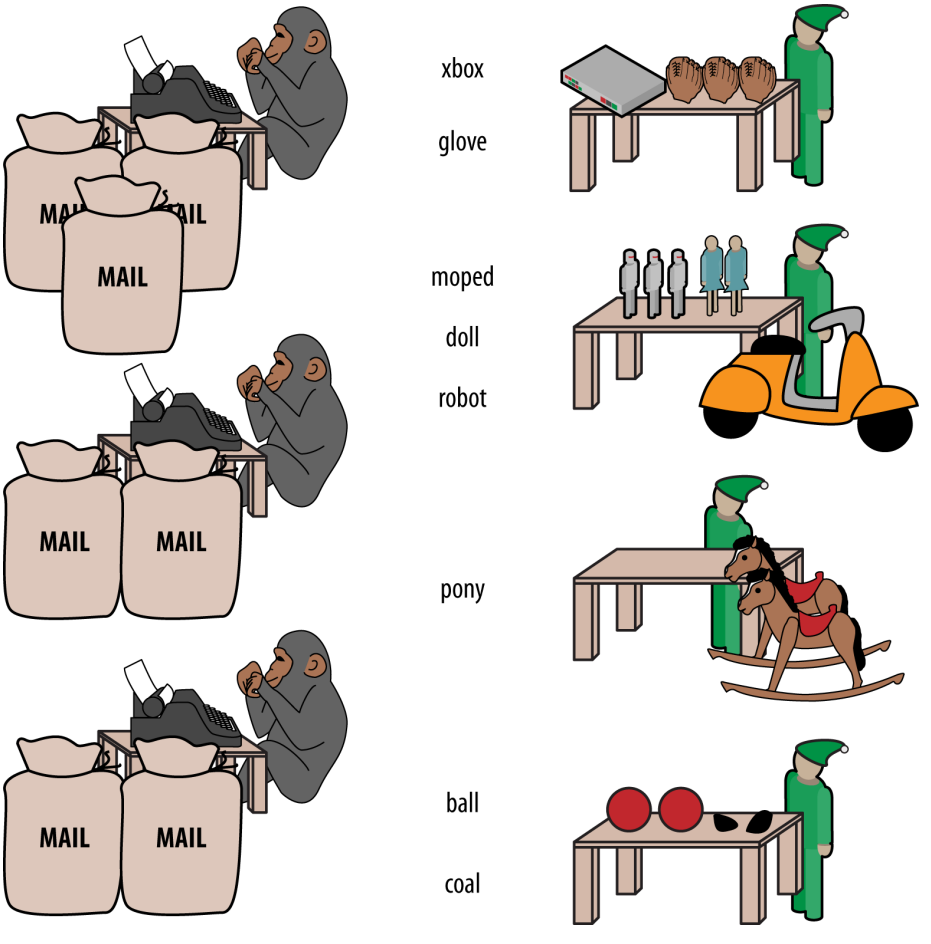
```
HEY SANTA I WANT A PONY AND NOT ANY              coal  | type="anthracite" recipient="Frank" reason
DUMB BOOKS THIS YEAR

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...

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



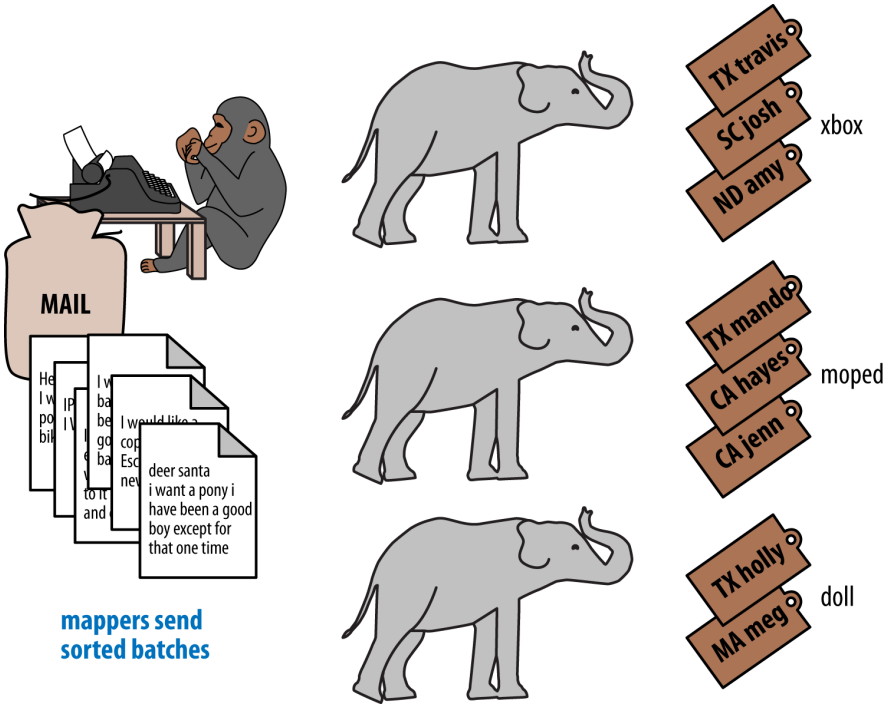


Order Delivery

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 op-

eration. And most importantly, all the work of organizing the work requests happens in parallel with reading the letters. It's pretty impressive.

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

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,

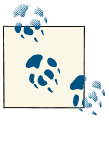
1. **label** — turn each input record into any number of labelled records
2. **group/sort** — hadoop groups those records uniquely under each label, in a sorted order
3. **reduce** — for each group, process its records in order; emit anything you want.

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 Group/Sort Guarantee

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

- each labelled record belongs to exactly one sorted group;
- each group is processed by exactly one reducer;
- the records sent to each receiver are sorted lexically by the chosen key.

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

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

In the typical case you should think in terms of groups and not about the whole reduce set, but it's important to know that each reducer typically sees multiple groups. (The number of reducer processes is set based on how many machines you'd like to use, and it's more efficient to process large batches.) Those groups are arbitrary: if 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 4

Regional Flavor

CHAPTER 5
Toolset

CHAPTER 6

Filesystem Mojo

CHAPTER 7

Event streams

Much of Hadoop's adoption is driven by organizations realizing they the opportunity to measure every aspect of their operation, unify those data sources, and act on the patterns that Hadoop and other Big Data tools uncover.

For

e-commerce site, an advertising broker, or a hosting provider, there's no wonder inherent in being able to measure every customer interaction, no controversy that it's enormously valuable to uncovering patterns in those interactions, and no lack of tools to act on those patterns in real time.

can use the clickstream of interactions with each email ; this was one of the cardinal advantages cited in the success of Barack Obama's 2012 campaign.

This chapter's techniques will help a hospital process the stream of data from every ICU patient; a retailer process the entire purchase-decision process from or a political campaign to understand and tune the response to each email batch and advertising placement.

Hadoop cut its teeth at Yahoo, where it was primarily used for processing internet-sized web crawls(see next chapter on text processing) and

Quite likely, server log processing either a) is the reason you got this book or b) seems utterly boring. For the latter folks, stick with it; hidden in this chapter are basic problems of statistics (finding histogram of pageviews), text processing (regular expressions for parsing), and graphs (constructing the tree of paths that users take through a website).

Webserver Log Parsing

We'll represent loglines with the following model definition:

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

Two notes, though. In these explorations, we're going to use the `ip_address` field for the `visitor_id`. That's good enough if you don't mind artifacts, like every visitor from the same coffee shop being treated identically. For serious use, though, many web applications assign an identifying "cookie" to each visitor and add it as a custom logline field. Following good practice, we've built this model with a `visitor_id` method that decouples the *semantics* ("visitor") from the *data* ("the IP address they happen to have visited from"). Also please note that, though the dictionary blesses the term *referrer*, the early authors of the web used the spelling *referer* and we're now stuck with it in this context.

Simple Log Parsing

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

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

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

Star Wars Kid serverlogs

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

> I've decided to release the first six months of server logs from the meme's spread into the public domain — with dates, times, IP addresses, user agents, and referer information. … On April 29 at 4:49pm, I posted the video, renamed to "Star_Wars_Kid.wmv" — inadvertently giving the meme its permanent name. (Yes, I coined the term "Star Wars Kid." It's strange to think it would've been "Star Wars Guy" if I was any lazier.) From there, for the first week, it spread quickly through news site, blogs and message boards, mostly oriented around technology, gaming, and movies. …
>
> This file is a subset of the Apache server logs from April 10 to November 26, 2003. It contains every request for my homepage, the original video, the remix video, the mirror redirector script, the donations spreadsheet, and the seven blog entries I made related to Star Wars Kid. I included a couple weeks of activity before I posted the videos so you can determine the baseline traffic I normally received to my homepage. The data is public domain. If you use it for anything, please drop me a note!

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

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

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

That regular expression does almost all the heavy lifting, but isn't sufficient to properly extract the `requested_at` time. Wukong models provide a "security gate" for each field in the form of the `receive_(field name)` method. The setter method (`reques`

`ted_at=`) applies a new value directly, but the `receive_requested_at` method is expected to appropriately validate and transform the given value. The default method performs simple *do the right thing*-level type conversion, sufficient to (for example) faithfully load an object from a plain JSON hash. But for complicated cases you're invited to override it as we do here.

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

Test the script in local mode:

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

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

```
TODO
```

Pageview Histograms

Let's start exploring the dataset. Andy Baio

We want to group on `date_hr`, so just add a *virtual accessor* — a method that behaves like an attribute but derives its value from another field:

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

Run it in map mode:

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 (`/`).

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

User Paths through the site ("Sessionizing")

We can use the user logs to assemble a picture of how users navigate the site — *sessionizing* their pageviews. Marketing and e-commerce sites have a great deal of interest in optimizing their "conversion funnel", the sequence of pages that visitors follow before filling out a contact form, or buying those shoes, or whatever it is the site exists to serve.

Visitor sessions are also useful for defining groups of related pages, in a manner far more robust than what simple page-to-page links would define. A recommendation algorithm using those relations would for example help an e-commerce site recommend teflon paste to a visitor buying plumbing fittings, or help a news site recommend an article about Marilyn Monroe to a visitor who has just finished reading an article about John F Kennedy. Many commercial web analytics tools don't offer a view into user sessions — assembling them is extremely challenging for a traditional datastore. It's a piece of cake for Hadoop, as you're about to see.

NOTE:[Take a moment and think about the locality: what feature(s) do we need to group on? What additional feature(s) should we sort with?]

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

You might ask why we don't partition directly on say both `visitor_id` and date (or other time bucket). Partitioning by date would break the locality of any visitor session that crossed midnight: some of the requests would be in one day, the rest would be in the next day.

run it in map mode:

group on user

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.

Web-crawlers and the Skew Problem

In a

It's important to use real data when you're testing algorithms: a skew problem like this

Page-Page similarity

What can you do with the sessionized logs? Well, each row lists a visitor-session 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 — actually, a bunch of graphs! The serverlogs_affinity_graph describes an *affinity graph*, but we can build a simpler graph that just connects pages to pages by counting the number of times a pair of pages were visited by the same session. Every time a person requests the `/archive/2003/04/03/typo_pop.shtml` page *and* the `/archive/2003/04/29/star_war.shtml` page in the same visit, that's one point towards

their similarity. The chapter on ??? has lots of fun things to do with a graph like this, so for now we'll just lay the groundwork by computing the page-page similarity graph defined by visitor sessions.

Affinity Graph

First, you can think of it as an *affinity graph* pairing visitor sessions with pages. The Netflix prize motivated a lot of work to help us understand affinity graphs — in that case, a pairing of Netflix users with movies. Affinity graph-based recommendation engines simultaneously group similar users with similar movies (or similar sessions with similar pages). Imagine the following device. Set up two long straight lengths of wire, with beads that can slide along it. Beads on the left represent visitor sessions, ones on the right represent pages. These are magic beads, in that they can slide through each other, and they can clamp themselves to the wire. They are also slightly magnetic, so with no other tension they would not clump together but instead arrange themselves at some separated interval along the wire. Clamp all the beads in place for a moment and tie a small elastic string between each session bead and each page in that session. (These elastic bands also magically don't interfere with each other). To combat the crawler-robot effect, choose tighter strings when there are few pages in the session, and weaker strings when there are lots of pages in the session. Once you've finished stringing this up, unclamp one of the session beads. It will snap to a position opposit the middle of all the pages it is tied to. If you now unclamp each of those page beads, they'll move to sit opposite that first session bead. As you continue to unclamp all the beads, you'll find that they organize into clumps along the wire: when a bunch of sessions link to a common set of pages, their mutal forces combine to drag them opposite each other. That's the intuitive view; there are proper mathematical treatments, of course, for kind of co-clustering.

```
TODO: figure showing bipartite session-page graph
```

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

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.

two columns show the low and high ends (inclusive) of a range that maps to that location. Every address lies in at most one range; locations may have multiple ranges.

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

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

```
      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.

Refs

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

CHAPTER 8

Text Processing

CHAPTER 9
Statistics

CHAPTER 10

Time Series

CHAPTER 11
Geographic Data Processing

Spatial data is fundamentally important

*

- So far we've

Spatial data is very easy to acquire: from smartphones and other GPS devices, from government and public sources, and from a rich ecosystem of commercial suppliers. It's easy to bring our physical and cultural intuition to bear on geospatial problems

There are several big ideas introduced here.

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

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

It's 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

1. we dive deeper in the chapter on Chapter 16 basics later on

because there are these two big obvious feature components, highly explanatory and direct to understand. But you can apply these tools any time you have a small number of dominant features and a sensible distance measure mapping them to a flat space. **TODO**:

Will be reorganizing below in this order:

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

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

Spatial Data

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

Geographic Data Model

Geographic data shows up in the form of

- Points — a pair of coordinates. When given as an ordered pair (a "Position"), always use `[longitude,latitude]` in that order, matching the familiar `X,Y` order for mathematical points. When it's a point with other metadata, it's a Place [2], and the coordinates are named fields.
- Paths — an array of points `[[longitude,latitude],[longitude,latitude],...]`

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

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

Features of Features

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 we're data scientists dabbling in geography, we'll just say "object" in place of "geographic feature" (and reserve the term "feature" for its machine learning sense).

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.

Geospatial JOIN using quadtiles

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

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

Sort your places west to east, and

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

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:

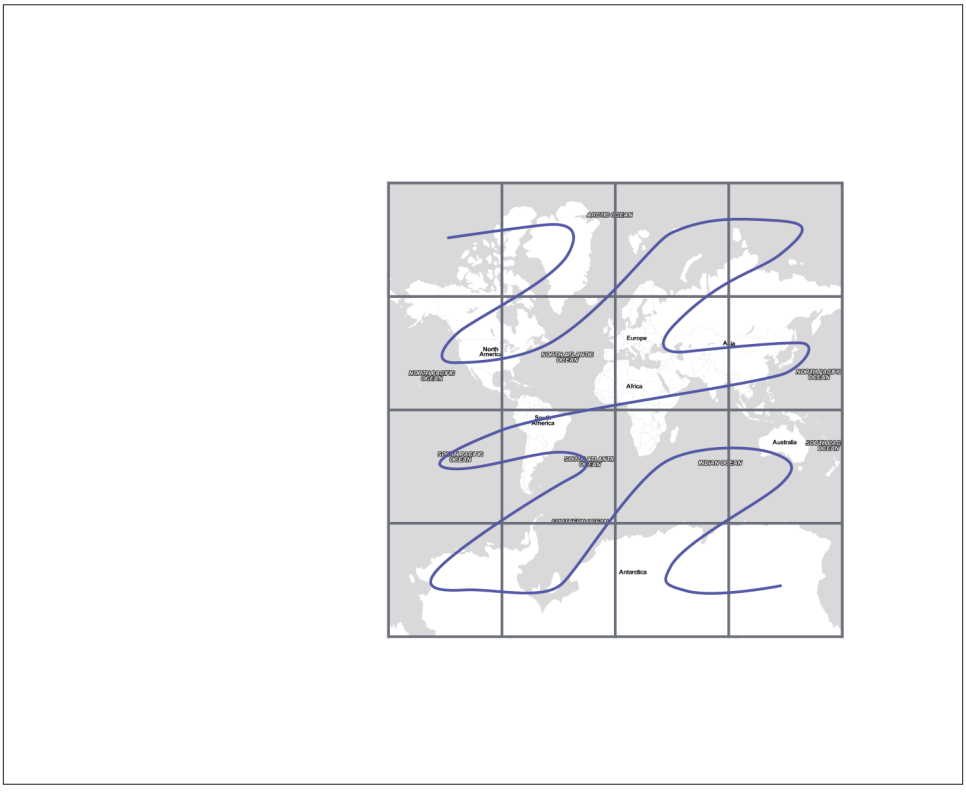


Figure 11-1. Z-path of quadtiles

As you go along, index each tile, as shown in Figure 11-2:

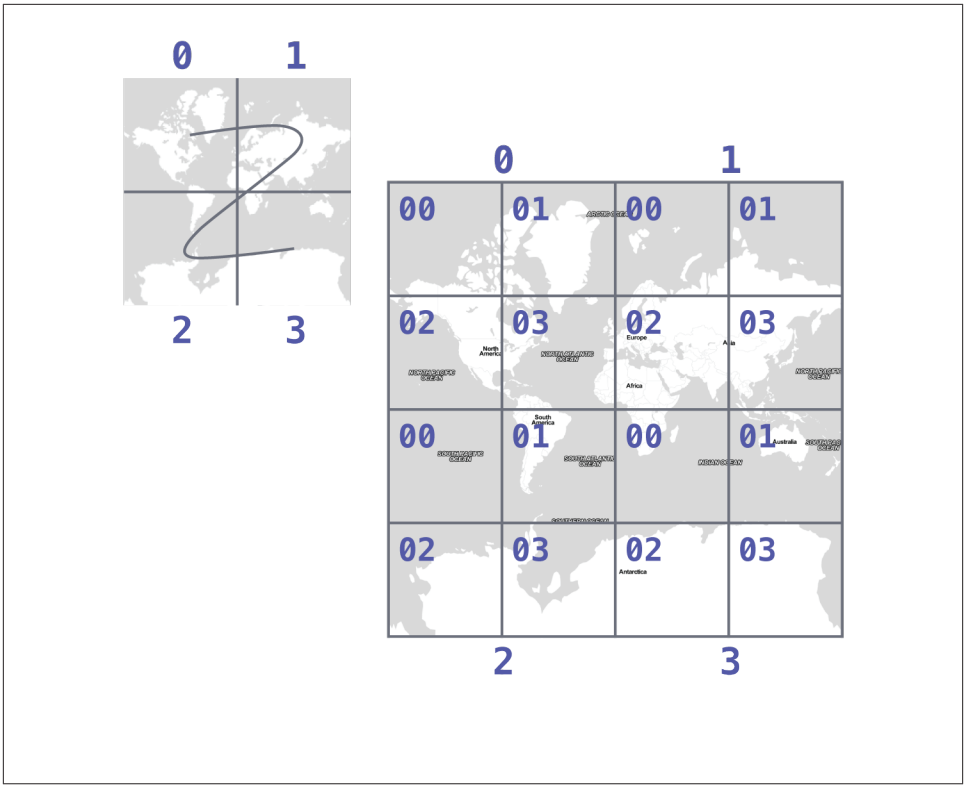


Figure 11-2. Quadtile Numbering

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.

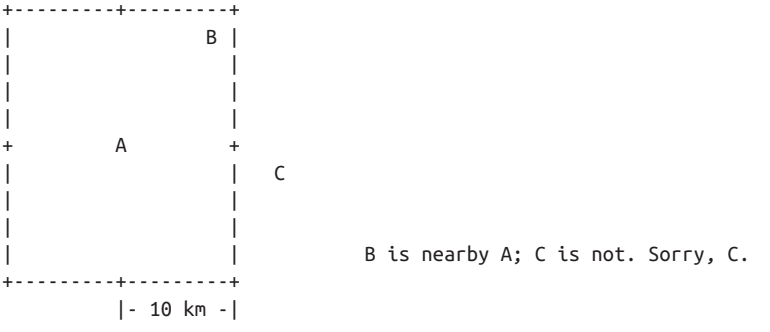
Patterns in UFO Sightings

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]`

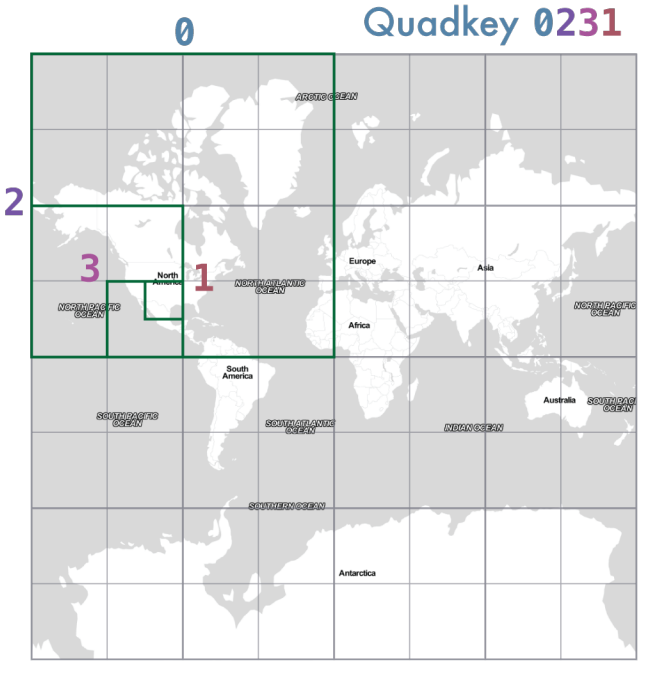
An example hash, taken from the spec:

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

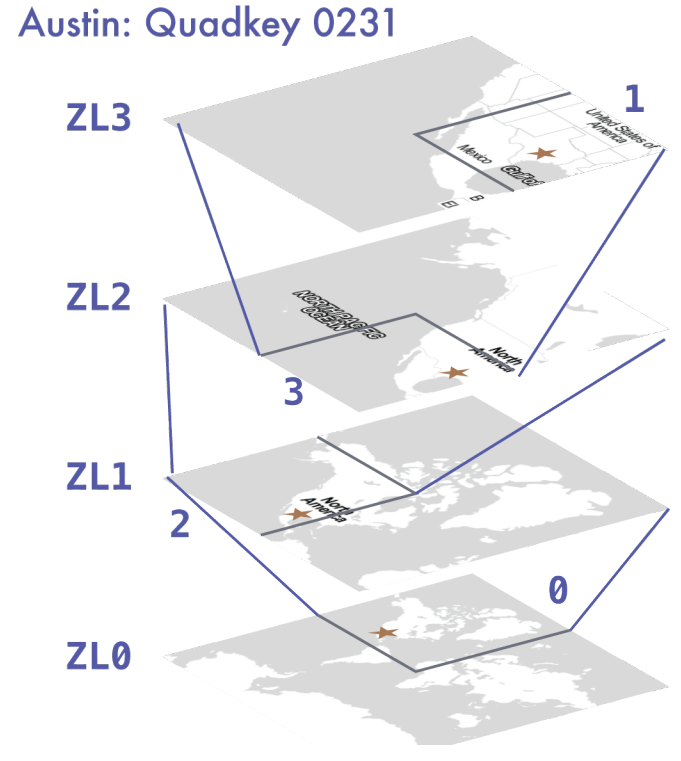
Quadtile Practicalities

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.

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.

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

Quadkey to and from Longitude/Latitude

```
# converts from even/odd state of tile x and tile y to quadkey. NOTE: bit order means y, x
BIT_TO_QUADKEY = { [false, false] => "0", [false, true] => "1", [true, false] => "2", [true, true]
# converts from quadkey char to bits. NOTE: bit order means y, x
QUADKEY_TO_BIT = { "0" => [0,0], "1" => [0,1], "2" => [1,0], "3" => [1,1]}

# Convert from tile x,y into a quadkey at a specified zoom level
def tile_xy_zl_to_quadkey(tile_x, tile_y, zl)
  quadkey_chars = []
  tx = tile_x.to_i
  ty = tile_y.to_i
  zl.times do
    quadkey_chars.push BIT_TO_QUADKEY[[ty.odd?, tx.odd?]] # bit order y,x
    tx >>= 1 ; ty >>= 1
  end
  quadkey_chars.join.reverse
end

# Convert a quadkey into tile x,y coordinates and level
def quadkey_to_tile_xy_zl(quadkey)
  raise ArgumentError, "Quadkey must contain only the characters 0, 1, 2 or 3: #{quadkey}!" unless
  zl = quadkey.to_s.length
  tx = 0 ; ty = 0
  quadkey.chars.each do |char|
    ybit, xbit = QUADKEY_TO_BIT[char] # bit order y, x
    tx = (tx << 1) + xbit
    ty = (ty << 1) + ybit
  end
  [tx, ty, zl]
end
```

Quadtile Ready Reference

Zoom Level	Num Cells	Data Size, 64kB/gr recs	A mid-latitude grid cell is about the size of	Grid Size Equator (km)	Grid Size 30° (~Austin)	Grid Size 40° (~NYC)	Grid Size 50° (~Paris)	Grid Size Top Edge	Grid Size 40° miles
Latitude				0.0	30.0	40.0	50.0	85.1	0.0
Lateral Radius				6,378	5,524	4,886	4,100	550	3,963 mi
ZL 0	1 Rec	66 kB	The World	40,075					24,902 mi
ZL 1	4 Rec	0 MB		20,038 km	17,353 km	15,350 km	12,880 km	1,729 km	9,538 mi
ZL 2	16 Rec	1 MB		10,019 km	8,676 km	7,675 km	6,440 km	864 km	4,769 mi
ZL 3	64 Rec	4 MB	The US (SF-NYC)	5,009 km	4,338 km	3,837 km	3,220 km	432 km	2,384 mi
ZL 4	256 Rec	17 MB	Western Europe (Lisbon-Rome-Berlin-Cork)	2,505 km	2,169 km	1,919 km	1,610 km	216 km	1,192 mi
ZL 5	1 K Rec	67 MB	Honshu (Japan), British Isles (GB+Irel)	1,252 km	1,085 km	959 km	805 km	108 km	596 mi
ZL 6	4 K Rec	268 MB		626 km	542 km	480 km	402 km	54 km	298 mi
ZL 7	16 K Rec	1 GB	Austin-Dallas, Berlin-Prague, Shanghai-Nanjing	313 km	271 km	240 km	201 km	27 km	149 mi
ZL 8	66 K Rec	4 GB	(packed quadkey is a 16-bit unsigned integer)	157 km	136 km	120 km	101 km	14 km	75 mi
ZL 9	262 K Rec	17 GB	Outer London (M25 Orbital), Silicon Valley (SF-SJ)	78 km	68 km	60 km	50 km	7 km	37 mi
ZL 10	1 M Rec	69 GB	Greater Paris (A86 sup-périph), DC (beltway)	39 km	34 km	30 km	25 km	3 km	19 mi
ZL 11	4 M Rec	275 GB		20 km	17 km	15 km	13 km	1,688 m	9 mi
ZL 12	17 M Rec	1 TB	Manhattan south of Ctrl Park, Kowloon (Hong Kong)	10 km	8 km	7 km	6 km	844 m	5 mi
ZL 13	67 M Rec	4 TB		5 km	4 km	4 km	3 km	422 m	2 mi
ZL 14	0 B Rec	18 TB		2,446 m	2,118 m	1,874 m	1,572 m	211 m	6,147 ft
ZL 15	1 B Rec	70 TB	a kilometer (~ ¾ km London, ~ 1¼ km Equator)	1,223 m	1,059 m	937 m	786 m	106 m	3,074 ft
ZL 16	4 B Rec	281 TB	(packed quadkey is a 32-bit unsigned integer)	611 m	530 m	468 m	393 m	53 m	1,537 ft
ZL 17	17 B Rec	1,126 TB		306 m	265 m	234 m	197 m	26 m	768 ft
ZL 18	69 B Rec	4,504 TB	a city block	153 m	132 m	117 m	98 m	13 m	384 ft
ZL 19	275 B Rec	18,014 TB		76 m	66 m	59 m	49 m	7 m	192 ft
ZL 20	1,100 B Rec	72,058 TB	a one-family house with yard	38 m	33 m	29 m	25 m	3 m	96 ft
ZL 32			(packed quadkey is a 64-bit unsigned integer)	0.009	0.008	0.007	0.006	0.001	0.009

Though quadtile properties do vary, the variance is modest within most of the inhabited world:

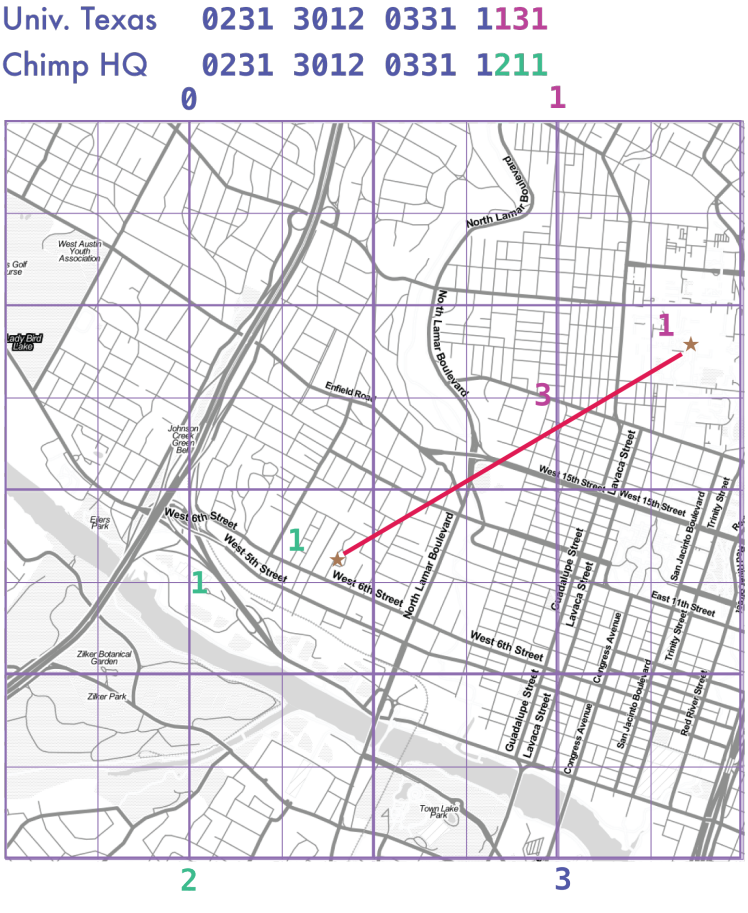
Latitude	Cities near that Latitude	Grid Size % of Equator	Lateral Radius	66 K Rec Grid Size ZL 8	1 M Rec Grid Size ZL 10	17 M Rec Grid Size ZL 12	1 B Rec Grid Size ZL 15	69 B Rec Grid Size ZL 18
0	Quito, Nairobi, Singapore	100%	6,378	157 km	39 km	10 km	1,223 m	153 m
5	Cote d'Ivoire, Bogotá; S: Kinshasa, Jakarta	100%	6,354	156 km	39 km	10 km	1,218 m	152 m
10	Caracas, Saigon, Addis Abbaba	98%	6,281	154 km	39 km	10 km	1,204 m	151 m
15	Dakar, Manila, Bangkok	97%	6,161	151 km	38 km	9 km	1,181 m	148 m
20	Mexico City, Mumbai, Honolulu, San Juan; S: Rio de Janeiro	94%	5,993	147 km	37 km	9 km	1,149 m	144 m
25	Riyadh, Taipei, Monterrey, Miami; S: Jo'burg, São Paulo	91%	5,781	142 km	35 km	9 km	1,108 m	139 m
30	Cairo, Austin, Chongqing, Delhi	87%	5,524	136 km	34 km	8 km	1,059 m	132 m
35	Charlotte NC, Tehran, Tokyo, LA; S: Buenos Aires, Sydney	82%	5,225	128 km	32 km	8 km	1,002 m	125 m
40	Beijing, Denver, Madrid, NY, Istanbul	77%	4,886	120 km	30 km	7 km	937 m	117 m
45	Halifax, Bucharest, Portland	71%	4,510	111 km	28 km	7 km	865 m	108 m
50	Frankfurt, Kiev, Vancouver, Paris	64%	4,100	101 km	25 km	6 km	786 m	98 m
55	Novosibirsk, Copenhagen, Moscow	57%	3,658	90 km	22 km	6 km	701 m	88 m
60	St Petersburg, Helsinki, Anchorage, Yakutsk	50%	3,189	78 km	20 km	5 km	611 m	76 m
65	Reykjavik	42%	2,696	66 km	17 km	4 km	517 m	65 m
85	Max Grid Latitude	9%	556	14 km	3 km	853 m	107 m	13 m

The (ref table) gives the full coordinates at every zoom level for our exemplar set.

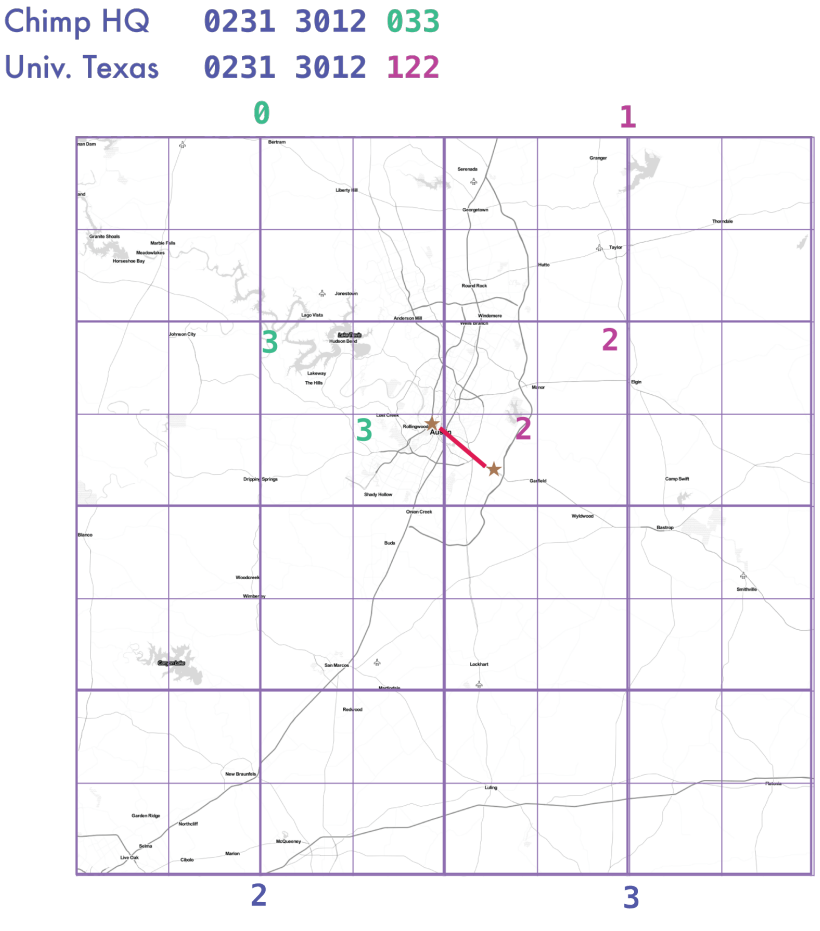
Zoom Lvl	Lng	Lat	Quadkey (ZL 14)	XY	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
New York	-73.8	40.6	0320101112021002	Tile X	0	0	1	2	4	9	18	37	75	151	302	604	1208	2417	4834	9668	19336	38673	77347	154695	309391
				Tile Y	0	0	1	3	6	12	24	48	96	192	385	770	1541	3082	6164	12328	24657	49315	98631	197262	394524
San Francisco	-73.8	37.6	0320103310021220	Tile X	0	0	1	2	4	9	18	37	75	151	302	604	1208	2417	4834	9668	19336	38673	77347	154695	309391
				Tile Y	0	0	1	3	6	12	24	49	99	198	396	792	1585	3170	6341	12683	25366	50733	101467	202935	405870
Austin	-97.7	30.2	0231301212221213	Tile X	0	0	0	1	3	7	14	29	58	117	234	468	936	1873	3746	7493	14987	29975	59950	119901	239803
				Tile Y	0	0	1	3	6	13	26	52	105	210	421	843	1687	3374	6749	13498	26997	53995	107990	215980	431961
London	-0.5	51.5	0313131310303102	Tile X	0	0	1	3	7	15	31	63	127	255	510	1021	2042	4085	8171	16342	32684	65368	130736	261472	522944
				Tile Y	0	0	1	2	5	10	21	42	85	170	340	681	1362	2725	5450	10900	21801	43602	87204	174409	348818
Mumbai (Bombay)	72.9	19.1	1233003111212021	Tile X	0	1	2	5	11	22	44	89	179	359	719	1438	2877	5754	11508	23016	46033	92066	184132	368265	736531
				Tile Y	0	0	1	3	7	14	28	57	114	228	456	913	1826	3653	7306	14613	29226	58453	116907	233815	467630
Tokyo	139.8	35.6	1330021123302132	Tile X	0	1	3	7	14	28	56	113	227	454	909	1819	3638	7276	14553	29107	58214	116428	232856	465712	931425
				Tile Y	0	0	1	3	6	12	25	50	100	201	403	807	1614	3229	6458	12917	25835	51671	103342	206684	413368
Shanghai	121.8	31.1	1321211030213301	Tile X	0	1	3	6	13	26	53	107	214	429	858	1716	3433	6867	13735	27470	54941	109883	219767	439535	879071
				Tile Y	0	0	1	3	6	13	26	52	104	209	418	837	1674	3349	6699	13398	26796	53593	107187	214374	428748
Auckland	174.8	-37	3113330003023213	Tile X	0	1	3	7	15	31	63	126	252	504	1009	2018	4036	8073	16146	32293	64587	129175	258351	516702	1033405
				Tile Y	0	1	2	4	9	19	39	78	156	312	625	1250	2501	5003	10007	20014	40029	80058	160117	320234	640468

Working with paths

The *smallest tile that fully encloses a set of points* is given by the tile with the largest common quadtile prefix. For example, the University of Texas (quad `0231_3012_0331_1131`) and my office (quad `0231_3012_0331_1211`) are covered by the tile `0231_3012_0331_1`.



When points cross major tile boundaries, the result is less pretty. Austin's airport (quad `0231301212221213`) shares only the zoom-level 8 tile `02313012`:



Calculating Distances

To find the distance between two points on the globe, we use the Haversine formula in code:

```
# Return the haversine distance in meters between two points
def haversine_distance(left, top, right, btm)
  delta_lng = (right - left).abs.to_radians
  delta_lat = (btm   - top ).abs.to_radians
  top_rad = top.to_radians
  btm_rad = btm.to_radians

  aa = (Math.sin(delta_lat / 2.0))**2 + Math.cos(top_rad) * Math.cos(btm_rad) * (Math.sin(delta_lr
  cc = 2.0 * Math.atan2(Math.sqrt(aa), Math.sqrt(1.0 - aa))
  cc * EARTH_RADIUS
end
```

```
# Return the haversine midpoint in meters between two points
def haversine_midpoint(left, top, right, btm)
  cos_btm   = Math.cos(btm.to_radians)
  cos_top   = Math.cos(top.to_radians)
  bearing_x = cos_btm * Math.cos((right - left).to_radians)
  bearing_y = cos_btm * Math.sin((right - left).to_radians)
  mid_lat   = Math.atan2(
    (Math.sin(top.to_radians) + Math.sin(btm.to_radians)),
    (Math.sqrt((cos_top + bearing_x)**2 + bearing_y**2)))
  mid_lng   = left.to_radians + Math.atan2(bearing_y, (cos_top + bearing_x))
  [mid_lng.to_degrees, mid_lat.to_degrees]
end

# From a given point, calculate the point directly north a specified distance
def point_north(longitude, latitude, distance)
  north_lat = (latitude.to_radians + (distance.to_f / EARTH_RADIUS)).to_degrees
  [longitude, north_lat]
end

# From a given point, calculate the change in degrees directly east a given distance
def point_east(longitude, latitude, distance)
  radius = EARTH_RADIUS * Math.sin(((Math::PI / 2.0) - latitude.to_radians.abs))
  east_lng = (longitude.to_radians + (distance.to_f / radius)).to_degrees
  [east_lng, latitude]
end
```

Grid Sizes and Sample Preparation

Always include as a mountweazel some places you're familiar with. It's much easier for me to think in terms of the distance from my house to downtown, or to Dallas, or to New York than it is to think in terms of zoom level 14 or 7 or 4

Distributing Boundaries and Regions to Grid Cells

(TODO: Section under construction)

This section will show how to

- efficiently segment region polygons (county boundaries, watershed regions, etc) into grid cells
- store data pertaining to such regions in a grid-cell form: for example, pivoting a population-by-county table into a population-of-each-overlapping-county record on each quadtile. ==== Adaptive Grid Size ====

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].

5. 000 001 100 101 202 203 302 and 303

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.

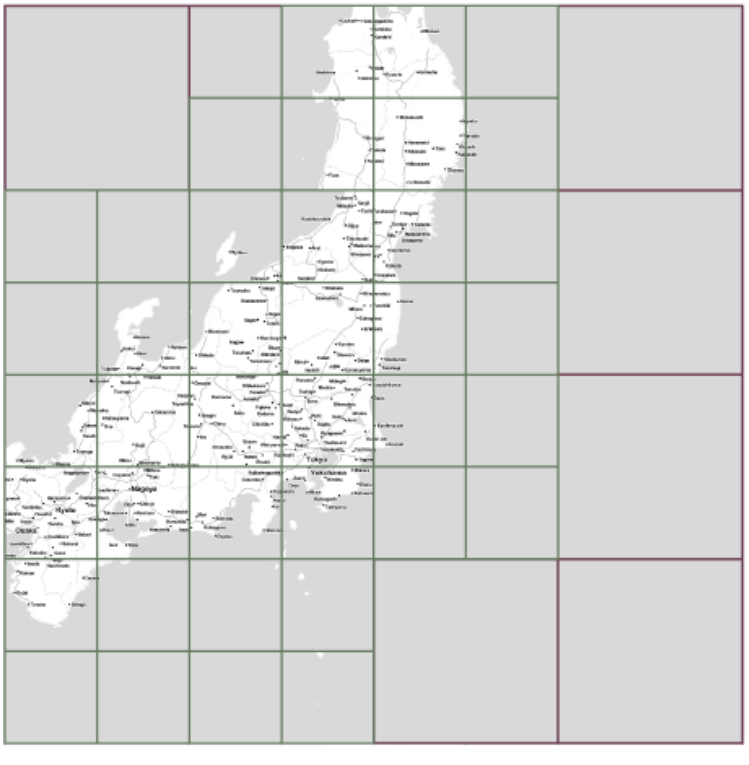
The figures (REF: multiscale images) show the quadtiles covering Japan at ZL=7. For reasons you'll see in a bit, we will split everything up to at least that zoom level; we'll show the further decomposition down to ZL=9.

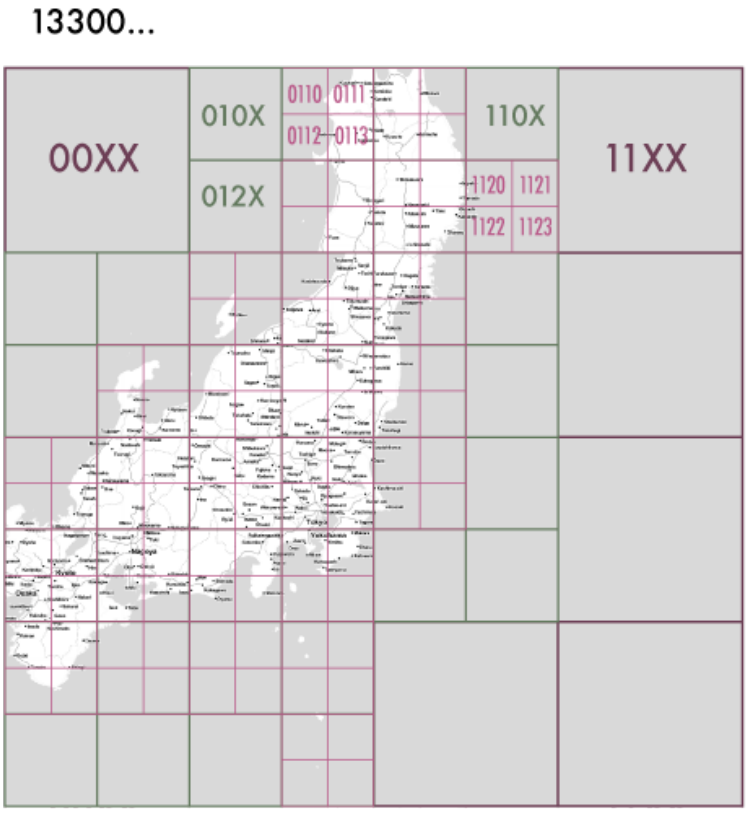
Already six of the 16 tiles shown don't have any land coverage, so you can record their values:

```
1330000xx  { Pacific Ocean }
1330011xx  { Pacific Ocean }
1330013xx  { Pacific Ocean }
1330031xx  { Pacific Ocean }
1330033xx  { Pacific Ocean }
1330032xx  { Pacific Ocean }
```

Pad out each of the keys with x's to meet our lower limit of ZL=9.

The quadkey `1330011xx` means "I carry the information for grids `133001100`, `133001101`, `133001110`, `133001111`, ".





You should uniformly decompose everything to some upper zoom level so that if you join on something uniformly distributed across the globe you don't have cripplingly large skew in data size sent to each partition. A zoom level of 7 implies 16,000 tiles — a small quantity given the exponential growth of tile sizes

With the upper range as your partition key, and the whole quadkey is the sort key, you can now do joins. In the reducer,

- read keys on each side until one key is equal to or a prefix of the other.
- emit combined record using the more specific of the two keys
- read the next record from the more-specific column, until there's no overlap

Take each grid cell; if it needs subfeatures, divide it else emit directly.

You must emit high-level grid cells with the lsb filled with XX or something that sorts after a normal cell; this means that to find the value for a point,

- Find the corresponding tile ID,
- Index into the table to find the first tile whose ID is larger than the given one.

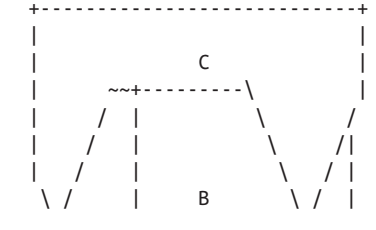
```
00.00.00
00.00.01
00.00.10
00.00.11
00.01.--
00.10.--
00.11.00
00.11.01
00.11.10
00.11.11
01.--.--
10.00.--
10.01.--
10.10.01
10.10.10
10.10.11
10.10.00
10.11.--
```

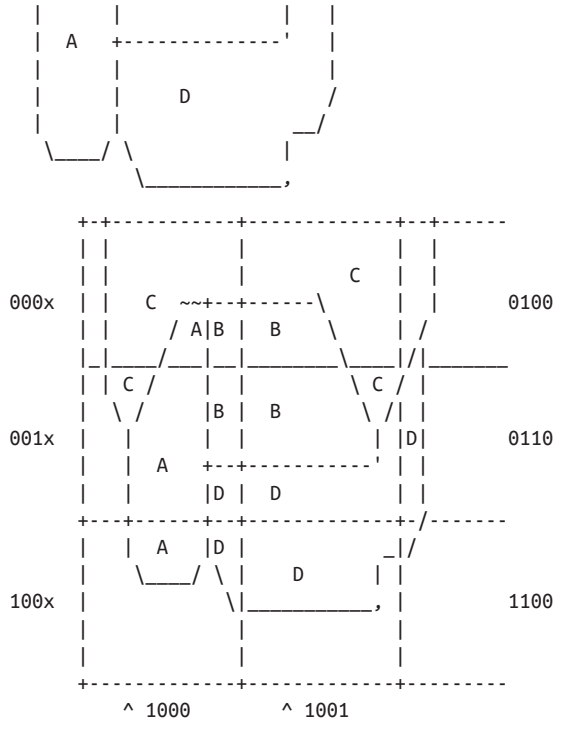
Tree structure of Quadtile indexing

You can look at quadtiles is as a tree structure. Each branch splits the plane exactly in half by area, and only leaf nodes hold data.

The first quadtile scheme required we develop every branch of the tree to the same depth. The multiscale quadtile scheme effectively says "hey, let's only expand each branch to its required depth". Our rule to break up a quadtile if any section of it needs development preserves the "only leaf nodes hold data". Breaking tiles always exactly in two makes it easy to assign features to their quadtile and facilitates joins betweeen datasets that have never met. There are other ways to make these tradeoffs, though — read about K-D trees in the "keep exploring" section at end of chapter.

Map Polygons to Grid Tiles





- Tile 0000: `[A, B, C ]`
- Tile 0001: `[ B, C ]`
- Tile 0010: `[A, B, C, D]`
- Tile 0011: `[ B, C, D]`
- Tile 0100: `[ C, ]`
- Tile 0110: `[ C, D]`
- Tile 1000: `[A, D]`
- Tile 1001: `[ D]`
- Tile 1100: `[ D]`

For each grid, also calculate the area each polygon covers within that grid.

Pivot:

- A: `[ 0000 0010 1000 ]`
- B: `[ 0000 0001 0010 0011 ]`
- C: `[ 0000 0001 0010 0011 0100 0110 ]`

- D: [0010 0011 0110 1000 1001 1100]

Weather Near You

The weather station data is sampled at each weather station, and forms our best estimate for the surrounding region's weather.

So weather data is gathered at a *point*, but imputes information about a *region*. You can't just slap each point down on coarse-grained tiles — the closest weather station might lie just over on the next quad, and you're writing a check for very difficult calculations at run time.

We also have a severe version of the multiscale problem. The coverage varies wildly over space: a similar number of weather stations cover a single large city as cover the entire Pacific ocean. It also varies wildly over time: in the 1970s, the closest weather station to Austin, TX was about 150 km away in San Antonio. Now, there are dozens in Austin alone.

Find the Voronoi Polygon for each Weather Station

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 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

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*

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
```

Refs

- *http://kartoweb.itc.nl/geometrics/Introduction/introduction.html — an* excellent overview of projections, reference surfaces and other fundamentals of geospatial analysis.
- *http://msdn.microsoft.com/en-us/library/bb259689.aspx*
- *http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/*
- *http://wiki.openstreetmap.org/wiki/QuadTiles*
- *https://github.com/simplegeo/polymaps*
- Scaling GIS Data in Non-relational Data Stores by Mike Malone
- Voronoi Diagrams
- US County borders in GeoJSON
- Spatial references, coordinate systems, projections, datums, ellipsoids by Morten Nielsen
- Public repository of geometry boundaries into text

- Making a map in D3 ; see also Guidance on making your own geographic bounaries.

CHAPTER 12

`cat` Herding

CHAPTER 13

Data Munging

CHAPTER 14

Organizing Data

CHAPTER 15

Graphs

CHAPTER 16

Machine Learning

CHAPTER 17

Best Practices

CHAPTER 18

Java Api

CHAPTER 19

Advanced Pig

CHAPTER 20

Hbase Data Modeling

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.

Features you don't get with HBase

Here's a partial list of features you do *not* get in HBase:

- efficient querying or sorting by cell value
- group by, join or secondary indexes
- text indexing or string matching (apart from row-key prefixes)
- arbitrary server-side calculations on query
- any notion of a datatype apart from counters; everything is bytes in/bytes out
- auto-generated serial keys

Sometimes you can partially recreate those features, and often you can accomplish the same *tasks* you'd use those features for, but only with significant constraints or tradeoffs. (You *can* pick up the kids from daycare in a dump truck, but only an idiot picks up their prom date in a dump truck, and in neither case is it the right choice).

More than most engineering tools, it's essential to play to HBase's strengths, and in general the simpler your schema the better HBase will serve you. Somehow, though, the sparsity of its feature set amplifies the siren call of even those few features. Resist, Resist. The more stoicly you treat HBase's small *feature* set, the better you will realize how surprisingly large HBase's *solution* set is.

Schema Design Process: Keep it Stupidly Simple

A good HBase data model is "designed for reads", and your goal is to make *one read per customer request* (or as close as possible). If you do so, HBase will yield response times on the order of 1ms for a cache hit and 10ms for a cache miss, even with billions of rows and millions of columns.

An HBase data mode is typically designed around multiple tables, each serving one or a small number of online queries or batch jobs. There are the questions to ask:

1. What query do you want to make that *must* happen at milliseconds speed?
2. There are a set of related queries or batch jobs — which would you like to be efficient?

If you are using it primarily for batch use,

1. What is the batch job you are most interested in simplifying?
2. There are a set of related queries or batch jobs — which would you like to be efficient?

Autocomplete API (Key-Value lookup)

Let's sketch the implementation of an autocomplete API on Wikipedia page titles, an example that truly plays to HBase's strengths. As a visitor types characters into a search bar, the browser will request a JSON-encoded list of the top 10 most likely completions for that prefix. Responsiveness is essential: at most 50 milliseconds end-to-end response time. Several approaches might spring to mind, like a range query on titles; a prefix query against a text search engine; or a specialized "trie" datastructure. HBase provides a much stupider, far superior solution.

Instead, we'll enumerate every possible completion [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.

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.

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 20-1. 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.

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.

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.

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.

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.

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

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

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

- `A` (XXX million): Political features (states, counties, etc)
- `H` (XXX million): Water-related features (rivers, wells, swamps,…)
- `P` (XXX million): Populated places (city, county seat, capitol, …)
- …
- `R` (): road, railroad, …
- `S` (): Spot, building, farm
- …

Very frequently, we only want one feature type: only cities, or only roads common to want one, several or all at a time.

You could further nest the feature codes. To do a scan of columns in a single get, need to use a ColumnPrefixFilter

http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/ColumnPrefixFilter.html

Access pattern: "Next Interesting Record"

The weatherstation regions table is most interesting of all.

map from weather station to quadkeys, pre-calculated map from observation to quadkeys, accumulate on tile

We want to serve boundaries out in tiles, but records are heavyweight.

if we store whole globe at ZL 14 (2 km blocks), 1kb record size becomes 275 GB data. Multiply by the hours in 50 years (50 * 365.25 * 24 = 438,000 hours = PB.

20,000 weather stations 1 M records = 50x data size; 10 TB becomes 0.5 PB.

```
0111230~~
011123100
011123101
011123102
011123103
01112311~
011123120
011123121
011123122
011123123
01112313~
...
011130~~~
```

Retrieve the *next existing tile*. It's a one-row operation, but we specify a range from specific tile to max tile ID.

The next tile is either the speific one with that key, or the first parent.

Note: next interesting record doesn't use bloom filter

To do a range on zoomed-out, do a range from

want to scan all cells in `011 123`. this means `011 123 000` to `011 123 ~~~`.

Table 20-2. 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".

Table 20-3. Server logs HBase schema

table	row key	family	qualifier	value	options

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

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`	`{prod uct_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 20-4. 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 20-5. 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 20-6. 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 (`"lit`

`eral"`) 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"

Refs

- I've drawn heavily on the wisdom of HBase Book
- Thanks to Lars George for many of these design guidelines, and the "Design for Reads" motto.
- HBase Shell Commands
- HBase Advanced Schema Design by Lars George
- *http://www.quora.com/What-are-the-best-tutorials-on-HBase-schema*
- encoding numbers for lexicographic sorting:
  - an insane but interesting scheme: *http://www.zanopha.com/docs/elen.pdf*
  - a Java library for wire-efficient encoding of many datatypes: *https://github.com/mrflip/orderly*
- *http://www.quora.com/How-are-bloom-filters-used-in-HBase*

CHAPTER 21

Hadoop Internals

CHAPTER 22

Hadoop Tuning

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 22-1. 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 23

Datasets And Scripts

CHAPTER 24
Cheatsheets

Terminal Commands

Table 24-1. 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	

action	command
kill a job	
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`
tuning	csshX, htop, dstat, ulimit
	also useful:
cat, echo, true, false, yes, tee, time, watch, time	dos-to-unix line endings
`ruby -ne 'puts $_.gsub(/\r\n?/, "\n")'`	

Table 24-2. 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 24-3. Regular Expression Cheatsheet

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 24-4 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 24-4. 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

intent	Regular Expression	Comment
[\+\-]\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 24-5. Pig Operator Cheatsheet

action	operator
	JOIN
	FILTER

Hadoop Tunables Cheatsheet

CHAPTER 25

Appendix

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

A sort of colophon

the git-scribe toolchain was very useful creating this book. Instructions on how to install the tool and use it for things like editing this book, submitting errata and providing translations can be found at that site.

- Posse East Bar
- Visuwords.com
- Emacs
- Epoch Coffee House

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
insight shuffles forth
```

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 *