



Big Data for Chimps

O'REILLY®

Philip Kromer

Big Data for Chimps

Philip (flip) Kromer

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo



Big Data for Chimps

by Philip (flip) Kromer

Copyright © 2014 O'Reilly Media. All rights reserved.

Printed in the United States of America.

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

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

Editor: Amy Jollymore and Meghan Blanchette

: First Edition

Revision History for the First Edition:

2014-01-25: Working Draft

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

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Table of Contents

Part I. Mechanics: How Hadoop and Map/Reduce Work

1. Hadoop Basics.....	3
Introduction	4
Map-only Jobs: Process Records Individually	4
Run the Job	7
See Progress and Results	9
Outro	12
2. Map/Reduce.....	13
Introduction	18
Simulation	19
Example: Reindeer Games	19
UFO Sighting Data Model	20
Group the UFO Sightings by Time Bucket	20
The Map-Reduce Haiku	22
Map Phase, in Light Detail	23
Group-Sort Phase, in Light Detail	23
Example: Close Encounters of the Reindeer Kind	26
Put UFO Sightings And Places In Context By Location Name	26
Extend UFO Sighting Records With Location Data	27
Partition, Group and Secondary Sort	28
Partition	28
Group	29
Secondary Sort	29
Playing with Partitions: How Partition, Group and Sort affect a Job	30
Hadoop's Contract	31
The Mapper's Input Guarantee	31
The Reducer's Input Guarantee	32

The Map Phase Processes Records Individually	33
How Hadoop Manages Midstream Data	36
Mappers Spill Data In Sorted Chunks	36
Partitioners Assign Each Record To A Reducer By Label	36
Reducers Receive Sorted Chunks From Mappers	37
Reducers Read Records With A Final Merge/Sort Pass	37
Reducers Write Output Data and Commit	38
A Quick Note on Storage (HDFS)	38
Outro	39
3. Introduction to Pig.....	41
Pig Helps Hadoop work with Tables, not Records	41
Wikipedia Visitor Counts	42
Fundamental Data Operations	43
LOAD Locates and Describes Your Data	46
Simple Types	46
Complex Type 1, Tuples: Fixed-length Sequence of Typed Fields	47
Complex Type 2, Bags: Unbounded Collection of Tuples	48
Complex Type 3, Maps: Collection of Key-Value Pairs for Lookup	48
Defining the Schema of a Transformed Record	49
STORE Writes Data to Disk	49
Development Aids: DESCRIBE, ASSERT, EXPLAIN, LIMIT..DUMP, ILLUSTRATE	50
DESCRIBE shows the schema of a table	50
ASSERT checks that your data is as you think it is	50
DUMP shows data on the console with great peril	51
ILLUSTRATE magically simulates your script's actions, except when it fails to work	51
EXPLAIN shows Pig's execution graph	52
Pig Functions act on fields	52
Moving right along ...	54

Part II. Tactics: Analytic Patterns

4. Analytic Patterns part 1: Pipeline Operations.....	59
Eliminating Data	59
Selecting Records that Satisfy a Condition: FILTER and Friends	60
Selecting Records that Satisfy Multiple Conditions	61
Selecting or Rejecting Records with a null Value	61
Selecting Records that Match a Regular Expression (MATCHES)	63
Matching Records against a Fixed List of Lookup Values	66
Project Only Chosen Columns by Name	67

Using a FOREACH to Select, Rename and Reorder fields	67
Extracting a Random Sample of Records	68
Extracting a Consistent Sample of Records by Key	69
Sampling Carelessly by Only Loading Some part- Files	70
Selecting a Fixed Number of Records with LIMIT	71
Other Data Elimination Patterns	71
Transforming Records	72
Transform Records Individually using FOREACH	72
A nested FOREACH Allows Intermediate Expressions	73
Formatting a String According to a Template	74
Assembling Literals with Complex Types	76
Specifying Schema for Complex Types	79
Manipulating the Type of a Field	80
Ints and Floats and Rounding, Oh My!	81
Calling a User-Defined Function (UDF) from an External Package	82
Operations that Break One Table Into Many	87
Directing Data Conditionally into Multiple Data Flows (SPLIT)	87
Splitting into files by key by using a Pig Storefunc UDF	88
Splitting a Table into Uniform Chunks	90
Operations that Treat the Union of Several Tables as One	91
Treat Several Pig Relation Tables as a Single Table (Stacking Rowsets)	91
Clean Up Many Small Files by Merging into Fewer Files	92
5. Analytic Patterns: Grouping Operations	95
Introduction	95
Grouping Records into a Bag by Key	95
Counting Occurrences of a Key	97
Representing a Collection of Values with a Delimited String	98
Representing a Complex Data Structure with a Delimited String	99
Representing a Complex Data Structure with a JSON-encoded String	101
Group and Aggregate	104
Aggregate Statistics of a Group	104
Completely Summarizing a Field	105
Summarizing Aggregate Statistics of a Full Table	107
Summarizing a String Field	109
Calculating the Distribution of Numeric Values with a Histogram	110
Binning Data for a Histogram	111
Choosing a Bin Size	112
Interpreting Histograms and Quantiles	113
Binning Data into Exponentially Sized Buckets	114
Creating Pig Macros for Common Stanzas	115
Distribution of Games Played	116

Extreme Populations and Confounding Factors	116
Don't Trust Distributions at the Tails	120
Calculating a Relative Distribution Histogram	121
Re-injecting Global Values	121
Calculating a Histogram Within a Group	122
Dumping Readable Results	124
The Summing Trick	127
Counting Conditional Subsets of a Group — The Summing Trick	127
Summarizing Multiple Subsets of a Group Simultaneously	128
Testing for Absence of a Value Within a Group	130
Refs	132
6. Analytic Patterns: Joining Tables.....	133
Matching Records Between Tables (Inner Join)	133
Joining Records in a Table with Directly Matching Records from Another Table (Direct Inner Join)	133
A Join is a Map/Reduce Job with a secondary sort on the Table Name	135
Handling Nulls and Non-matches in Joins and Groups	137
Enumerating a Many-to-Many Relationship	139
Joining a Table with Itself (self-join)	140
Joining Records Without Discarding Non-Matches (Outer Join)	141
Joining Tables that do not have a Foreign-Key Relationship	143
Joining on an Integer Table to Fill Holes in a List	145
Selecting Only Records That Lack a Match in Another Table (anti-join)	146
Selecting Only Records That Possess a Match in Another Table (semi-join)	147
An Alternative to Anti-Join: use a COGROUP	147
7. Analytic Patterns: Ordering Operations.....	149
Sorting All Records in Total Order	149
Sorting by Multiple Fields	149
Sorting on an Expression (You Can't)	150
Sorting Case-insensitive Strings	150
Dealing with Nulls When Sorting	150
Floating Values to the Top or Bottom of the Sort Order	151
Sorting Records within a Group	151
Select Rows with the Top-K Values for a Field	154
Finding Records Associated with Maximum Values	154
Top K Records within a table using ORDER..LIMIT	155
Shuffle a set of records	155
8. Analytic Patterns: Finding Duplicate and Unique Records.....	157
Handling Duplicates	157

Eliminating Duplicate Records from a Table	157
Eliminating Duplicate Records from a Group	158
Eliminating All But One Duplicate Based on a Key	158
Selecting Records with Unique (or with Duplicate) Values for a Key	159
Set Operations	159
Set Operations on Full Tables	161
Distinct Union	161
Distinct Union (alternative method)	161
Set Intersection	162
Set Difference	162
Symmetric Set Difference: $(A-B)+(B-A)$	162
Set Equality	163
Constructing a Sequence of Sets	164
Set Operations Within a Group	164

Part III. Strategy: Practical Data Explorations

9. Spatial and Geographic Analysis.....	169
Matching Points within a Given Distance (Pointwise Spatial Join)	175
Distance on a Sphere	176
Spatial Aggregations on Regions	182
Quadtree Decomposition and Numbering	186
Exporting data for Presentation by a Tileserver	187
Mechanics of Spatial Data	188
Spatial Data Types	189
Spatial Operations in Pig	190
Shape Transforming Operations	191
Constructing and Converting Geometry Objects	192
Properties of Shapes	192
Operations that Combine Shapes	193
Testing the Relationship of Two Shapes	194
Spatial Join with Regions	196
How the Merge/Sweep Join Works	196
Sweep Stack	197
Turning Points of Measurements Into Regions of Influence	199
Weather Near You	200
Find the Voronoi Polygon for each Weather Station	200
The Quadtile Grid System	201
Quadtile Practicalities	202
Converting points to quadkeys (quadtile indexes)	202
Interesting quadtile properties	205

Quadtile Ready Reference	207
Data Formats	207
GeoJSON	208
Well-Known Text + Tab-Separated Values	210
Well-Known Binary (WKB)	211
Other Important Formats	211
Converting Among Geospatial Data Formats	212

PART I

Mechanics: How Hadoop and Map/Reduce Work

CHAPTER 1

Hadoop Basics

Chimpanzee and Elephant Start a Business

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

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

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

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

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

The fact that each chimpanzee's work is independent of any other's — no interoffice memos, no meetings, no requests for documents from other departments — made this the perfect first contract for the Chimpanzee & Elephant, Inc. crew. JT and Nanette, however, were cooking up a new way to put their million-chimp army to work, one that could radically streamline the processes of any modern paperful office¹. JT and Nanette would soon have the chance of a lifetime to try it out for a customer in the far north with a big, big problem.

Introduction

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

Map-only Jobs: Process Records Individually

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

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

1. Some chimpanzee philosophers have put forth the fanciful conceit of a “paper-less” office, requiring impossibilities like a sea of electrons that do the work of a chimpanzee, and disks of magnetized iron that would serve as scrolls. These ideas are, of course, pure lunacy!
2. Sharp-eyed readers will note that this language is really called *Pig Latin*. That term has another name in the Hadoop universe, though, so we've chosen to call it Igpay Atinlay — Pig Latin for “Pig Latin”.

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

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

Igpay Atinlay translator, actual version.

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

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

Igpay Atinlay translator, pseudocode.

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

Ruby helper

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

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

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

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

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

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



Are you running on a cluster?

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

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

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

Run the following commands to copy your data to HDFS:

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

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

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

Run the Job

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

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

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

The Jobtracker Console

When you are running on a distributed Hadoop cluster, the jobtracker offers a built-in console for monitoring and diagnosing jobs. You typically access it by pointing your web browser at `http://hostname.of.jobtracker:50030` (replace “hostname.of.jobtracker” with, you know, the hostname of your jobtracker and if that jobtracker is running on your local machine, you can use `http://hostname.of.jobtracker:50030``). At the top, the

cluster summary shows how many jobs are running and how many worker slots exist. Clicking on the hyperlinked value under “nodes” will take you to a screen summarizing all the task trackers in the cluster. Keep an eye out for other such hyperlinked values within the jobtracker console — they lead to screens describing those elements in details.

Further down the page, you will see sections for running, completed and failed jobs. The last part of each job’s name is an index showing the order it was received and appears in the flurry of messages when you launched your job.

Clicking on the job name will take you to a page summarizing that job. We will walk through the page from the bottom up because that is how to best understand the job’s progress. The very bottom of the page contains a fun pair of bar charts showing the progress of each Map and Reduce task, from zero to 100-percent complete. A healthy job with many tasks should look like the screenshots below (CODE: Screenshot). During the Map phase, you should see a rolling wave of bars: completed tasks at 100 percent on the left, pending tasks at zero percent on the right and a waveform of running tasks that smoothly advance from zero to 100 percent at a fairly uniform pace. During the Reduce phase, you should see the full set of bars advanced at nearly the same rate up the page. Each bar has three sections (we will learn later more about what they mean but it is enough for now to know how they should behave).

You should observe slow progress through the shuffle stage beginning part way through the Map phase of the job and steady progress at a slightly higher pace as soon as the Map phase concludes. Unless you are heavily burdening your Reducers, the graph should walk right through the Sort stage and begin making steady uniform progress through the final Reduce step. (CODE: Check that shuffle progress is displayed as non-0 during Map phase).

The job is not completely finished when the last Reducer hits 100 percent — there remains a Commit phase with minor bookkeeping chores, and replication of the output data — but the delay from end of Reduce to successful job completion should be small.

The main thing to watch for in the Reduce phase is rapid progress by most of your Reducers and painfully slow progress by a few of them — the “skewed reducer” problem. Because of either a simple mistake on your part or a deep challenge resulting from the structure of your data, Hadoop has sent nearly all the records to those few machines. Those simple mistakes are described in Chapter (REF); defense against highly-skewed data is, in a sense, the motivation for most of the methods presented in the middle section of the book.

Do not put too much faith in the “percent complete” numbers for the job as a whole and even for its individual tasks. These really only show the fraction of data processed, which is an imperfect indicator of progress and harder to determine than you might think. Among other pecadillos, some compressed input formats report no progress mid-task; they linger at zero then go straight to 100 percent.

Above the job progress bar graphs is a hot mess of a table showing all sorts of metrics about your job, such as how much data read and written at each phase. We will run down the important ones a bit later in the book (REF).

Above that section is a smaller table giving the count of pending, running, complete, killed and failed jobs. Most of the values in that table are hyperlinks that begin the annoyingly long trip required to see your logs. Clicking on the completed tasks number takes you to a screen listing all the tasks; clicking on a task ID takes you to a screen listing the machine or machines running it. Clicking on the attempt ID shows a page describing that machine's progress through the task; and all the way on the right side of the table on that page, you will find three sets of links reading "4KB," "8KB," "All." (TECH: check details). Those links lead, at long last, to the log files for that job on that machine. The "All" link shows you the full contents but if your job is so screwed up that the log file will flood your browser, the "8KB" link shows the truncated tale.

Lastly, near the top of a page is a long URL that ends with "job.xml". Do not go there now, but keep it in mind for when you have run out of ideas as to why a job is failing: it is a monstrous but useful file listing every single configuration value set by your job.

You can compare its output to the earlier by running

```
hadoop fs -cat ./output/latinized_magis/*
```

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

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

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

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

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

See Progress and Results

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

So let's run it on a corpus large enough to show off the power of distributed computing. Shakespeare's combined works are too small — at (CODE find size) even the prolific

bard's lifetime of work won't make Hadoop break a sweat. Luckily, we've had a good slice of humanity typing thoughts into wikipedia for several years, and the corpus containing every single wikipedia article is enough to warrant Hadoop's power (and tsoris³).

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

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

The most important numbers to note are the number of running tasks (there should be some unless your job is finished or the cluster is congested) and the number of failed tasks (for a healthy job on a healthy cluster, there should never be any). Don't worry about killed tasks; for reasons we'll explain later on, it's OK if a few appear late in a job. We will describe what to do when there are failing attempts later in the section on debugging Hadoop jobs (REF), but in this case, there shouldn't be any. Clicking on the number of running Map tasks will take you to a window that lists all running attempts (and similarly for the other categories). On the completed tasks listing, note how long each attempt took; for the Amazon M3.xlarge machines we used, each attempt took about x seconds (CODE: correct time and machine size). There is a lot of information here, so we will pick this back up in chapter (REF), but the most important indicator is that your attempts complete in a uniform and reasonable length of time. There could be good justification for why task 00001 is still running after five minutes while all other attempts finished in ten seconds, but if that's not what you thought would happen you should dig deeper⁴.

You should get in the habit of sanity-checking the number of tasks and the input and output sizes at each job phase for the jobs you write. In this case, the job should ultimately require x Map tasks, no Reduce tasks and on our x machine cluster, it completed in x minutes. For this input, there should be one Map task per HDFS block, x GB of input with the typical one-eighth GB block size, means there should be 8x Map tasks. Sanity checking the figure will help you flag cases where you ran on all the data rather than the one little slice you intended or vice versa; to cases where the data is organized inefficiently; or to deeper reasons that will require you to flip ahead to chapter (REF).

3. trouble and suffering

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

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

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

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

How a job is born, the thumbnail version

Apart from one important detail, the mechanics of how a job is born should never become interesting to a Hadoop user. But since some people’s brains won’t really believe that the thing actually works unless we dispel some of the mystery, here’s a brief synopsis.

When you run `wukong ...` or `pig ...` or otherwise launch a Hadoop job, your local program contacts the jobtracker to transfer information about the job and the Java `.jar` file each worker should execute. If the input comes from an HDFS, the jobtracker (TECH: ?job client?) consults its namenode for details about the input blocks, figures out a job ID and any other remaining configuration settings, and accepts the job. It replies to your job client with the job ID and other information, leading to the happy mess of log messages your program emits. (TECH: check the details on this) Your local program continues to run during the full course of the job so that you can see its progress, but is now irrelevant — logging out or killing the local program has no impact on the job’s success.

As you have gathered, each Hadoop worker runs a tasktracker daemon to coordinate the tasks run by that machine. Like JT’s chimpanzees, those tasktrackers periodically report progress to the jobtracker, requesting new work whenever there is an idle slot. The jobtracker never makes outward contact with a task tracker — this ensures work is only distributed to healthy machines at a rate they can handle. Just as JT strives to ensure that chimpanzees are only assigned passages held by their cubicle mates, the jobtracker schedules strives to assign each map attempt to a machine that holds its input blocks (known as “mapper-local” task). But if too many blocks of a file hotspot on a small

number of datanodes, mapper slots with no remaining mapper-local blocks to handle still receive task attempts, and simply pull in their input data over the network.

The one important detail to learn in all this is that *task trackers do not run your job, they only launch it*. Your job executes in a completely independent child process with its own Java settings and library dependencies. In fact, if you are using Hadoop streaming programs like Wukong, your job runs in even yet its own process, spawned by the Java child process. We've seen people increase the tasktracker memory sizes thinking it will improve cluster performance — the only impact of doing so is to increase the likelihood of out-of-memory errors.

Outro

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

CHAPTER 2

Map/Reduce

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

Trouble in Toyland

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

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

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

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

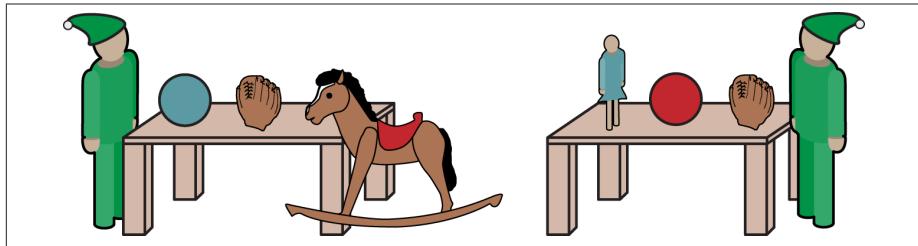


Figure 2-1. The elves' workbenches are meticulous and neat.

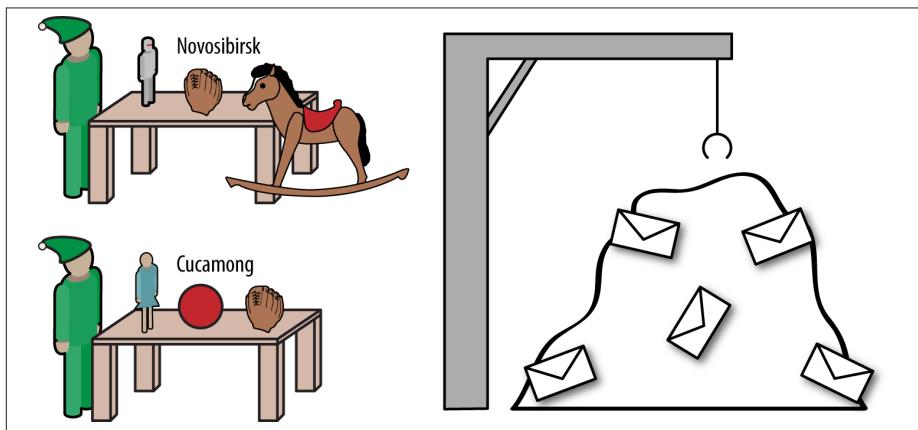


Figure 2-2. Little boys and girls' mail is less so.

Chimpanzees Process Letters into Labelled Toy Forms

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

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

The chimps' job was to take letters one after another from a mailbag, and fill out a toyform for each request. A toyform has a prominent label showing the type of toy, and

a body with all the information you'd expect: Name, Nice/Naughty Status, Location, and so forth. You can see some examples here:

Deer SANTA

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

I have been good this year

love julia

```
# Good kids, generates a toy for Julia and a toy for her brother  
# Toy Forms:  
#   doll | type="green hair" recipient="Joe's sister Julia"  
#   robot | type="optimus prime" recipient="Joe"
```

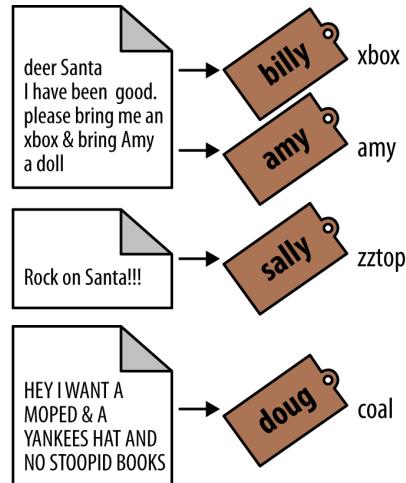
Greetings to you Mr Claus, I came to know of you in my search for a reliable and reputable person to handle a very confidential business transaction, which involves the transfer of a large sum of money...

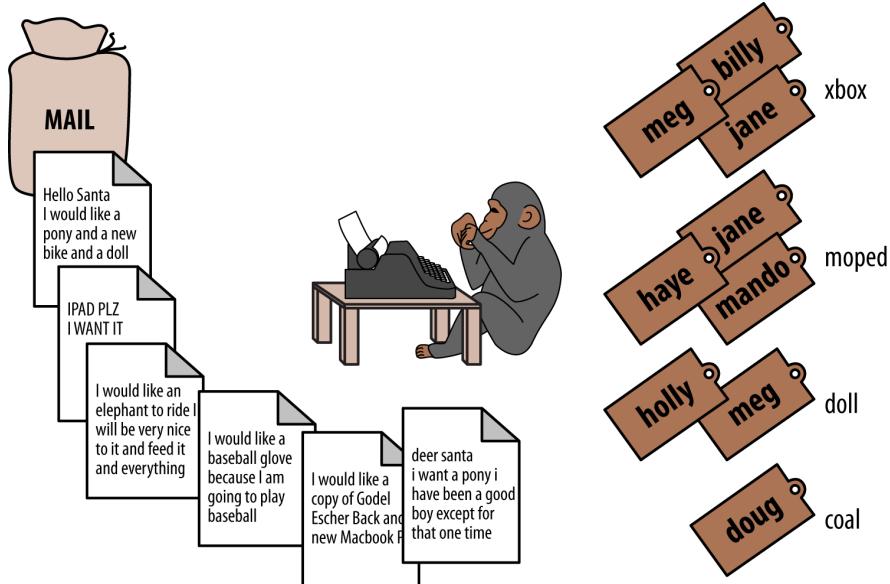
```
# Spam  
# (no toy forms)
```

HEY SANTA I WANT A YANKEES HAT AND NOT
ANY DUMB BOOKS THIS YEAR

FRANK

```
# Frank is a jerk. He will get a lump of coal.  
# Toy Forms:  
#   coal | type="anthracite" recipient="Frank" reason="doesn't like to read"
```



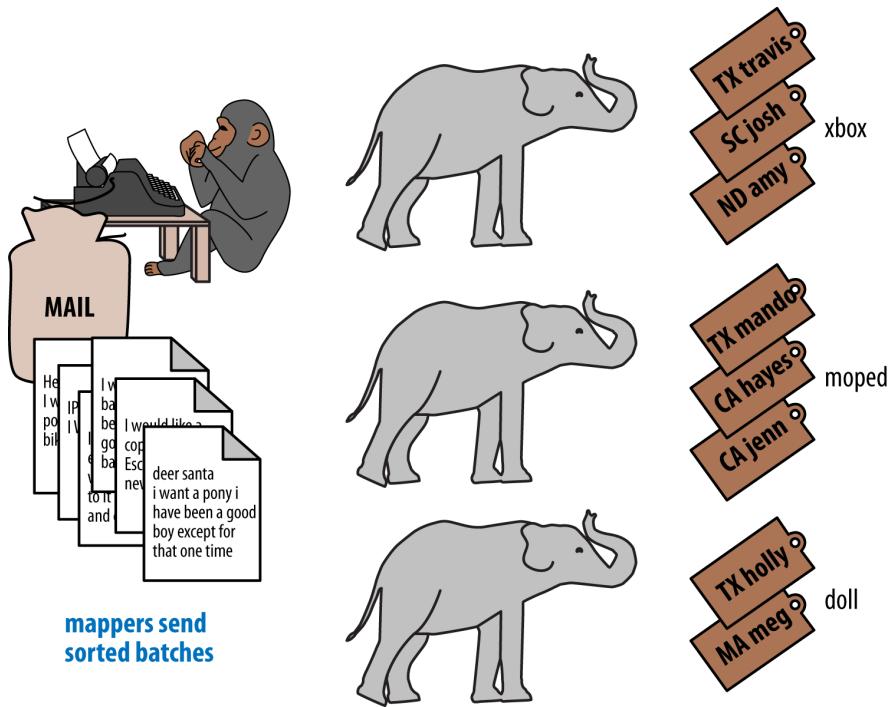


The first note, from a very good girl who is thoughtful for her brother, creates two toyforms: one for Joe's robot and one for Julia's doll. The second note is spam, so it creates no toyforms. The third one yields a toyform directing Santa to put coal in Frank's stocking.

Pygmy Elephants Carry Each Toyform to the Appropriate Workbench

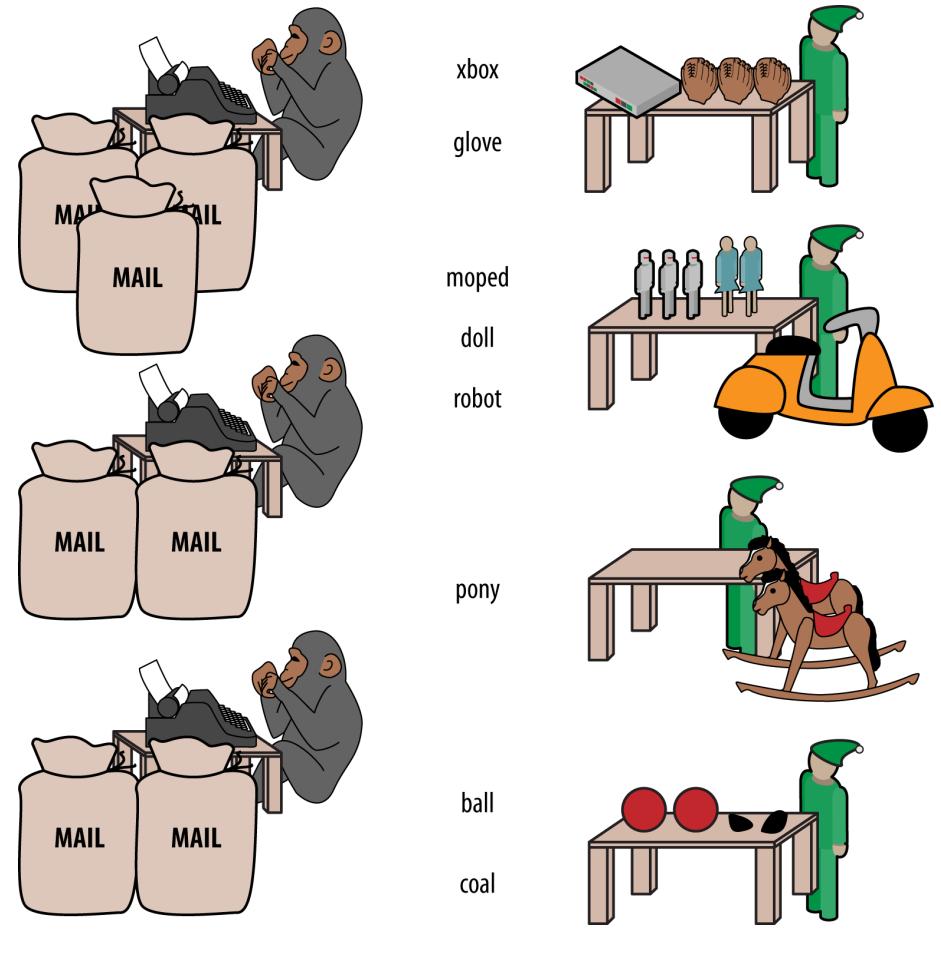
Here's the new wrinkle on top of the system used in the translation project. Next to every desk now stood a line of pygmy elephants, each dressed in a capes that listed the types of toy it would deliver. Each desk had a pygmy elephant for Archery Kits and Dolls, another one for Xylophones and Yo-Yos,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

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



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

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



Introduction

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

Hadoop’s real power comes from the ability to process data in context, using what’s known as the Map/Reduce paradigm. Every map/reduce job is a program with the same three phases: map, group-sort, and reduce. In the map phase, your program processes its input in any way you see fit, emitting labelled output records. In the group-sort phase, Hadoop groups and sorts those records according to their labels. Finally, in the reduce

phase, your program processes each sorted group and Hadoop stores its output. That grouping-by-label part is where the magic lies: it ensures that no matter where the relevant records started, they arrive at the same place in a predictable manner, ready to be synthesized.

If Map/Reduce is the core of Hadoop's operation, then getting to *think* in Map/Reduce terms is the key to effectively using Hadoop. In turn, thinking in Map/Reduce requires that you develop an innate, physical sense of how Hadoop moves data around. You can't understand the fundamental patterns of data analysis in Hadoop — grouping, filtering, joining records, and so forth — without knowing the basics.¹

It can take some time to wrap one's head around Map/Reduce, though, so we're going to move slowly in this chapter. We'll open with a straightforward example map/reduce program: aggregating records from a dataset of Unidentified Flying Object sightings to find out when UFOs are most likely to appear. Next, we'll revisit our friends at Elephant and Chimpanzee, Inc, to explore how a Map/Reduce dataflow works. We'll round out the chapter with a deeper, more technical look into map/reduce from a Hadoop perspective.

The one thing we won't be doing too much of yet is actually writing lots of Hadoop programs. That will come in Chapter 5 (REF), which has example after example demonstrating core map/reduce programming patterns — those patterns are difficult to master without a grounding in this chapter's material. But if you're the type of reader who learns best by seeing multiple examples in practice and then seeing its internal mechanics, skim that chapter and then come back.

Simulation

Santa Corp does not want any future logistical surprises, and so along with their new streamlined manufacturing workflow they would like to perform scenario planning. Ms Claus, the CIO of Santa Corp, has heard about this new “map/reduce”

Example: Reindeer Games

Santa Claus and his elves are busy year-round, but outside the holiday season Santa's flying reindeer do not have many responsibilities. As flying objects themselves, they spend a good part of their multi-month break pursuing their favorite hobby: UFOlogy (the study of Unidentified Flying Objects and the search for extraterrestrial civilization). So you can imagine how excited they were to learn about the data set of more than 60,000 documented UFO sightings we worked with in the first chapter.

1. When he lectures on Hadoop, Q often gets questions to the effect of, “Can I do X in Hadoop?” and the answer is always, “If you can express that problem or algorithm in Map/Reduce terms, then, yes.”

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

UFO Sighting Data Model

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

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

Group the UFO Sightings by Time Bucket

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

Mapper

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

1. read and understand each letter
2. create a new intermediate item having a label (the type of toy) and information about the toy (the work order)
3. hand it to the elephant which delivers to that toy's workbench

We're going to write a Hadoop *mapper* which performs a similar purpose:

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

The program looks like this:

```

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

```

You can test the mapper on the commandline:

```

$ cat ./data/geo/ufo_sightings/ufo_sightings-sample.json | 
./examples/geo/ufo_sightings/count_ufo_shapes.rb --map | 
head -n25 | wu-lign
disk      1972-06-16T05:00:00Z 1999-03-02T06:00:00Z  Provo (south of), UT      disk      severa
sphere    1999-03-02T06:00:00Z 1999-03-02T06:00:00Z  Dallas, TX      sphere    60 sec
triangle   1997-07-03T05:00:00Z 1999-03-09T06:00:00Z  Bochum (Germany),      triangle  ca. 2m
light      1998-11-19T06:00:00Z 1998-11-19T06:00:00Z  Phoenix (west valley), AZ light    15min
triangle   1999-02-27T06:00:00Z 1999-02-27T06:00:00Z  San Diego, CA      triangle  10 min
triangle   1997-09-15T05:00:00Z 1999-02-17T06:00:00Z  Wedgefield, SC      triangle  15 min
...

```

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

Reducer

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

```

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

```

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

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

Test the full map/reduce stack from the commandline:

```
$ ./examples/geo/ufo_sightings/count_ufo_shapes.rb --run \
  ./data/geo/ufo_sightings/ufo_sightings-sample.json - | wu-lign

1985-06-01T05:00:00Z    1999-01-14T06:00:00Z    North Tonawanda, NY chevron 1 hr      7 lights i
1999-01-20T06:00:00Z    1999-01-31T06:00:00Z    Olney, IL      chevron 10 seconds Stargazing
1998-12-16T06:00:00Z    1998-12-16T06:00:00Z    Lubbock, TX      chevron 3 minutes Object sou
   %%% end of group %%% ct: 3 chevron
1999-01-16T06:00:00Z    1999-01-16T06:00:00Z    Deptford, NJ      cigar 2 Hours An aircraf
   %%% end of group %%% ct: 1 cigar
1947-10-15T06:00:00Z    1999-02-25T06:00:00Z    Palmira,          circle 1 hour After a co
1999-01-10T06:00:00Z    1999-01-11T06:00:00Z    Tyson's Corner, VA circle 1 to 2 sec Bright gre
...
...
```

Plot the Data

When people work with data, their end goal is to uncover some answer or pattern. They most often employ Hadoop to turn Big Data into small data, then use traditional analytics techniques to turn small data into insight. One such technique is to *plot* the information. If a picture is worth a thousand words, then even a basic data plot is worth reams of statistical analysis. (TODO-qem: I think that line is original, but it sounds familiar. Must check around to make sure I didn't just pinch someone's quote.) That's because the human eye often gets a rough idea of a pattern faster than people can write code to divine the proper mathematical result. Here, we've used the free, open-source **R programming language** to see how UFO sightings are distributed around the country.

²

The Map-Reduce Haiku

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

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

— The Map/Reduce Haiku

More prosaically,

description	phase	explanation
process and label	map	turn each input record into any number of labelled records

- That said, people sometimes want to run R *inside* Hadoop, to analyze large-scale datasets. If you're interested in using R and Hadoop together, please check out Q's other book, *Parallel R* (O'Reilly) <http://shop.oreilly.com/product/0636920021421.do>

description	phase	explanation
sorted context groups	group-sort	Hadoop groups those records uniquely under each label, in a sorted order. (You'll see this also called the shuffle/sort phase)
synthesize (process context groups)	reduce	for each group, process its records in order; emit anything you want.

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

The records in stage 1 (*label*) are out of context. The mappers see each record exactly once, but with no promises as to order, and no promises as to which mapper sees which record. We've *moved the compute to the data*, allowing each process to work quietly on the data in its work space. Over at C&E, letters and translation passages aren't pre-organized and they don't have to be; J.T. and Nanette care about keeping all the chimps working steadily and keeping the hallways clear of inter-office document requests.

Once the map attempt finishes, each *partition* (the collection of records destined for a common reducer) is dispatched to the corresponding machine, and the mapper is free to start a new task. 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.

Map Phase, in Light Detail

Digging a little deeper into the mechanics of it all, a mapper receives one record at a time. By default, Hadoop works on text files, and a record is one line of text. (Hadoop supports other file formats and other types of storage beside files, but for the most part the examples in this book will focus on processing files on disk in a readable text format.) The whole point of the mapper is to "label" the record so that the group-sort phase can track records with the same label.

Hadoop feeds the mapper that one record, and in turn, the mapper spits out one or more *labelled records*. Usually the values in each record fields are some combination of the values in the input record and simple transformation of those values. But the output is allowed to be anything — the entire record, some subset of fields, the phase of the moon, the contents of a web page, nothing, ... — and at times we'll solve important problems by pushing that point. The mapper can output those records in any order, at any time in its lifecycle, each with any label.

Group-Sort Phase, in Light Detail

In the group-sort phase, Hadoop transfers all the map output records in a partition to the corresponding reducer. That reducer merges the records it receives from all mappers, so that each group contains all records for its label regardless of what machine it

came from. What's nice about the group-sort phase is that you don't have to do anything for it. Hadoop takes care of moving the data around for you. What's less nice about the group-sort phase is that it is typically the performance bottleneck. We'll learn how to take care of Hadoop so that it can move the data around smartly.

Code
==== Reducers, in Light Detail

Whereas the mapper sees single records in isolation, a reducer receives one key (the label) and *all* records that match that key. In other words, a reducer operates on a group of related records. Just as with the mapper, as long as it keeps eating records and doesn't fail the reducer can do anything with those records it pleases and emit anything it wants. It can nothing, it can contact a remote database, it can emit nothing until the very end and then emit one or a zillion records. The output can be text, it can be video files, it can be angry letters to the President. They don't have to be labelled, and they don't have to make sense. Having said all that, usually what a reducer emits are nice well-formed records resulting from sensible transformations of its input, like the count of records, the largest or smallest value from a field, or full records paired with other records. And though there's no explicit notion of a label attached to a reducer output record, it's pretty common that within the record's fields are values that future mappers will use to form labels.

Once you understand the label-group-process data flow we've just introduced, you understand enough about map/reduce to reason about the large-scale motion of data and thus your job's performance. But to understand how we can extend this one simple primitive to encompass the whole range of data analysis operations, we need to attach more nuance to the intermediate phase, and the importance of sorting to Hadoop's internal operation.

Elephant and Chimpanzee Save Christmas part 2: A Critical Bottleneck Emerges

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

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

To clear his mind, JT wandered over to the reindeer ready room, eager to join in the cutthroat games of poker Rudolph and his pals regularly ran. During a break in the action, JT found himself idly sorting out the deck of cards by number, to check that none of his Reindeer friends slipped an extra ace or three into the deck. As he did so, something in his mind flashed back to the unfinished toys on the assembly floor: mounds of number blocks, stacks of Jack-in-the-boxes, rows of dolls. Sorting the cards by number had naturally organized them into groups by kind as well: he saw all the numbers in blocks in a run, followed by all the jacks, then the queens and the kings and the aces.

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

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

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



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

The elves were now producing a steady stream of toys as fast as their hammers could fly, with an economy of motion they’d never experienced. Since both the parts and the

toy forms were in the same order by toyform ID, as the toymakers would pull the next toy form from the file they would always find the parts for it first at hand. Get the toy form for a wooden toy train and you would find a train chassis next in the chassis cart, small wooden wheels next in the wheel cart, and magnetic bumpers next in the small parts cart. Get the toy form for a rolling duck on a string, and you would find instead, a duck chassis, large wooden wheels and a length of string at the head of their respective carts.

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

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

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

Example: Close Encounters of the Reindeer Kind

In the last problem we solved for our Reindeer friends, we only cared that the data came to the reducer in groups. We had no concerns about which reducers handled which groups, and we had no concerns about how the data was organized within the group. The next example will draw on the full scope of the framework, equipping you to understand the complete contract that Hadoop provides the end user.

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

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

Put UFO Sightings And Places In Context By Location Name

When you are writing a Map/Reduce job, the first critical question is how to group the records in context for the Reducer to synthesize. In this case, we want to match every UFO sighting against the corresponding Geonames record with the same city, state and country, so the Mapper labels each record with those three fields. This ensures records

with the same location name all are received by a single Reducer in a single group, just as we saw with toys sent to the same workbench or visits “sent” to the same time bucket. The Reducer will also need to know which records are sightings and which records are places, so we have extended the label with an “A” for places and a “B” for sightings. (You will see in a moment why we chose those letters.) While we are at it, we will also eliminate Geonames records that are not populated places.

```
class UfoSighting
  include Wu::Model
  field :sighted_at,    Time
  field :reported_at,   Time
  field :shape,          Symbol
  field :city,           String
  field :state,          String
  field :country,        String
  field :duration_str,  String
  field :location_str,  String
  #
  field :longitude,     Float
  field :latitude,       Float
  field :city,           String
  field :region,         String
  field :country,        String
  field :population,    Integer
  field :quadkey,        String
  #
  field :description,   String
end
```

Extend UFO Sighting Records With Location Data

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

Now that you’ve seen the partition, sort and secondary sort in action, it’s time to attach more formal and technical detail to how it works.

Partition, Group and Secondary Sort

As we mentioned in the opening, the fundamental challenge of Big Data is how to put records into relevant context, even when it is distributed in a highly non-local fashion. Traditional databases and high-performance computing approaches use a diverse set of methods and high-cost hardware to brute-force the problem but at some point, the joint laws of physics and economics win out. Hadoop, instead, gives you exactly and *only one* “locality” primitive — only one way to express which records should be grouped in context — namely, *partition-group-sort* -ing the records by their label. The sidebar (REF) about the Hadoop contract describes the precise properties of this operation but here is a less formal explanation of its essential behavior.

Partition

The partition key portion of the label governs how records are assigned to Reducers; it is analogous to the tear-sheet that mapped which toy types went to which workbench. Just as there was only one workbench for dolls and one workbench for ponies, each partition maps to *exactly one* Reducer. Since there are generally a small number of Reducers and an arbitrary number of partitions, each Reducer will typically see many partitions.

The default partitioner (`HashPartitioner`) assigns partitions to Reducers arbitrarily, in order to give a reasonably uniform distribution of records to Reducers. It does not know anything specific about your data, though, so you could get unlucky and find that you have sent all the tweets by Justin Bieber and Lady Gaga to the same Reducer or all the census forms for New York, L.A. and Chicago to the same Reducer, leaving it with an unfairly large portion of the midstream data. If the partitions themselves would be manageable and you are simply unlucky as to which became neighbors, just try using one fewer Reduce slots — this will break up the mapping into a different set of neighbors.

For a given cluster with a given number of Reduce slots, the assignment of partitions by the hash Reducer will be stable from run to run, but you should not count on it any more than that.

The naive `HashPartitioner` would not work for the elves, we assume — you don’t want the toyforms for ponies to be handled by the same workbench processing toyforms for pocketwatches. For us too, some operations require a specific partitioning scheme (as you will see when we describe the total sort operation (REF)), and so Hadoop allows you to specify your own partitioner. But this is rarely necessary, and in fact your authors have gone their whole careers without ever writing one. If you find yourself considering writing a custom partitioner, stop to consider whether you are going against the grain of Hadoop’s framework. Hadoop knows what to do with your data and, typically, the fewer constraints you place on its operation, the better it can serve you.

Group

The group key governs, well, the actual groups your program sees. All the records within a group arrive together — once you see a record from one group, you will see all of them in a row and you will never again see a record from a preceding group.

Secondary Sort

Within the group, the records are sent in the order given by the sort key. When you are using the Hadoop streaming interface (the basis for Wukong, MrJobs and the like), the only datatype is text, and so records are sorted lexicographically by their UTF-8 characters. (TECHREVIEW: is it UTF-8 or binary strings?)

This means that:

- Zoo comes after Apple, because A comes before Z
- Zoo comes *before* apple, because upper-case characters precede lower-case characters
- 12345 comes before 42, and both of them come before Apple, Zoo or apple
- 12345 comes after ` 42` because we used spaces to pad out the number 42 to five characters.
- apple and zoo come before шимпанзе, because the basic ASCII-like characters (like the ones on a US keyboard) precede extended unicode-like characters (like the russian characters in the word for “chimpanzee”).
- ### (hash marks) come before Apple and zoo; and |||| (pipes) comes after all of them. Remember these characters — they are useful for forcing a set of records to the top or bottom of your input, a trick we'll use in the geodata chapter (REF). The dot (.), hyphen (-), plus (+) hash (#) come near the start of the 7-bit ASCII alphanumeric set. The tilde (~), pipe ()| come at the end. All of them precede extended-character words like шимпанзе.



It's very important to recognize that *numbers are not sorted by their numeric value unless you have control over their Java type*. The simplest way to get numeric sorting of positive numbers is to pad numeric outputs a constant width by prepended spaces. In Ruby, the expression `%10d" % val` produces an ten-character wide string (wide enough for all positive thirty-two bit numbers). There's no good way in basic Hadoop Streaming to get negative numbers to sort properly — yes, this is very annoying. (TECHREVIEW: is there a good way?)

In the common case, the partition key, group key and sort key are the same, because all you care is that records are grouped. But of course it's also common to have the three keys not be the same. The prior example, (REF) a JOIN of two tables, demonstrated a common pattern for use of the secondary sort; and the roll-up aggregation example that follows illustrates both a secondary sort and a larger partition key than group key.

The set defined by the partition key must be identical or a superset of the sets defined by the group key, or your groups will be meaningless. Hadoop doesn't impose that constraint on you, so just be sure to think at least once. The easiest way to do this (and the way we almost always to this) is to have the partition key be the same as or an extension of the group key, and the sort key be the same as or an extension of the group key.

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

It is very important to get a good grasp of how the partition and group keys relate, so let's step through an exercise illustrating their influence on the distribution of records.

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

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

Running it instead with `--partition_keys=2 --sort_keys=3` tells Hadoop to *partition* on the century/year, but do a secondary sort on the month as well. All records that share a century and year now go to the same reducer, while the reducers still see months as continuous chunks. Now there are only six (or fewer) reducers that receive data — all of 2008 goes to one reducer, similarly 2009, 2010, and the rest of the years in the dataset. In our runs, we saw years X and Y (CODE adjust reducer count to let us prove

the point, insert numbers) land on the same reducer. This uneven distribution of data across the reducers should cause the job to take slightly longer than the first run. To push that point even farther, running with `--partition_keys=1 --sort_keys=3` now partitions on the century — which all the records share. You'll now see 19 reducers finish promptly following the last mapper, and the job should take nearly twenty times as long as with `--partition_keys=3`.

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

Hadoop's Contract

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

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

The Mapper's Input Guarantee

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

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

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

The Reducer's Input Guarantee

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

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

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

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

You should typically think in terms of groups and not about the whole reduce set: imagine each partition is sent to its own reducer. It's important to know, however, that each reducer typically sees multiple partitions. (Since it's more efficient to process large batches, a certain number of reducer processes are started on each machine. This is in contrast to the mappers, who run one task per input split.) Unless you take special measures, the partitions are distributed arbitrarily among the reducers³. They are fed to the reducer in order by key.

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

3. Using a “consistent hash”; see (REF) the chapter on Statistics



The traditional terms for the Hadoop phases are very unfortunately chosen. The name “map” isn’t that bad, though it sure gets confusing when you’re using a `HashMap` in the map phase of a job that maps locations to coordinates for a mapping application. Things get worse after that, though. Hadoop identifies two phases, called shuffle and sort, between the map and reduce. That division is irrelevant to you, the end user, and not even that essential internally. “Shuffling” is usually taken to mean “placing in random order”, which is exactly not the case. And at every point of the intermediate phase, on both mapper and reducer, the data is being sorted (rather than only right at the end). This is horribly confusing, and we won’t use those terms. Instead, we will refer to a single intermediate phase called the “group-sort phase”. Last and worst is the phrase “Reducer”. There is no obligation on a reducer that it eliminate data, that its output be smaller in size or fewer in count than its input, that its output combine records from its input or even pay attention to them at all. Reducers quite commonly emit more data than they receive, and if you’re not careful explosively so. We’re stuck with the name “Map/Reduce”, and so we’re also stuck calling this the “Reduce” phase, but put any concept of reduction out of your mind.

The Map Phase Processes Records Individually

The Map phase receives 0, 1 or many records individually, with no guarantees from Hadoop about their numbering, order or allocation.⁴ Hadoop does guarantee that every record arrives in whole to exactly one Map task and that the job will only succeed if every record is processed without error.

The Mapper receives those records sequentially — it must fully process one before it receives the next — and can emit 0, 1 or many inputs of any shape or size. The chimpanzees working on the SantaCorp project received letters but dispatched toy forms. Julia’s thoughtful note produced two toy forms, one for her doll and one for Joe’s robot, while the spam letter produced no toy forms.

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

4. In special cases, you may know that your input bears additional guarantees — for example, the “Merge Join” described in Chapter (REF) requires its inputs to be in total sorted order. It is on you, however, to enforce and leverage those special properties.

Please be aware, however, that it is only appropriate to access external resources from within a Hadoop job in exceptionally rare cases. Hadoop processes data in batches, which means failure of a single record results in the retry of the entire batch. It also means that when the remote resource is unavailable or responding sluggishly, Hadoop will spend several minutes and unacceptably many retries before abandoning the effort. Lastly, Hadoop is designed to drive every system resource at its disposal to its performance limit.⁵

For another extreme example, Hadoop's *distcp* utility, used to copy data from cluster to cluster, moves around a large amount of data yet has only a trivial input and trivial output. In a *distcp* job, each mapper's input is a remote file to fetch; the action of the mapper is to write the file's contents directly to the HDFS as a datanode client; and the mapper's output is a summary of what was transferred.

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

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

The Hadoop Contract

Here in one place is a casually rigorous summation of the very few guarantees Hadoop provides your Map/Reduce program. Understanding these is a critical tool for helping you to create and reason about Hadoop workflows.

- Each record is processed in whole by *exactly one* Mapper.
- Each Mapper receives records from *exactly one* contiguous split of input data, in the same order as those records appear in the source.

5. We will drive this point home in the chapter on Event Log Processing (REF), where we will stress test a web server to its performance limit by replaying its request logs at full speed.

- There are no guarantees on how long a split is, how many there are, the order in which they are processed or the assignment of split to Mapper slot.
- In both Mapper and Reducer, there is no requirement on you to use any of the structure described here or even to use the records' contents at all. You do not have to do anything special when a partition or group begins or ends and your program can emit as much or as little data as you like before, during or after processing its input stream.
- In a Mapper-only job, each Mapper's output is placed in *exactly one* uniquely-named, immutable output file in the order the records were emitted. There are no further relevant guarantees for a Mapper-Only job.
- Each Mapper output record is processed in whole by *exactly one* Reducer.
- Your program must provide each output record with a label consisting of a partition key, group key and sort key; these expressly govern how Hadoop assigns records to Reducers.
- All records sharing a partition key are sent to the same Reducer; if a Reducer sees one record from a partition, it will see all records from that partition, and no other Reducer will see any record from that partition.
- Partitions are sent contiguously to the Reducer; if a Reducer receives one record from a partition, it will receive all of them in a stretch, and will never again see a record from a prior partition.
- Partitions themselves are ordered by partition key within the Reducer input.
- A custom partitioner can assign each partition to specific Reducer, but you should not depend on any pairing provided by the default partitioner (the `HashPartitioner`).
- Within each partition, records are sent within contiguous groups; if a Reducer receives one record from a group, it will receive all of them in a stretch, and will never again see a record from a prior group.
- Within a partition, records are sorted first by the group key, then by the sort key; this means groups themselves are ordered by group key within the Reducer input. (TECHREVIEW: Check that this is consistent with the Java API and the Pig UDF API.)
- Each Reducer's output is placed in *exactly one* uniquely-named, immutable output file in the order the records were emitted.

You can tell how important we feel it is for you to internalize this list of guarantees, or we would not have gotten all, like, formal and stuff.

How Hadoop Manages Midstream Data

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

Mappers Spill Data In Sorted Chunks

As your Map task produces each labeled record, Hadoop inserts it into a memory buffer according to its order. Like the dexterous chimpanzee, the current performance of CPU and memory means this initial ordering imposes negligible overhead compared to the rate that data can be read and processed. When the Map task concludes or that memory buffer fills, its contents are flushed as a stream to disk. The typical Map task operates on a single HDFS block and produces an output size not much larger. A well-configured Hadoop cluster sets the sort buffer size accordingly⁶.

If there are multiple spills, Hadoop performs the additional action of merge-sorting the chunks into a single spill.⁷

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

Partitioners Assign Each Record To A Reducer By Label

The default partitioner, which we find meets almost all our needs, is called the “RandomPartitioner.”⁸ It aims to distribute records uniformly across the Reducers by giving each key the same chance to land on any given Reducer. It is not really random in the sense of nondeterministic; running the same job with the same configuration will distribute records the same way. Rather, it achieves a uniform distribution of keys by generating a cryptographic digest — a number produced from the key with the property that any change to that key would instead produce an arbitrarily distinct number. Since

6. The chapter on Hadoop Tuning For The Brave And Foolish (REF) shows you how; that most common case produces only a single spill.
7. This can be somewhat expensive, so in Chapter (REF), we will show you how to avoid unnecessary spills.) Whereas the pygmy elephants each belonged to a distinct workbench, a Hadoop Mapper produces only that one unified spill. That's ok — it is easy enough for Hadoop to direct the records as each is sent to its Reducer.
8. In the next chapter (REF), you will meet another partitioner, when you learn how to do a total sort.

the numbers thus produced have high and uniform distribution, the digest MODULO the number of Reducers reliably balances the Reducer's keys, no matter their raw shape and size.⁹



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

Reducers Receive Sorted Chunks From Mappers

Partway through your job's execution, you will notice its Reducers spring to life. Before each Map task concludes, it streams its final merged spill over the network to the appropriate Reducers¹⁰. Just as above, the Reducers file each record into a sort buffer, spills that buffer to disk as it fills and begins merge/sorting them once a threshold of spills is reached.

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

Reducers Read Records With A Final Merge/Sort Pass

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

Your program's Reducer receives the records from each group in sorted order, outputting records as it goes. Your reducer can output as few or as many records as you like at any time: on the start or end of its run, on any record, or on the start or end of a

9. If you will recall, $x \text{ MODULO } y$ gives the remainder after dividing x and y . You can picture it as a clock with y hours on it: 15 MODULO 12 is 3; 4 MODULO 12 is 4; 12 MODULO 12 is 0.
10. NOTE: Note that this communication is direct; it does not use the HDFS.

group. It is not uncommon for a job to produce output the same size as or larger than its input — “Reducer” is a fairly poor choice of names. Those output records can also be of any size, shape or format; they do not have to resemble the input records, and they do not even have to be amenable to further Map/Reduce processing.

Reducers Write Output Data and Commit

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

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

A Quick Note on Storage (HDFS)

If you’re a Hadoop *administrator* responsible for cluster setup and maintenance, you’ll want to know a lot about Hadoop’s underlying storage mechanism, called HDFS. As an *analyst* who writes jobs to run on a Hadoop cluster, though, you need to know just one key fact:

HDFS likes big files.

Put another way, *HDFS doesn’t like small files*, and “small” is “anything that weighs less than 64 megabytes.” If you’re interested in the technical specifics, you can check out the blog post on “The Small Files Problem”¹¹. Really, you just want to know that small files will really gum up the works.

This often leads people to ask: “How do I use Hadoop on, say, image analysis? I want to a large number of images that are only a few kilobytes in size.” For that, check out a Hadoop storage format called a *SequenceFile*.¹²

11. <http://blog.cloudera.com/blog/2009/02/the-small-files-problem/>

12. Also, Q wrote a handy tool to wrap up your small files into big SequenceFiles. Check out *forqlift* at <http://qethanm.cc/projects/forqlift/>

Outro

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

Introduction to Pig

Pig Helps Hadoop work with Tables, not Records

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

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

```
sightings = LOAD '/path/to/geo/ufo_sightings/ufo_sightings.tsv' AS (
    sighted_at: chararray, reported_at: chararray, location_str: chararray, shape: chararray,
    duration_str: chararray, description: chararray, lng: float, lat: float,
    city: chararray, county: chararray, state: chararray, country: chararray);

year_month_ct      = FOREACH sightings GENERATE SUBSTRING(sighted_at, 0, 7) AS yrmo;
sightings_hist     = FOREACH (GROUP year_month_ct BY yrmo) GENERATE
    group, COUNT_STAR(year_month_ct);
STORE sightings_hist INTO '/data/out/ufo/sightings_hist';
```

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

To run the Pig job, go into the example code repository and run

```
pig 04-intro_to_pig/a-ufo_visits_by_month.pig
```

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

Wikipedia Visitor Counts

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

```
SET DEFAULT_PARALLEL 3
;

pageviews = LOAD '/data/rawd/wikipedia/pagecounts/pagecounts-20081002-230001.log' AS (
    filename:chararray, ns:chararray, wikipedia_id:chararray, views:long, bytes:long
) USING PigStorage('\t', 'tagFile');

year_month_ct      = FOREACH pageviews  GENERATE SUBSTRING(sighted_at, 0, 7) AS yrmo;
sightings_hist     = FOREACH (GROUP year_month_ct BY yrmo) GENERATE
    group, COUNT_STAR(year_month_ct);
STORE sightings_hist INTO '/data/out/ufo/sightings_hist';

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

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

The second script instructed Pig to explicitly sort the output by minute, an additional operation. We did not do that in the first example because its data was so small that we had instructed Hadoop to use a single Reducer. As you will recall from the previous chapter (REF), Hadoop uses a Sort to prepare the Reducer groups, so its output was naturally ordered. If there are multiple Reducers, however, that would not be enough

to give you a Result file you can treat as ordered. By default, Hadoop assigns partitions to Reducers using the ‘RandomPartitioner’, designed to give each Reducer a uniform chance of claiming any given partition. This defends against the problem of one Reducer becoming overwhelmed with an unfair share of records but means the keys are distributed willy-nilly across machines. Although each Reducer’s output is sorted, you will see records from 2008 at the top of each result file and records from 2012 at the bottom of each result file.

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

Hadoop, however, has no intrinsic way to make that mapping happen. Even if it figured out, say, that the earliest buckets were in 2008 and the latest buckets were in 2012, if we fed it a dataset with skyrocketing traffic in 2013, we would end up sending an overwhelming portion of results to that Reducer. In the second job, Pig sampled the set of output keys, brought them to the same Reducer, and figured out the set of partition breakpoints to distribute records fairly.

In general, Pig offers many more optimizations beyond these and we will talk more about them in the chapter on Advanced Pig (REF). In our experience, as long as you’re willing to give Pig a bit of coaching, the only times it will author a dataflow that is significantly less performant comes when Pig is *overly* aggressive about introducing an optimization. And in those cases the impact is more like a bunch of silly piglets making things take 50% longer than they should, rather than a stampede of boars blowing up your cluster. The ORDER BY example is a case in point: for small-to-medium tables the intermediate sampling stage to calculate partitions can have a larger time cost than the penalty for partitioning badly would carry. Sometimes you’re stuck paying an extra 20 seconds on top of each one-minute job so that Pig and Hadoop can save you an order of magnitude off your ten-minute-and-up jobs.

Fundamental Data Operations

Pig’s operators — and fundamental Hadoop processing patterns — can be grouped into several families.

A control operation either influences or describes the data flow itself. A pipelinable operation is one that does not require a reduce step of its own: the records can each be handled in isolation, and so they do not have to be expensively assembled into context.

All structural operations must put records into context: placing all records for a given key into common context; sorting involves placing each record into context with the record that precedes it and the record that follows it; eliminating duplicates means putting all potential duplicates into common context, and so forth.

Control Operations

- Serialization operations (LOAD, STORE) load and store data into file systems or datastores.
- Directives (DESCRIBE, ILLUSTRATE, REGISTER, and others) to Pig itself. These do not modify the data, they modify Pig's execution: outputting debug information, registering external UDFs, and so forth.

Pipelinable Operations

With no structural operations, these operations create a mapper-only job with the composed pipeline. When they come before or after a structural operation, they are composed into the mapper or reducer.

- Transformation operations (FOREACH, FOREACH..FLATTEN(tuple)) modify the contents of records individually. The count of output records is exactly the same as the count of input records, but the contents and schema of the records can change arbitrarily.
- Filtering operations (FILTER, SAMPLE, LIMIT, ASSERT) accept or reject each record individually. These can yield the same or fewer number of records, but each record has the same contents and schema as its input.
- Repartitioning operations (SPLIT, UNION) don't change records, they just distribute them into new tables or data flows. UNION outputs exactly as many records as the sum of its inputs. Since SPLIT is effectively several FILTERs run simultaneously, its total output record count is the sum of what each of its filters would produce.
- Ungrouping operations (FOREACH..FLATTEN(bag)) turn records that have bags of tuples into records with each such tuple from the bags in combination. It is most commonly seen after a grouping operation (and thus occurs within the Reduce) but can be used on its own (in which case like the other pipelinable operations it produces a Mapper-Only job). The FLATTEN itself leaves the bag contents unaltered and substitutes the bag field's schema with the schema of its contents. When flattening on a single field, the count of output records is exactly the count of elements in all bags. (Records with empty bags will disappear in the output). Multiple FLATTEN clauses yield a record for each possible combination of elements, which can be explosively higher than the input count.

Structural Operations

These jobs require a Map and Reduce phase.

- Grouping operations (GROUP, COGROUP, CUBE, ROLLUP) place records into context with each other. They make no modifications to the input records' contents, but do rearrange their schema. You will often find them followed by a FOREACH that is able to take advantage of the group context. The GROUP and COGROUP yield one output record per distinct GROUP value.
- Joining operations (JOIN, CROSS) match records between tables. JOIN is simply an optimized COGROUP/FLATTEN/FOREACH sequence, but it is importantly enough different in use that we'll see its output size follows the same logic as FLATTEN. (CROSS too, except for the "important" part: we'll have very little to say about it and discourage its use).
- Sorting operations (ORDER BY, RANK) perform a total sort on their input; every record in file 00000 is in sorted order and comes before all records in 00001 and so forth for the number of output files. These require two jobs: first, a light Mapper-Only pass to understand the distribution of sort keys, next a Map/Reduce job to perform the sort.
- Uniquing and (DISTINCT, specific COGROUP forms) select/reject/collapse duplicates, or find records associated with unique or duplicated records. these are typically accomplished with specific combinations of the above, but involve

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

Pig is an extremely sparse language. By having very few Operators and very uniform syntax ¹, the language makes it easy for the robots to optimize the dataflow and for humans to predict and reason about its performance.

We will not explore every nook and cranny of its syntax, only illustrate its patterns of use. The online Pig manual at <http://pig.apache.org/> is quite good and for a deeper exploration, consult *Programming Pig* by Alan Gates (<http://shop.oreilly.com/product/0636920018087.do>). If the need for a construction never arose naturally in a pattern

1. Something SQL users but non-enthusiasts like your authors appreciate.

demonstration or exploration², we omitted it, along with options or alternate forms of construction that are either dangerous or rarely-used³.

In the remainder of this chapter, we'll illustrate the mechanics of using Pig and the essential of its control flow operations by demonstrating them in actual use. In the following several chapters (REF), we'll cover patterns of pipelinable and of structural operations. In each case the goal is not only to understand its use, but to understand how to implement the corresponding patterns in a plain map-reduce approach — and therefore how to reason about their performance. Finally, the chapter on Advanced Pig (TODO ref) will cover some deeper-level topics, such as a few important optimized variants of the JOIN statement and how to extend Pig with new functions and loaders.

LOAD Locates and Describes Your Data

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

```
articles = LOAD './data/wp/articles.tsv' AS (
    page_id: long, namespace: int, wikipedia_id: chararray,
    revision_id: long, timestamp: long,
    title: chararray, redirect: chararray,
    text: chararray);
hadoop_articles = FILTER articles BY text MATCHES '.*Hadoop.*';
STORE hadoop_articles INTO './data/tmp/hadoop_articles.tsv';
```

Simple Types

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

The `chararray` type loads text as UTF-8 encoded strings (the only kind of string you should ever traffic in). String literals are contained in single quotes — `'hello, world'`. Regular expressions are supplied as string literals, as in the example above: `'.*[Hh]adoop.*'`. The `bytearray` type does no interpretation of its contents whatso-

2. An example of the first is `UNION ONSCHEMA` — useful but not used.
3. it's legal in Pig to load data without a schema — but you shouldn't, and so we're not going to tell you how.

ever, but be careful — the most common interchange formats (`tsv`, `xml` and `json`) cannot faithfully round-trip data that is truly freeform.

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

Boolean, date and the `biginteger/bigdecimal` types are recent additions to Pig, and you will notice rough edges around their use.

Complex Type 1, Tuples: Fixed-length Sequence of Typed Fields

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

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

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

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

Complex Type 2, Bags: Unbounded Collection of Tuples

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

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

You can also address values within a bag using `bag_name.(subfield_a, sub field_b)`, but this time the result is a bag with the given projected tuples. You'll see examples of this shortly when we discuss FLATTEN and the various group operations. Note that the *only* type a bag holds is tuple, even if there's only one field — a bag of just park ids would have schema `bag{tuple(park_id:chararray)}`.

Complex Type 3, Maps: Collection of Key-Value Pairs for Lookup

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

```
REGISTER piggybank.jar
raw_tweets = LOAD '/tmp/tweets.json' USING org.apache.pig.piggybank.storage.JsonLoader(
    'created_at:chararray, text:chararray, user:map[]');
```

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

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

Defining the Schema of a Transformed Record

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

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

In the chapter on Advanced Pig (REF), we'll cover some further topics: loading from alternate file formats or from databases; how Pig and Hadoop assign input file splits to mappers; and custom load/store functions.

STORE Writes Data to Disk

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

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

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

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

The behavior of only evaluating on demand is an incredibly useful feature for development work. One of the best pieces of advice we can give you is to checkpoint all the time. Smart data scientists iteratively develop the first few transformations of a project, then

save that result to disk; working with that saved checkpoint, develop the next few transformations, then save it to disk; and so forth. Here's a demonstration:

```
great_start = LOAD '...' AS (...);  
-- ...  
-- lots of stuff happens, leading up to  
-- ...  
important_milestone = JOIN [...];  
  
-- reached an important milestone, so checkpoint to disk.  
STORE important_milestone INTO './data/tmp/important_milestone';  
important_milestone = LOAD './data/tmp/important_milestone' AS (...schema...);
```

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

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

Development Aids: **DESCRIBE**, **ASSERT**, **EXPLAIN**, **LIMIT**.., **DUMP**, **ILLUSTRATE**

DESCRIBE shows the schema of a table

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

ASSERT checks that your data is as you think it is

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

DUMP shows data on the console with great peril

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

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

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

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

Lastly, while we're on the subject of development tools that don't work perfectly in Pig: the Pig shell gets confused too easily to be useful. You're best off just running your script directly. For local mode development, tools like watchr (REF) can intelligently relaunch a script every time you hit save, streamlining development.

EXPLAIN shows Pig's execution graph

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

Pig Functions act on fields

Pig wouldn’t be complete without a way to *act* on the various fields. It offers a sparse but essential set of built-in functions — the Pig cheatsheet (TODO ref) at the end of the book gives a full list. The whole middle of the book is devoted to examples of Pig and map/reduce programs in practice (and in particular a chapter on Statistics), so we’ll just list the highlights here:

- **Math functions** for all the things you’d expect to see on a good calculator: LOG/LOG10/EXP, RANDOM, ROUND/’DOUND_TO’/FLOOR/CEIL, ABS, trigonometric functions, and so forth.
- **String comparison:**
 - matches tests a value against a regular expression:
 - Compare strings directly using ==. EqualsIgnoreCase does a case-insensitive match, while STARTSWITH/ENDSWITH test whether one string is a prefix or suffix of the other.
 - SIZE returns the number of characters in a chararray, and the number of bytes in a bytearray. Be reminded that characters often occupy more than one byte: the string *Motörhead* has nine characters, but because of its umlaut-ed ö the string occupies ten bytes. You can use SIZE on other types, too; but to find the number of elements in a bag, use COUNT_STAR and not SIZE.
 - INDEXOF finds the character position of a substring within a chararray
- **Transform strings:**
 - CONCAT concatenates all its inputs into a new string; SPRINTF uses a supplied template to format its inputs into a new string; BagToString joins the contents of a bag into a single string, separated by a supplied delimiter
 - LOWER converts a string to lowercase characters; UPPER to all uppercase
 - TRIM strips leading and trailing whitespace
 - REPLACE(string, 'regexp', 'replacement') substitutes the replacement string wherever the given regular expression matches, as implemented by

`java.string.replaceAll`. If there are no matches, the input string is passed through unchanged.

- `REGEX_EXTRACT(string, regexp, index)` applies the given regular expression and returns the contents of the indicated matched group. If the regular expression does not match, it returns NULL. The `REGEX_EXTRACT_ALL` function is similar, but returns a tuple of the matched groups.
- `STRSPLIT` splits a string at each match of the given regular expression
- `SUBSTRING` selects a portion of a string based on position
- **Datetime Functions**, such as `CurrentTime`, `ToUnixTime`, `SecondsBetween` (duration between two given datetimes)
- **Aggregate functions** that act on bags:
 - `AVG`, `MAX`, `MIN`, `SUM`
 - `COUNT_STAR` reports the number of elements in a bag, including nulls; `COUNT` reports the number of non-null elements. `IsEmpty` tests that a bag has elements. Don't use the quite-similar-sounding `SIZE` function on bags: it's much less efficient.
- **Bag Functions** TODO COMPLETE LIST
 - `Extremal`
 - `FirstTupleInBag`
 - `BagConcat`
 - `Stitch / Over`
 - `SUBTRACT(bag_a, bag_b)` returns a new bag having all the tuples that are in the first but not in the second, and `DIFF(bag_a, bag_b)` returns a new bag having all tuples that are in either but not in both. These are rarely used, as the bags must be of modest size — in general us an inner JOIN as described below.
 - `TOP(num, column_index, bag)` selects the top `num` of elements from each tuple in the given bag, as ordered by `column_index`. This uses a clever algorithm that doesn't require an expensive total sort of the data — you'll learn about it in the Statistics chapter (TODO ref)
- **Conversion Functions** to perform higher-level type casting: `TOTUPLE`, `TOBAG`, `TOMAP`
- Add all the DataFu operations: <http://datafu.incubator.apache.org/docs/datafu/guide/bag-operations.html> and coalesce <http://datafu.incubator.apache.org/docs/datafu/guide/more-tips-and-tricks.html> and maybe others
 - `Hasher`

Moving right along ...

This chapter was a gentle introduction to Pig and its basic operations. In the next two chapters, we'll see Pig in action.⁴

4. Your authors struggled to not make a “see Pig fly” joke. Oh, wait, looks like we did ...

PART II

Tactics: Analytic Patterns

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

This part of the book will equip you to think tactically, to think in terms of the changes you would like to make to the data. Each section introduces a repeatedly-useful data transformation pattern, demonstrated in Pig (and, where we'd like to reinforce the record-by-record action, in Wukong as well). The next part of the book will focus on *strategic* techniques, to assemble the patterns you're about to see into an explanation that tells a coherent story.

One of this book's principles is to center demonstrations around an interesting and realistic problem from some domain. And whenever possible, we endeavor to indicate how the approach would extend to other domains, especially ones with an obvious business focus. The tactical patterns, however, are exactly those tools that crop up in nearly every domain: think of them as the screwdriver, torque wrench, lathe and so forth of your toolkit. Now, if this book were called "Big Mechanics for Chimps", we might introduce those tools by repairing and rebuilding a Volkswagen Beetle engine, or by building another lathe from scratch. Those lessons would carry over to anywhere machine tools apply: air conditioner repair, fixing your kid's bike, building a rocket ship to Mars.

So we will center this part of the book on what Nate Silver calls "the perfect data set": the sea of numbers surrounding the sport of baseball. The members of the Retrosheet and Baseball Databank projects have provided an extraordinary resource: comprehensive statistics from the birth of the game in the late 1800s until the present day, freely available and redistributable. There is an overview of the stats we'll use in (REF sidebar),

and further information in the “Overview of Datasets” appendix (REF). Even if you’re not a baseball fan, we’ve minimized the number of concepts you’ll need to learn.

In particular, we will be hopping in and out of two main storylines as each pattern is introduced. One is a graphical biography of each player and team — the data tables for a website that can display timelines, maps and charts of the major events and people in the history of a team or player. This is explanatory analytics, where the goal is to summarize the answers to well-determined questions for presentation. We will demonstrate finding the geographical coordinates for each stadium or assembling the events in a player’s career in a way that you can apply any time you want to show things on a map or display a timeline. When we demonstrate the self-join by listing each player’s teammates, we’re showing you how to list all other products purchased in the same shopping cart as a product, or all pages co-visited by a user during a website session, and any other occasion where you want to extend a relationship by one degree.

The other storyline is to find indicators of exceptional performance, supplying a quantitative basis for the age-old question “Who are the greatest players in the game?”. This is exploratory analytics, where the work is as much to determine the questions as to assemble the answers. Quantifying “how many great seasons did this player have?” or “” is. As we pursue this exploration, you should recognize not just a way for fantasy baseball players to get an edge, but strategies for quantifying the behavior of any sort of outlier. Here, it’s baseball players, but similar questions will apply when examining agents posing security threats, load factor on public transit routes, factors causing manufacturing defects, cell strains with a significantly positive response, and many other topics of importance.

In many cases, though, a pattern has no natural demonstration in service of those primary stories, and so we’ll find questions that could support an investigation of their own: “How can we track changes in each team’s roster over time?”, “Is the stereotypical picture of the big brawny home-run hitter true?” For these we will usually just show the setup and stop at the trailhead. But when the data comes forth with a story so compelling it demands investigation (“Does God really hate Cleveland?”, “Why are baseball players more likely to die in January and be born in August?”) we will take a brief side trip to follow the tale.

Each pattern is followed by a “pattern in use” synopsis that suggests alternative business contexts for this pattern, lists important caveats and associated patterns, and how to reason about its performance. These will become most useful once you’ve read the book a first time and (we hope) begin using it as your go-to reference. If you find the level of detail here to be a bit intense, skim past them on the first reading.

What’s most important is that you learn the mechanics of each pattern, ignoring the story if you must. The best thing you can do is to grab a data set of your own — from your work or research, or one of the datasets listed in our Overview of Datasets (REF) — and translate the patterns to that domain. Don’t worry about finding an overarching

theme like our performance-outliers storyline, just get a feel for the craft of using Hadoop at scale.

Analytic Patterns part 1: Pipeline Operations

This chapter focuses exclusively on what we'll call *pipelineable operations*. A pipelineable operations is one that can handle each record in isolation, like the translator chimps from Chimpanzee & Elephant's first job. That property makes those operations trivially parallelizable: they require no reduce phase of their own.

When a script has only pipelineable operations, they give rise to one mapper-only job which executes the composed pipeline stages. When pipelinable operations are combined with the structural operations you'll meet in the next chapter, they are composed with the stages of the mapper or reducer (depending on whether they come before or after the structural operation).

All of these are listed first and together for two reasons. One, they are largely fundamental; it's hard to get much done without FILTER or FOREACH. Two, the way you reason about the performance impact of these operations is largely the same. Since these operations are trivially parallelizable, they scale efficiently and the computation cost rarely impedes throughput. And when pipelined, their performance cost can be summarized as "kids eat free with purchase of adult meal". For datasets of any material size, it's very rare that the cost of preliminary or follow-on processing rivals the cost of the reduce phase. Finally, since these operations handle records in isolation, their memory impact is modest. So learn to think of these together.

Eliminating Data

The first round of patterns will focus on methods to shrink your dataset. This may sound counterintuitive to the novice ear: isn't the whole point of "Big Data" that we get to work with the entire dataset at once? We finally develop models based on the entire population, not a sample thereof, so why should we scale down our data?

The primary reason is to focus on a subset of records: only website requests with an external referrer, only security events with high threat levels, only accounts of than \$1 million. And even when you work with every *record* in a dataset, you may be interested in a subset of *fields* relevant to your research. For reasons of memory and computational efficiency, and also your sanity, you'd do yourself a favor to immediately trim a working dataset down to just those records and fields relevant to the task at hand.¹ Furthermore, you may wish to test some code on a small sample before unleashing it on a long-running job.² Last, but not least, you may want to draw a random sample just to spot-check a dataset when it's too computationally expensive to inspect every element.

The goal of course isn't to *eliminate* data, it's to be *selective* about your data, and so we will introduce you to a variety of techniques for doing so.

Selecting Records that Satisfy a Condition: FILTER and Friends

The first step to eliminating (or being selective about) data is to reject records that don't match certain criteria. Pig's FILTER statement does this for you. It doesn't remove the data — all data in Hadoop and thus Pig is immutable — rather like all Pig operations it creates a new table that omits certain records from the input.

The baseball stats go back to 1871 (!), but it took a few decades for the game to reach its modern form. Let's say we're only interested in seasons since 1900. In Pig, we apply the FILTER operation³:

```
modern_stats = FILTER bats BY (year_id >= 1900);
```

The range of conditional expressions you'd expect are present: == (double-equals) to express an equality condition, != for not-equals, and >, >=, <, <= for inequalities; IN for presence in a list; and MATCHES for string pattern matching. (More on those last two in a bit.)

1. This will certainly simplify debugging. It also plays to Q's favorite refrain of, *know your data*. If you're working on a dataset and there are additional fields or records you don't plan to use, can you be certain they won't somehow creep into your model? The worst-case scenario here is what's called a feature leak, wherein your target variable winds up in your training data. (In essence: imagine saying you can predict today's high temperature, so long as you are first provided today's high temperature.) A feature leak can lead to painful surprises when you deploy this model to the real world.
2. This is generally a good habit to develop, especially if you're one to kick off jobs before leaving the office, going to bed, or boarding a long-haul flight.
3. In this and in further scripts, we're going omit the LOAD, STORE and other boilerplate except to prove a point. See the example code (REF) for fully-working snippets

Selecting Records that Satisfy Multiple Conditions

In a data exploration, it's often important to exclude subjects with sparse data, either to eliminate small-sample-size artifacts, or because they are not in the focus of interest. In our case, we will often want to restrict analysis to regular players — those who have seen significant playing time in a season — while allowing for injury or situational replacement. Since major-league players come to bat a bit over 4 times a game on average in a season of 154 to 162 games (it increased in 1960), we can take 450 plate appearances (roughly 2/3 of the maximum) as our threshold ⁴.

In Pig, you can also combine conditional statements with AND, OR, NOT. The following selects what we'll call “qualified modern seasons”: regular players, competing in the modern era, in either of the two modern leagues.

```
modsig_stats = FILTER bats BY  
  (PA >= 450) AND (year_id >= 1900) AND ((lg_id == 'AL') OR (lg_id == 'NL'));
```

Selecting or Rejecting Records with a null Value

Another table we'll be working with is the ‘people’ table. It describes players’ vital statistics: their name; where and when they were born and died; when their career started and ended; their height and weight; and so forth. The data is quite comprehensive, but in some cases the fields have `null` values. Nulls are used in practice for many things:

- *Missing/unknown Value* — the case for a fraction of early players’ birthplaces or birth dates
- *No Value Applies* — players who are still alive have `null` in the fields for date and location of death
- *Ill-formed Value* — if a corrupt line creates an unparseable cell (eg a value of 'Bob' for an int), Pig will write a warning to the log but otherwise load it without complaint as `null`.
- *Illegal value* — Division by zero and similar misbehavior results in a `null` value (and not an error, warning, or log statement)
- “*Other*” — People will use a `null` value in general to represent “it’s complicated, but maybe some other field has details”.

We can exclude players whose birth year or birth place is unknown with a FILTER statement:

```
borned = FILTER people BY (birth_year IS NOT NULL) AND (birth_place IS NOT NULL);
```

4. Not coincidentally, that figure of 450 PA is close to the “qualified” season threshold of 3.1 plate appearances per team game that are required for seasonal performance awards

For those coming from a SQL background, Pig's handling of `null` values will be fairly familiar. For the rest of us, good luck. Null values generally disappear without notice from operations, and generally compare as `null` (which signifies neither `false` nor `true`). And so `null` is not less than 5.0, it is not greater than 5.0, and it is not equal to 5.0. A null value is not equal to `null`, and is not *unequal* to `null`. You can see why for programmers it can be hard to track all this. All the fiddly collection of rules are well detailed in the Pig manual, so we won't go deep into them here — we've found the best way to learn what you need is to just see lots of examples, which we endeavor to supply in abundance.

Pattern in Use

Blocks like the following will show up after each of the patterns or groups of patterns we cover. Not every field will be present every time, since there's not always anything interesting to say.

- *Where You'll Use It* — (*The business or programming context.*) Everywhere. Like the f-stop on your camera, composing a photo begins and ends with throttling its illumination.
- *Standard Snippet* — (*Just enough of the code to remind you how it's spelled.*) `somerecords = FILTER myrecords BY (criteria AND criteria ...);`
- *Hello, SQL Users* — (*A sketch of the corresponding SQL command, and important caveats for people coming from a SQL background.*) `SELECT bat_season.* FROM bat_season WHERE year_id >= 1900;`
- *Important to Know* — (*Caveats about its use. Things that you won't understand / won't buy into the first time through the book but will probably like to know later.*)
 - Filter early, filter often. The best thing you can do with a large data set is make it smaller.
 - SQL users take note: `==`, `!=` — not `=` or anything else.
 - Programmers take note: `AND`, `OR` — not `&&`, `||`.
- *Output Count* — (*How many records in the output: fewer, same, more, explosively more?*) Zero to 100% of the input record count. Data size will decrease accordingly
- *Records* — (*A sketch of what the records coming out of this operation look like*) Identical to input
- *Data Flow* — (*The Hadoop jobs this operation gives rise to. In this chapter, all the lines will look like this one; in the next chapters that will change*) Pipelinable: it's composed onto the end of the preceding map or reduce, and if it stands alone becomes a map-only job.

- *Exercises for You* — (A mission to carry forward, if you choose. Don’t go looking for an answer section — we haven’t done any of them. In many cases you’ll be the first to find the answer.) Play around with ‘null’s and the conditional operators until you have a good sense of its quirks.
- *See Also* — (Besides the patterns in its section of the book, what other topics might apply if you’re considering this one? Sometimes this is another section in the book, sometimes it’s a pointer elsewhere) The Distinct operations, some Set operations, and some Joins are also used to eliminate records according to some criteria. See especially the Semi-Join and Anti-Join (REF), which select or reject matches against a large list of keys.

Selecting Records that Match a Regular Expression (MATCHES)

A MATCHES expression employs regular expression pattern matching against string values. Regular expressions are given as plain chararray strings; there’s no special syntax, as Python/Ruby/Perl/etc-ists might have hoped. See the sidebar (REF) for important details and references that will help you master this important tool.

This operation uses a regular expression to select players with names similar to either of your authors’ names:

```
-- Name contains a Q; is `Flip` or anything in the Philip/Phillip/... family. (?i) means be case-insensitive
namesakes = FILTER people BY (nameFirst MATCHES '(?i).*(q|flip|phil+ip).*');
```

It’s easy to forget that people’s names can contain spaces, dots, dashes, apostrophes; start with lowercase letters or apostrophes, and have accented or other non-latin characters⁵. So as a less silly demonstration of MATCHES, this snippet extracts all names which do not start with a capital letter or which contain a non-word non-space character:

```
funnychars = FILTER people BY (nameFirst MATCHES '^([A-Z]|.*[^\\w\\s]).*');
```

TODO: state very explicitly that these are just Java regexes, and point to docco on that
 TODO: maybe rephrase the below as “How to convert from a sane Regexp to a Pig/Java Regexp”

There are many players with non-word,non-space characters, but none whose names are represented as starting with a lowercase character. However, in early drafts of the book this query caught a record with the value “nameFirst” — the header rows from a source datafile had contaminated the table. Sanity checks like these are a good idea always, even moreso in Big Data. When you have billions of records, a one-in-a-million exception will appear thousands of times.

5. A demonstration of the general principle that if you believe an analysis involving people will be simple, you’re probably wrong.

Important Notes about String Matching

Regular expressions are incredibly powerful and we urge all readers to acquire basic familiarity. There is no better path to mastery than the [regexp.info](#) website, and we've provided a brief cheatsheet at the end of the book (REF). Here are some essential clarifications about Pig in particular:

- Regular expressions in Pig are supplied to the MATCHES operator as plain strings. A single backslash serves the purposes of the string literal and does not appear in the string sent to the regexp engine. To pass along the shorthand `[^\w\ \s]` (non-word non-space characters), we have to use two backslashes.
- Yes, that means matching a literal backslash in the target string is done with four backslashes: `\\\\'!`
- Options for matching are supplied within the string. For example, `(?i)` matches without regard to case (as we did above), `(?m)` to do multi-line matches, and so forth — see the documentation.
- Pig Regular Expressions are implicitly anchored at the beginning and end of the string, the equivalent of adding `^` at the start and `$` at the end. (This mirrors Java but is unlike most other languages.) Use `.*` at both ends, as we did above, to regain the conventional “greedy” behavior. Supplying explicit `^` or `$` when intended is a good habit for readability.
- MATCHES is an expression, like AND or == — you write `str MATCHES regexp`. The other regular expression mechanisms you'll meet are functions — you write `REGEX_EXTRACT(str, regexp, 1)`. You will forget we told you so the moment you finish this book.
- Appearing in the crop of results: Peek-A-Boo Veach, Quincy Trouppe, and Flip Lafferty.
- You're allowed to have the regular expression be a value from the record, though Pig is able to pre-compile a constant (literal) regexp string for a nice speedup.
- Pig doesn't offer an exact equivalent to the SQL % expression for simple string matching. The rough equivalents are dot-star `(.*)` for the SQL % (zero or more arbitrary characters), dot `(.)` for the SQL _ (a single character); and square brackets (e.g. `[a-z]`) for a character range, similar to SQL.
- The string equality expression is case sensitive: 'Peek-A-Boo' does not equal 'peek-a-boo'. For case-insensitive string matching, use the `EqualsIgnoreCase` function: `EqualsIgnoreCase('Peek-A-Boo', 'peek-a-boo')` is true. This simply invokes Java's `String.equalsIgnoreCase()` method and does not support regular expressions.



Sadly, the Nobel Prize-winning physicists Gerard 't Hooft, Louis-Victor Pierre Raymond de Broglie, or Tomonaga Shin'ichirō never made the major leagues. Or tried out, as far as we know. But their names are great counter-examples to keep in mind when dealing with names. Prof de Broglie's full name is 38 characters long, has a last name that starts with a lowercase letter, and is non-trivial to segment. "Tomonaga" is a family name, though it comes first. You'll see Prof. Tomonaga's name given variously as "Tomonaga Shin'ichiro", "Sin-Itiro Tomonaga", or "田中 真一郎", each one of them correct, and the others not, depending on context. Prof. 't Hooft's last name starts with an apostrophe, a lower-case-letter, and contains a space. You're well advised to start a little curio shelf in your workshop for counterexample collections such as these, and we'll share some of ours throughout the book.

Pattern in Use

- *Where You'll Use It* — Wherever you need to select records by a string field. For selecting against small lists. For finding ill-formed records. Matching against a subsection of a composite key — Can you figure out what `game_id MATCHES '... (19|20).*' in the games table does?`
- *Standard Snippet* — `FILTER recs BY (str MATCHES '.*pattern.*')`, sure, but also `FOREACH recs GENERATE (str MATCHES '.*(kitty|cat|meow).*' ? 'cat' : 'notcat') AS catness.`
- *Hello, SQL Users* — Similar to but more powerful than the `LIKE` operator. See the sidebar (ref) for a conversion guide.
- *Important to Know* --
 - Mostly, that these are incredibly powerful, and even if they seem arcane now they're much easier to learn than it first seems.
 - You're far better off learning one extra thing to do with a regular expression than most of the other string conditional functions Pig offers.
 - ... and enough other Importants to Know that we made a sidebar of them (REF).
- *Records* — You can use this in a filter clause but also anywhere else an expression is permitted, like the preceding snippet
- *Data Flow* — Pipelinable: it's composed onto the end of the preceding map or reduce, and if it stands alone becomes a map-only job.
- *Exercises for You* — Follow the [regexp.info tutorial](#), but *only up to the part on Grouping & Capturing*. The rest you are far better off picking up once you find you need it.

- *See Also* — The Pig REGEX_EXTRACT and REPLACE functions. Java’s Regular Expression documentation for details on its pecadilloes (but not for an education about regular expressions).

Matching Records against a Fixed List of Lookup Values

If you plan to filter by matching against a small static list of values, Pig offers the handy IN expression: true if the value is equal (case-sensitive) to any of the listed values. This selects the stadiums used each year by the current teams in baseball’s AL-east division:

```
al_east_parks = FILTER park_team_years BY
    team_id IN ('BAL', 'BOS', 'CLE', 'DET', 'ML4', 'NYA', 'TBA', 'TOR', 'WS2');
```

Sometimes a regular expression alternation can be the right choice instead. `bubba MATCHES 'shrimp (kabobs|creole|gumbo)soup|stew|salad|and potatoes|burger|sandwich)' OR bubba MATCHES '(pineapple|lemon|coconut|pepper|pan.fried|deep.fried|stir.fried) shrimp'` is more readable than `bubba IN ('shrimp kabobs', 'shrimp creole', 'shrimp gumbo', ...)`.

When the list grows somewhat larger, an alternative is to read it into a set-membership data structure⁶, but ultimately large data sets belong in data files.

The general case is handled by using a join, as described in the next chapter (REF) under “Selecting Records Having a Match in Another Table (semi-join)”. See in particular the specialized merge join and HashMap (replicated) join, which can offer a great speedup if you meet their qualifications. Finally, you may find yourself with an extremely large table but with few elements expected to match. In that case, a Bloom Filter may be appropriate. They’re discussed more in the statistics chapter, where use a Bloom Filter to match every phrase in a large document set against a large list of place names, effectively geolocating the documents.

Pattern in Use

- *Where You’ll Use It* — File types or IP addresses to select/reject from web logs. Keys for exemplar records you’re tracking through a dataflow. Stock symbols you’re researching. Together with “Summarizing Multiple Subsets of a Group Simultaneously” (REF), enumerate members of a cohort (`((state IN ('CA', 'WA', 'OR') ? 1 : 0) AS is_western, ...)`).
- *Standard Snippet* — `foo IN ('this', 'that', 'the_other')`, or any of the other variants given above

6. For a dynamic language such as Ruby, it can often be both faster and cleaner to reformat the table into the language itself than to parse a data file. Loading the table is now a one-liner (`require "lookup_table"`), and there’s nothing the Ruby interpreter does faster than interpret Ruby.

- *Hello, SQL Users* — This isn't anywhere near as powerful as SQL's IN expression. Most importantly, you can't supply another table as the list.
- *Important to Know* — A regular expression alternation is often the right choice instead.
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values. Data size should decrease greatly.
- *Data Flow* — Pipelinable: it's composed onto the end of the preceding map or reduce, and if it stands alone becomes a map-only job.

Project Only Chosen Columns by Name

While a FILTER selects *rows* based on an expression, Pig's FOREACH selects specific *fields* chosen by name. The fancy word for this simple action is “projection”. We'll try to be precise in using *project* for choosing columns, *select* for choosing rows by any means, and *filter* where we specifically mean selecting rows that satisfy a conditional expression.

The tables we're using come with an overwhelming wealth of stats, but we only need a few of them to do fairly sophisticated explorations. The gamelogs table has more than 90 columns; to extract just the teams and the final score, use a FOREACH:

```
game_scores = FOREACH games GENERATE
  away_team_id, home_team_id, home_runs_ct, away_runs_ct;
```

Using a FOREACH to Select, Rename and Reorder fields

You're not limited to simply restricting the number of columns; you can also rename and reorder them in a projection. Each record in the table above has *two* game outcomes, one for the home team and one for the away team. We can represent the same data in a table listing outcomes purely from each team's perspective:

```
games_a = FOREACH games GENERATE
  year_id, home_team_id AS team,
  home_runs_ct AS runs_for, away_runs_ct AS runs_against, 1 AS is_home:int;

games_b = FOREACH games GENERATE
  away_team_id AS team,      year_id,
  away_runs_ct AS runs_for, home_runs_ct AS runs_against, 0 AS is_home:int;

team_scores = UNION games_a, games_b;

DESCRIBE team_scores;
--   team_scores: {team: chararray,year_id: int,runs_for: int,runs_against: int,is_home: int}
```

The first projection puts the `home_team_id` into the `team` slot, renaming it `team`; retains the `year_id` field unchanged; and files the home and away scores under `runs_for` and

`runs_against`. Lastly, we slot in an indicator field for home games, supplying both the name and type as a matter of form. Next we generate the corresponding table for away games, then stack them together with the UNION operation (to which you'll be properly introduced in a few pages). All the tables have the identical schema shown, even though their values come from different columns in the original tables.

Pattern in Use

- *Where You'll Use It* — Nearly everywhere. If FILTER is the f-stop of our camera, this is the zoom lens.
- *Standard Snippet* — FOREACH `recs GENERATE only, some, columns;`
- *Important to Know* — As you can see, we take a lot of care visually aligning sub-expressions within the code snippets. That's not because we've tidied up the house for students coming over — this is what the code we write and the code our teammates expect us to write looks like.
- *Output Count* — Exactly the same as the input.
- *Records* — However you define them to be
- *Data Flow* — Pipelinable: it's composed onto the end of the preceding map or reduce, and if it stands alone becomes a map-only job.
- *See Also* — "Assembling Literals with Complex Type" (REF)

Extracting a Random Sample of Records

Another common operation is to extract a *uniform* sample — one where every record has an equivalent chance of being selected. For example, you could use this to test new code before running it against the entire dataset (and possibly having a long-running job fail due to a large number of mis-handled records). By calling the `SAMPLE` operator, you ask Pig to pluck out some records at random.

The following Pig code will return a randomly-selected 10% (that is, $1/10 = 0.10$) of the records from our baseball dataset:

```
some_seasons_samp = SAMPLE bat_seasons 0.10;
```

The SAMPLE operation does so by generating a random number to select records, which means each run of a script that uses SAMPLE will yield a different set of records. Sometimes this is what you want, or in the very least, you don't mind. In other cases, you may want to draw a uniform sample once, then repeatedly work through those *same* records. (Consider our example of spot-checking new code against a dataset: you'd need to run your code against the same sample in order to confirm your changes work as expected.)

Experienced software developers will reach for a “seeding” function — such as R’s `set.seed()` or Python’s `random.seed()` — to make the randomness a little less so. At the moment, Pig does not have an equivalent function. Even worse, it is not consistent *within the task* — if a map task fails on one machine, the retry attempt will generate different data sent to different reducers. This rarely causes problems, but for anyone looking to contribute back to the Pig project, this is a straightforward high-value issue to tackle.

Pattern in Use

- *Where You’ll Use It* — At the start of the exploration, to cut down on data size. In many machine learning algorithms. Don’t use it for simulations — you need to be taking aggressive charge of the sampling algorithm.
- *Important to Know*
 - A consistent sample is a much better practice, though we admit that can be more of a hassle. But records that dance around mean you can’t Know Thy Data as you should.
 - The DataFu package has UDFs for sampling with replacement and other advanced features.
- *Output Count* — Determined by the sampling fraction. As a rule of thumb, variances of things are square-root-ish; expect the size of a 10% sample to be in the 7%-13% range.
- *Records* — Identical to the input
- *Data Flow* — Pipelinable: it’s composed onto the end of the preceding map or reduce, and if it stands alone becomes a map-only job.
- *Exercises for You* — Modify Pig’s SAMPLE function to accept a seed parameter, and submit that patch back to the open-source project. This is a bit harder to do than it seems: sampling is key to efficient sorting and so the code to sample data is intertwined with a lot of core functionality.

Extracting a Consistent Sample of Records by Key

A good way to stabilize the sample from run to run is to use a *consistent hash digest*. A hash digest function creates a fixed-length fingerprint of a string whose output is otherwise unpredictable from the input and uniformly distributed — that is, you can’t tell which string the function will produce except by computing the digest, and every string is equally likely. For example, the hash function might give the hexadecimal-string digest `3ce3e909` for `Chimpanzee` but `07a05f9c` for `Chimp`. Since all hexadecimal strings have effectively equal likelihood, one-sixteenth of them will start with a zero, and so this filter would reject `Chimpanzee` but select `Chimp`.

Unfortunately, Pig doesn't have a good built-in hash digest function! Do we have to give up all hope? You'll find the answer later in the chapter (REF)⁷, but for now instead of using a good built-in hash digest function let's use a terrible hash digest function. A bit under 10% of player_ids start with the letter *s*, and any coupling between a player's name and performance would be far more subtle than we need to worry about. So the following simple snippet gives a 10% sample of batting seasons whose behavior should reasonably match that of the whole:

```
some_seasons = FILTER bat_seasons BY (SUBSTRING(player_id, 0, 1) == 's');
```

We called this a terrible hash function, but it does fit the bill. When applied to an arbitrary serial identifier it's not terrible at all — the Twitter firehose provides a 1% service tier which returns only tweets from users whose numeric ID ends in *00*, and a 10% tier with user IDs ending in *0*. We'll return to the subject with a proper hash digest function later on in the chapter, once you're brimming with even more smartitude than you are right now. We'll also have a lot more to say about sampling in the Statistics chapter (REF).

- *Where You'll Use It* — At the start of the exploration,
- *Important to Know*
 - If you'll be spending a bunch of time with a data set, using any kind of random sample to prepare your development sample might be a stupid idea. You'll notice that Red Sox players show up a lot of times in our examples — that's because our development samples are “seasons by Red Sox players” and “seasons from 2000-2010”, which lets us make good friends with the data.
- *Output Count* — Determined by the sampling fraction. As a rule of thumb, variances of things are square-root-ish; expect the size of a 10% sample to be in the 7%-13% range.
- *Records* — Identical to the input
- *Data Flow* — Pipelinable: it's composed onto the end of the preceding map or reduce, and if it stands alone becomes a map-only job.

Sampling Carelessly by Only Loading Some `part`- Files

Sometimes you just want to knock down the data size while developing your script, and don't much care about the exact population. If you find a prior stage has left you with 20 files `part-r-00000` through `part-r-00019`, specifying `part-r-0000[01]` (the first two out of twenty files) as the input to the next stage is a hamfisted but effective way to get a 10% sample. You can cheat even harder by adjusting the parallelism of the pre-

7. Spoiler alert: No, you don't have to give up all hope when Pig lacks a built-in function you require.

ceding stage to get you the file granularity you need. As long as you’re mindful that some operations leave the reducer with a biased selection of records, toggling back and forth between say `my_data/part-r-0000[01]` (two files) and `my_data/` (all files in that directory) can really speed up development.

Selecting a Fixed Number of Records with `LIMIT`

A much blunter way to create a smaller dataset is to take some fixed number K of records. Pig offers the `LIMIT` operator for this purpose. To select 25 records from our `bat_sons` data, you would run:

```
some_players = LIMIT player_year_stats 25;
```

This is somewhat similar to running the `head` command in Unix-like operating systems, or using the `LIMIT` clause in a SQL `SELECT` statement. However, unless you have explicitly imparted some order to the table (probably by sorting it with `ORDER`, which we’ll cover later (REF)), Pig gives you *no guarantee over which records it selects*. In the big data regime, where your data is striped across many machines, there’s no intrinsic notion of a record order. Changes in the number of mappers or reducers, in the data, or in the cluster may change which records are selected. In practice, you’ll find that it takes the first K records of the first-listed file (and so, as opposed to `SAMPLE`, generally gives the same outcome run-to-run), but it’s irresponsible to rely on that.

When you have a very large dataset, as long as you really just need any small piece of it, you can apply the previous trick as well and just specify a single input file. Invoking `LIMIT` on one file will prevent a lot of trivial map tasks from running.

Other Data Elimination Patterns

There are two tools we’ll meet in the next chapter that can be viewed as data elimination patterns as well. The `DISTINCT` and related operations are used to identify duplicated or unique records. Doing so requires putting each record in context with its possible duplicates — meaning they are not pure pipeline operations like the others here. Above, we gave you a few special cases of selecting records against a list of values. We’ll see the general case — selecting records having or lacking a match in another table, also known as semi-join and anti-join — when we meet all the flavors of the `JOIN` operation in the next chapter.

Transforming Records

Besides getting rid of old records, the second-most exciting thing to do with a big data set is to rip through them manufacturing new records ⁸. We've been quietly sneaking FOREACH into snippets, but it's time to make its proper acquaintance

Transform Records Individually using FOREACH

The FOREACH lets you develop simple transformations based on each record. It's the most versatile Pig operation and the one you'll spend the most time using.

To start with a basic example, this FOREACH statement combines the fields giving the city, state and country of birth for each player into the familiar comma-space separated combined form (Austin, TX, USA) ⁹.

```
birthplaces = FOREACH people GENERATE
    player_id,
    CONCAT(birth_city, ', ', birth_state, ', ', birth_country) AS birth_loc
;
```

The syntax should be largely self-explanatory: this runs through the people table, and outputs a table with two columns, the player ID and our synthesized string. In the output you'll see that when CONCAT encounters records with null values, it returned null as well without an error.

For the benefit of SQL aficionados, here's an equivalent SQL query:

```
SELECT
    player_id,
    CONCAT(birth_city, ', ', birth_state, ', ', birth_country) AS birth_loc
FROM people;
```

You'll recall we took some care when loading the data to describe the table's schema, and Pig makes it easy to ensure that the data continues to be typed. Run DESCRIBE birthplaces; to return the schema:

```
birthplaces: {player_id: chararray,birth_loc: chararray}
```

Since player_id carries through unchanged, its name and type convey to the new schema. Pig figures out that the result of CONCAT is a chararray, but it's up to us to award it with a new name (birth_loc).

8. Although you might re-rank things when we show you how to misuse Hadoop to stress-test a webserver with millions of concurrent requests per minute (REF)
9. The country field uses some ad-hoc mixture of full name and arbitrary abbreviations. In practice, we would have converted the country fields to use ISO two-letter abbreviations — and that's just what we'll do in a later section (REF)

A FOREACH won't cause a new Hadoop job stage; it's chained onto the end of the preceding operation (and when it's on its own, like this one, there's just a single a mapper-only job). It always produces exactly the same count of output records as input records, although as you've seen it can change the number of columns.

A nested FOREACH Allows Intermediate Expressions

Earlier we promised you a storyline in the form of an extended exploration of player performance. We've now gathered enough tactical prowess to set out¹⁰.

The stats in the 'bat_seasons' table are all "counting stats" — total numbers of hits, of games, and so forth — and certainly from the team's perspective the more hits the more better. But for comparing players, the counting stats don't distinguish between the player who eared 70 hits in a mere 200 trips to the plate before a season-ending injury, and the player who squandered 400 of his team's plate appearances getting to a similar total¹¹. We should also form "rate stats", normalizing those figures against plate appearances. The following simple metrics do quite a reasonable job of characterizing players' performance:

- *On-base percentage* (OBP) indicates how well the player meets offensive goal #1: get on base, thus becoming a potential run and *not* consuming a precious out. It is given as the fraction of plate appearances that are successful: $((H + BB + HBP) / PA)$ ¹². An OBP over 0.400 is very good (better than 95% of significant seasons).
- *Slugging Percentage* (SLG) indicates how well the player meets offensive goal #2: advance the runners on base, thus converting potential runs into points towards victory. It is given by the total bases gained in hitting (one for a single, two for a double, etc) divided by the number of at bats: (TB / AB) , where $TB := (H + h2B + 2*h3B + 3*HR)$. An SLG over 0.500 is very good.
- *On-base-plus-slugging* (OPS) combines on-base and slugging percentages to give a simple and useful estimate of overall offensive contribution. It's found by simply adding the figures: $(OBP + SLG)$. Anything above 0.900 is very good.

Doing this with the simple form of FOREACH we've been using would be annoying and hard to read — for one thing, the expressions for OBP and SLG would have to be repeated in the expression for OPS, since the full statement is evaluated together. Pig

10. We also warned you we'd wander away from it frequently — the bulk of it sits in the next chapter.

11. Here's to you, 1970 Rod Carew and 1979 Mario Mendoza

12. Although known as percentages, OBP and SLG are always given as fractions to 3 decimal places. For OBP, we're also using a slightly modified formula to reduce the number of stats to learn. It gives nearly identical results but you will notice small discrepancies with official figures

provides a fancier form of FOREACH (a *nested* FOREACH) that allows intermediate expressions:

```
bat_seasons = FILTER bat_seasons BY PA > 0 AND AB > 0;
core_stats = FOREACH bat_seasons {
    TB   = h1B + 2*h2B + 3*h3B + 4*HR;
    OBP  = 1.0f*(H + BB + HBP) / PA;
    SLG  = 1.0f*TB / AB;
    OPS  = SLG + OBP;
    GENERATE
        player_id, name_first, name_last, -- $0- $2
        year_id,   team_id,   lg_id,      -- $3- $5
        age,       G,          PA,          HBP, SH, BB, -- $6-$12
        H,          h1B,       h2B,       h3B, HR,   R,   RBI, -- $13-$19
        SLG,       OBP,       OPS,        -- $20-$22
    };
}
```

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

You'll notice that we multiplied by `1.0` while calculating `OBP` and `SLG`. If all the operands were integers, Pig would use integer arithmetic; instead of fractions between 0 and 1, the result would always be integer 0. Multiplying by the floating-point value `1.0` forces Pig to use floating-point math, preserving the fraction. Using a typecast — `SLG = (float)TB / AB` — as described below is arguably more efficient but inarguably uglier. The above is what we'd write in practice.

By the way, the filter above is sneakily doing two things. It obviously eliminates records where `PA` is equal to zero, but it also eliminates records where `PA` is `null`. (See the section “Selecting or Rejecting Records with `null` Values” (REF) above for details.)

In addition to applying arithmetic expressions and functions, there are a set of *operations* (`ORDER`, `DISTINCT`, `FOREACH`, `FILTER`, `LIMIT`) you can apply to bags within a nested FOREACH. We'll wait until the section on grouping operations to introduce their nested-foreach (“inner bag”) forms.

Formatting a String According to a Template

The `SPRINTF` function is a great tool for assembling a string for humans to look at. It uses the `printf`-style templating convention common to C and many other languages to assemble strings with consistent padding and spacing. It's best learned by seeing it in action:

```
formatted = FOREACH bat_seasons GENERATE
    SPRINTF('%4d\t%-9s %-19s\tOBP %5.3f / %-3s %-3s\t%4$012.3e',
            year_id, player_id,
```

```

CONCAT(name_first, ' ', name_last),
1.0f*(H + BB + HBP) / PA,
(year_id >= 1900 ? '.' : 'pre'),
(PA >= 450 ? 'sig' : '.')
) AS OBP_summary:chararray;

```

So you can follow along, here are some scattered lines from the results:

1954	aaronha01	Hank Aaron	OBP 0.318 / . sig	0003.183e-01
1897	ansonca01	Cap Anson	OBP 0.372 / pre sig	0003.722e-01
1970	carewro01	Rod Carew	OBP 0.407 / . .	0004.069e-01
1987	gwynnnto01	Tony Gwynn	OBP 0.446 / . sig	0004.456e-01
2007	pedrodu01	Dustin Pedroia	OBP 0.377 / . sig	0003.769e-01
1995	vanlawi01	William Van Lindingham	OBP 0.149 / . .	0001.489e-01
1941	willite01	Ted Williams	OBP 0.553 / . sig	0005.528e-01

The parts of the template are as follows:

- `%4d`: render an integer, right-aligned, in a four character slot. All the `year_id` values have exactly four characters, but if Pliny the Elder's rookie season from 43 AD showed up in our dataset, it would be padded with two spaces: ` 43`. Writing `%04d` (i.e. with a zero after the percent) causes zero-padding: `0043`.
- `\t` (backslash-t): renders a literal tab character. This is done by Pig, not in the `SPRINTF` function.
- `%-9s`: a nine-character string. Like the next field, it ...
- `%-20s`: has a minus sign, making it left-aligned. You usually want this for strings.
 - We prepared the name with a separate `CONCAT` statement and gave it a single string slot in the template, rather than using say `%-8s %11s`. In our formulation, the first and last name are separated by only one space and share the same 20-character slot. Try modifying the script to see what happens with the alternative.
 - Any value shorter than its slot width is padded to fit, either with spaces (as seen here) or with zeros (as seen in the last field). A value longer than the slot width is not truncated — it is printed at full length, shifting everything after it on the line out of place. When we chose the 19-character width, we didn't count on William Van Lindingham's corpulent cognomen contravening character caps, correspondingly corrupting columnar comparisons. Still, that only messes up Mr. Van Lindingham's line — subsequent lines are unaffected.
- `OBP`: Any literal text you care to enter just carries through. In case you're wondering, you can render a literal percent sign by writing `%%`.
- `%5.3f`: for floating point numbers, you supply two widths. The first is the width of the full slot, including the sign, the integer part, the decimal point, and the fractional part. The second number gives the width of the fractional part. A lot of scripts that

use arithmetic to format a number to three decimal places (as in the prior section) should be using SPRINTF instead.

- %-3s %-3s: strings indicating whether the season is pre-modern ($\backslash<\backslash= 1900$) and whether it is significant (≥ 450 PA). We could have used true/false, but doing it as we did here — one value tiny, the other with visual weight — makes it much easier to scan the data.
 - By inserting the / delimiter and using different phrases for each indicator, it's easy to grep for matching lines later — `grep -e '/.*sig'` — without picking up lines having 'sig' in the player id.
- %4\$09.3e: Two things to see here:
 - Each of the preceding has pulled its value from the next argument in sequence. Here, the 4\$ part of the specifier uses the value of the fourth non-template argument (the OBP) instead.
 - The remaining 012.3e part of the specifier says to use scientific notation, with three decimal places and twelve total characters. Since the strings don't reach full width, their decimal parts are padded with zeroes. When you're calculating the width of a scientific notation field, don't forget to include the *two* sign characters: one for the number and one for the exponent

We won't go any further into the details, as the SPRINTF function is well documented (REF) and examples of printf-style templating abound on the web. But this is a useful and versatile tool, and if you're able to mimic the elements used above you understand its essentials.

Assembling Literals with Complex Types

Another reason you may need the nested form of FOREACH is to assemble a complex literal. If we wanted to draw key events in a player's history — birth, death, start and end of career — on a timeline, or wanted to place the location of their birth and death on a map, it would make sense to prepare generic baskets of events and location records. We will solve this problem in a few different ways to demonstrate assembling complex types from simple fields.

Parsing a Date

Assembling Complex Types.

```
date_converted = FOREACH people {  
    beg_dt    =ToDate(CONCAT(beg_date, 'T00:00:00.000Z'));  
    end_dt    =ToDate(end_date, 'yyyy-MM-dd', '+0000');  
    birth_dt  =ToDate(SPRINTF('%s-%s-%sT00:00:00Z', birth_year, Coalesce(birth_month,1), Coalesce(birth_day,1)));  
    death_dt  =ToDate(SPRINTF('%s-%s-%sT00:00:00Z', death_year, Coalesce(death_month,1), Coalesce(death_day,1)));  
}
```

```
    GENERATE player_id, birth_dt, death_dt, beg_dt, end_dt, name_first, name_last;
};
```

One oddity of the people table's structure as it arrived to us is that the birth/death dates are given with separate fields, while the beginning/end of career dates are given as ISO date strings. We left that alone because this kind of inconsistency is the reality of data sets in practice — in fact, this is about as mild a case as you'll find. So one thing we'll have to do is pick a uniform date representation and go forward with it.

You may have heard the saying “The two hardest things in Computer Science are cache coherency and naming things”. Our nominations for the two most horrible things in Computer Science are time zones and character encoding¹³ Elsewhere you'll hear “. Our rule for Time Zones is “put it in UTC *immediately* and never speak of it again¹⁴. A final step in rendering data for an end-user interface may convert to local time, but at no point in data analysis should you tolerate anything but UTC. We're only working with dates right here, but we'll repeat that rule every chance we have in the book.

There are two and a half defensible ways to represent a date or time:

- As an **ISO 8601 Date/Time string in the UTC time zone**. It sounds scary when we say “ISO 8601”, but it's self-explanatory and you see all over the place: '2007-08-09T10:11:12Z' is an example of a time, and '2007-08-09' is an example of a date. It's compact enough to not worry about, there's little chance of it arriving in that format by accident, everything everywhere can parse it, and you can do ad-hoc manipulation of it using string functions (eg (int)SUBSTRING(end_date, 0,4) to extract a year). Use this format only if you are representing instants that come after the 1700s, only need seconds-level precision, and where human readability is more important than compactness (which we encourage).
- As an **integer number of epoch milliseconds in the UTC time zone**, which is to say as the number of elapsed milliseconds since midnight January 1st, 1970 UTC. (You may see this referred to as *UNIX time*.) It allows you to easily calculate durations, and is nearly universal as well. Its value fits nicely in an unsigned 64-bit long. We believe using fractional epoch time — e.g. 1186654272892.657 to mean 657 microseconds into the given second — is carrying the joke too far. If you care about micro- or nano-seconds, then you need to care about floating point error, and the leading part of the number consumes too much of your precision. Use this format only if you are representing instants that come after the start of the epoch; only need millisecond precision; and don't care about leap seconds.

-
13. Many people add “...and off-by-one errors” to the hardest-things list. If we are allowed to re-use the same joke, the two most horrible things in Computer Science are #1 Time Zones, #2 Character Enco, #2 Threads.ding.
 14. You can guess our rule for character encoding: “put it in UTF-8 *immediately* and never speak of it again

- A **domain representation chosen judiciously by an expert**. If neither of the above two representations will work for you then sorry: you need to get serious. Astronomers and anyone else working at century scale will likely use some form of [Julian Date](#); those working at nanosecond scale should look at [TAI](#); there are dozens of others. You'll probably have to learn things about leap seconds or sidereal times or the fluid space-time discontinuum that is the map of Time Zones, and you will wish you didn't have to. We're not going to deal with this category as it's far, far beyond the scope of the book.

In general we will leave times in their primitive data type (`long` for epoch milliseconds, `chararray` for ISO strings) until we need them to be proper `datetime` data structures. The lines above show a couple ways to create `datetime` values; here's the fuller catalog.

Epoch milliseconds are easily converted by calling `ToDate(my_epoch_millis)`. For an ISO format string with date, time and time zone, pass it as a single `chararray` string argument: `ToDate(beg_date)`. If its lacks the time-of-day or time zone part, you must fill it out first: `ToDate(CONCAT(beg_date, 'T00:00:00.000Z'))`. If the string has a non-standard format, supply two additional arguments: a template according to Java's [SimpleDateFormat](#), and unless the input has a timezone, the UTC time zone string `+0000`. For example, `ToDate(end_date, 'yyyy-MM-dd', '+0000')` demonstrates another way to parse an ISO date string: viable, but more expensive than the one-arg version.

For composite year-month-day-etc fields, create an ISO-formatted string and pass it to `ToDate`. Here's the snippet we used, in slow motion this time:

```
ToDate(
    SPRINTF('%s-%s-%sT00:00:00Z',
            birth_year, -- ISO format template
            birth_month IS NULL ? 1 : birth_month), -- if year is NULL, value will be null
            birth_day IS NULL ? 1 : birth_day)
        );
```



Apart from subtracting one epoch milliseconds from another to get a duration in milliseconds, you must *never do any date/time manipulation except through a best-in-class date library*. You can't calculate the difference of one year by adding one to the year field (which brought down Microsoft's [cloud storage product](#) on the leap day of February 29th, 2012), and you can't assume that the time difference from one minute to the next is 60 seconds (which [brought down HBase servers worldwide](#) when the leap second of 2012-06-30T23:59:60Z — note the :60 — occurred). This is no joke — companies go out of business because of mistakes like these.

Assembling a Bag

Assembling Complex Types.

```

graphable = FOREACH people {
    birth_month = Coalesce(birth_month, 1); birth_day = Coalesce(birth_day, 1);
    death_month = Coalesce(death_month, 1); death_day = Coalesce(death_day, 1);
    beg_dt    = ToDate(beg_date);
    end_dt   = ToDate('yyyy-MM-dd', end_date);
    birth_dt = ToDate(SPRINTF('%s-%s-%s', birth_year, birth_month, birth_day));
    death_dt = ToDate(SPRINTF('%s-%s-%s', death_year, death_month, death_day));
    --
    occasions = {
        ('birth', birth_year, birth_month, birth_day),
        ('death', death_year, death_month, death_day),
        ('debut', (int)SUBSTRING(beg_date,0,4), (int)SUBSTRING(beg_date,5,7), (int)SUBSTRING(beg_date,8,10)),
        ('lastg', (int)SUBSTRING(end_date,0,4), (int)SUBSTRING(end_date,5,7), (int)SUBSTRING(end_date,8,10))
    };
    --
    places = (
        (birth_dt, birth_city, birth_state, birth_country),
        (birth_dt, death_city, death_state, death_country),
        (beg_dt, null, null, null),
        (end_dt));
}

GENERATE
    player_id,
    occasions AS occasions:bag{t:(occasion:chararray, year:int, month:int, day:int)},
    places   AS places:tuple( birth:tuple(city, state, country),
                                death:tuple(city, state, country) )
;
};


```

The `occasions` intermediate alias is a bag of event tuples holding a chararray and three ints. Bags are disordered (unless you have transiently applied an explicit sorted), and so we've prefixed each event with a slug naming the occasion.

You can do this inline (non-nested FOREACH) but we wouldn't. If you find yourself with the error `Error during parsing. Encountered " "as" "AS "" at line X`, just pay for the ext

Assembling a Tuple

- how tuple is made

Specifying Schema for Complex Types

TODO: clean up

- how bag is made
- We may not have needed to write out the types — it's likely that `occasions:bag{t: (occasion, year, month, day)}` would suffice. But this is another scenario where

if you ask the question “Hey, do I need to specify the types or will Pig figure it out?” you’ve answered the question: yes, state them explicitly. The important point isn’t whether Pig will figure it out, it’s whether stupider-you at 3 am will figure it out.

- how tuple is made

Manipulating the Type of a Field

We used ‘CONCAT’ to combine players’ city, state and country of birth into a combined field without drama. But if we tried to do the same for their date of birth by writing `CONCAT(birth_year, '-', birth_month, '-', birth_day)`, Pig would throw an error: Could not infer the matching function for `org.apache.pig.builtin.CONCAT`.... You see, `CONCAT` understandably wants to consume and deliver strings, and so isn’t in the business of guessing at and fixing up types. What we need to do is coerce the `int` values — eg, 1961, a 32-bit integer — into `chararray` values — eg ‘1961’, a string of four characters. You do so using C-style typecast expression: `(chararray)birth_year`. Here it is in action:

```
birthplaces = FOREACH people GENERATE
    player_id,
    CONCAT((chararray)birth_year, '-', (chararray)birth_month, '-', (chararray)birth_day) AS birthdate;
```

In other cases you don’t need to manipulate the type going in to a function, you need to manipulate the type going out of your `FOREACH`. Here are several takes on a `FOREACH` statement to find the slugging average:

```
obp_1 = FOREACH bat_seasons {
    OBP = 1.0f * (H + BB + HBP) / PA; -- constant is a float
    GENERATE OBP; -- making OBP a float
};
-- obp_1: {OBP: float}

obp_2 = FOREACH bat_seasons {
    OBP = 1.0 * (H + BB + HBP) / PA; -- constant is a double
    GENERATE OBP; -- making OBP a double
};
-- obp_2: {OBP: double}

obp_3 = FOREACH bat_seasons {
    OBP = (float)(H + BB + HBP) / PA; -- typecast forces floating-point arithmetic
    GENERATE OBP AS OBP; -- making OBP a float
};
-- obp_3: {OBP: float}

obp_4 = FOREACH bat_seasons {
    OBP = 1.0 * (H + BB + HBP) / PA; -- constant is a double
    GENERATE OBP AS OBP:float; -- but OBP is explicitly a float
};
```

```

-- obp_4: {OBP: float}

broken = FOREACH bat_seasons {
    OBP = (H + BB + HBP) / PA;           -- all int operands means integer math and zero as result
    GENERATE OBP AS OBP:float;          -- even though OBP is explicitly a float
};

-- broken: {OBP: float}

```

The first stanza matches what was above. We wrote the literal value as `1.0f` — which signifies the `float` value 1.0 — thus giving OBP the implicit type `float` as well. In the second stanza, we instead wrote the literal value as `1.0` — type `double` — giving OBP the implicit type `double` as well. The third stanza takes a different tack: it forces floating-point math by typecasting the result as a `float`, thus also implying type `float` for the generated value¹⁵.

In the fourth stanza, the constant was given as a double. However, this time the `AS` clause specifies not just a name but an explicit type, and that takes precedence¹⁶. The fifth stanza exists just to re-prove the point that if you care about the types Pig will use, say something. Although the output type is a `float`, the intermediate expression is calculated with integer math and so all the answers are zero. Even if that worked, you'd be a chump to rely on it: use any of the preceding four stanzas instead.

Ints and Floats and Rounding, Oh My!

Another occasion for type conversion comes when you are trying to round or truncate a fractional number. The first four fields of the following statement turn the full-precision result of calculating OBP (`0.31827113`) into a result with three fractional digits (`0.318`), as OBP is usually represented.

```

rounded = FOREACH bat_seasons GENERATE
    (ROUND(1000.0f*(H + BB + HBP) / PA)) / 1000.0f AS round_and_typecast,
    ((int)(1000.0f*(H + BB + HBP) / PA)) / 1000.0f AS typecast_only,
    (FLOOR(1000.0f*(H + BB + HBP) / PA)) / 1000 AS floor_and_typecast,
    ROUND_TO( 1.0f*(H + BB + HBP) / PA, 3) AS what_we_would_use,
    SPRINTF('%5.3f', 1.0f*(H + BB + HBP) / PA) AS but_if_you_want_a_string_just_say_so,
    1.0f*(H + BB + HBP) / PA AS full_value
;

```

15. As you can see, for most of the stanzas Pig picked up the name of the intermediate expression (OBP) as the name of that field in the schema. Weirdly, the typecast in the third stanza makes the current version of Pig lose track of the name, so we chose to provide it explicitly
16. Is the intermediate result calculated using double-precision math, because it starts with a `double`, and then converted to `float`? Or is it calculated with single-precision math, because the result is a `float`? We don't know, and even if we did we wouldn't tell you. Don't resolve language edge cases by consulting the manual, resolve them by using lots of parentheses and typecasts and explicitness. If you learn fiddly rules like that — operator precedence is another case in point — there's a danger you might actually rely on them. Remember, you write code for humans to read and only incidentally for robots to run.

The `round_and_typecast` field shows a fairly common (and mildly flawed) method for chunking or partially rounding values: scale-truncate-rescale. Multiplying `0.31827113` by `1000.0f` gives a float result `318.27113`; rounding it gets an integer value `318`; rescaling by `1000.0f` gives a final result of `0.318f`, a float. The second version works mostly the same way, but has no redeeming merits. Use a typecast expression when you want to typecast, not for its side effects. This muddy formulation leads off with a story about casting things to type `int`, but only a careful ticking off of parentheses shows that we swoop in at the end and implicitly cast to float. If you want to truncate the fractional part, say so by using the function for truncating the fractional part, as the third formulation does. The `FLOOR` method uses machine numeric functions to generate the value. This is likely more efficient, and it is certainly more correct.

Floating-point arithmetic, like unicode normalization and anything cryptography, has far more complexity than anyone who wants to get things done can grasp. At some point, take time to become aware of the **built-in math functions** that are available¹⁷. You don't have to learn them, just stick the fact of their existence in the back of your head. If the folks at the IEEE have decided every computer on the planet should set aside silicon for a function to find the log of 1 plus x (`log1p`), or a function to find the remainder when dividing two numbers (`IEEEremainder`), you can bet there's a really good reason why your stupid way of doing it is some mixture of incorrect, inaccurate, or fragile.

That is why the formulation we would actually use to find a rounded number is the fourth one. It says what we mean ("round this number to three decimal places") and it draws on Java library functions built for just this purpose. The error between the `ROUND` formulation and the `ROUND_TO` formulation is almost certainly minuscule. But multiply "minuscule" by a billion records and you won't like what comes out.

Calling a User-Defined Function (UDF) from an External Package

TODO: clean up

In the section on "Extracting a Consistent Sample of Records by Key",

You can extend Pig's functionality with *User-Defined Functions* (UDFs) written in Java, Python, Ruby, Javascript and others. These have first-class functionality — almost all of Pig's native functions are actually Java UDFs that just happen to live in a builtin namespace. We'll describe how to author a UDF in a later chapter (REF), but this is a good time to learn how to call one.

The DataFu package is an collection of Pig extensions open-sourced by LinkedIn, and in our opinion everyone who uses Pig should install it. It provides the most important flavors of hash digest and checksum you need in practice, and explains how to choose

17. either as Pig built-ins, or through the Piggybank UDF library

the right one. For consistent hashing purposes, the right choice is the “Mumur 3” function¹⁸, and since we don’t need many bytes we’ll use the 32-bit flavor.

You must do two things to enable use of a UDF. First, so that pig can load the UDF’s code, call the REGISTER command with the path to the UDF’s .jar file. You only need to REGISTER a jar once, even if you’ll use more than one of its UDFs.

Second, use the DEFINE command to construct it. DEFINE takes two arguments, separated by spaces: the short name you will use to invoke the command, and the fully-qualified package name of its class (eg `datafu.pig.hash.Hasher`). Some UDFs, including the one we’re using, accept or require constructor arguments (always strings). These are passed function-call style, as shown below. There’s nothing wrong with DEFINE-ing a UDF multiple times with different constructor arguments — for example, adding a line `DEFINE DigestMD5 datafu.pig.hash.Hasher('md5');` would create a hash function that used the MD5 (REF) algorithm.

```
-- Please substitute the right path (and for citizens of the future, the right version number)
REGISTER      '/path/to/data_science_fun_pack/pig/datafu/datafu-pig/build/libs/datafu-pig-1.2.1.jar';
-- Murmur3, 32 bit version: a fast statistically smooth hash digest function
DEFINE Digest  datafu.pig.hash.Hasher('murmur3-32');

-- Prepend a hash of the player_id
keyed_seasons = FOREACH bat_seasons GENERATE Digest(player_id) AS keep_hash, *;

some_seasons  = FOREACH (
    FILTER keyed_seasons BY (SUBSTRING(keep_hash, 0, 1) == '0')
) GENERATE $0..;
```

There are three ways to accomplish this.

One is to use the REGISTER keyword, demonstrated below. This is by far the simplest option, but our least favorite. Every source file becomes contaminated by a line that is machine-dependent and may break when packages are updated.

Enabling UDFs by Importing a Macro File

Instead, we recommend you create and IMPORT a macro file containing the REGISTER and DEFINE statements. This is what we use in the sample code repo:

```
-- Paths
%DEFAULT dsfp_dir          '/path/to/data_science_fun_pack';

-- Versions; must include the leading dash when version is given
%DEFAULT datafu_version     '-1.2.1';
```

18. Those familiar with the MD5 or SHA hashes might have expected we’d use one of them. Those would work as well, but Murmur3 is faster and has superior statistical properties; for more, see the DataFu documentation. Oh and if you’re not familiar with any of the stuff we just said: don’t worry about it, just know that ‘murmur3-32’ is what you should type in.

```
%DEFAULT piggybank_version '';
%DEFAULT pigsy_version      '-2.1.0-SNAPSHOT';

REGISTER      '$dsfp_dir/pig/pig/contrib/piggybank/java/piggybank$piggybank_version.jar';
REGISTER      '$dsfp_dir/pig/datafu/datafu-pig/build/libs/datafu-pig$datafu_version.jar';
REGISTER      '$dsfp_dir/pig/pigsy/target/pigsy$pigsy_version.jar';

DEFINE Transpose  datafu.pig.util.TransposeTupleToBag();
DEFINE Digest     datafu.pig.hash.Hasher('murmur3-32');
```

First, we define a few string defaults. Making the common root path a %DEFAULT means you can override it at runtime, and simplifies the lines that follow. Parameterizing the versions makes them visible and also lets you easily toggle between versions from the commandline for smoke testing.

Next we register the jars, interpolating the paths and versions; then define the standard collection of UDFs we use. These definitions are executed for all scripts that import the file, but we were unable to detect any impact on execution time.

Enabling UDFs using Java Properties

Lastly, you can set the `pig.additional.jars` and `udf.import.list` java properties. For packages that you want to regard as being effectively built-in, this is our favorite method — but the hardest to figure out. We can't go into the details (see the Pig documentation, there are many) but we can show you how to match what we used above:

Using Pig Properties to Enable UDFs.

```
# Remove backslashes and spaces: these must sit on the same line
pig.additional.jars=\
  /path/to/data_science_fun_pack/pig/datafu/datafu-pig/build/libs/datafu-pig-1.2.1.jar:\
  /path/to/data_science_fun_pack/pig/pig/contrib/piggybank/java/piggybank.jar:\
  /path/to/data_science_fun_pack/pig/pigsy/target/pigsy-2.1.0.jar

# Remove backslashes and spaces: these also must sit on the same line
udf.import.list=\
  datafu.pig.bags:datafu.pig.hash:datafu.pig.stats:datafu.pig.sets:datafu.pig.util:\
  org.apache.pig.piggybank.evaluation:pigsy.text
```

A Quick Look into Baseball

Nate Silver calls Baseball the “perfect data set”. There are not many human-centered systems for which this comprehensive degree of detail is available, and no richer set of tables for truly demonstrating the full range of analytic patterns.

For readers who are not avid baseball fans, we provide a simple — some might say “oversimplified” — description of the sport and its key statistics. Please refer to Joseph Adler's *Baseball Hacks* (O'Reilly) or Marchi and Albert's *Analyzing Baseball Data with R* (Chapman & Hall) for more details.

The stats come in tables at multiple levels of detail. Putting people first as we like to do, the `people` table lists each player's name and personal stats such as height and weight, birth year, and so forth. It has a primary key, the `player_id`, formed from the first five letters of their last name, first two letters of their first name, and a two digit disambiguation slug. There are also primary tables for ballparks (`parks`) listing information on every stadium that has ever hosted a game and for teams (`teams`) giving every major-league team back to the birth of the game.

The core statistics table is `bat_seasons`, which gives each player's batting stats by season. (To simplify things, we only look at offensive performance.) The `player_id`, `year_id` fields form a primary key, and the `team_id` foreign key represents the team they played the most games for in a season. The `park_teams` table lists, for each team, all "home" parks they played in by season, along with the number of games and range of dates. We put "home" in quotes because technically it only signifies the team that bats last (a significant advantage), though teams nearly always play those home games at a single stadium in front of their fans. However, there are exceptions as you'll see in the next chapter (REF). The `park_id`, `team_id`, `year_id` fields form its primary key, so if a team did in fact have multiple home ballparks there will be multiple rows in the table.

There are some demonstrations where we need data with some real heft — not so much that you can't run it on a single-node cluster, but enough that parallelizing the computation becomes important. In those cases we'll go to the `games` table (100+ MB), which holds the final box score summary of every baseball game played, or to the full madness of the `events` table (1+ GB), which records every play for nearly every game back to the 1940s and before. These tables have nearly a hundred columns each in their original form. Not to carry the joke quite so far, we've pared them back to only a few dozen columns each, with only a handful seeing actual use.

We denormalized the names of players, parks and teams into some of the non-prime tables to make their records more recognizable. In many cases you'll see us carry along the name of a player, ballpark or team to make the final results more readable, even where they add extra heft to the job. We always try to show you sample code that represents the code we'd write professionally, and while we'd strip these fields from the script before it hit production, you're seeing just what we'd do in development. "Know your Data".

Acronyms and terminology

We use the following acronyms (and, coincidentally, field names) in our baseball dataset:

- G, *Games*
- PA: *Plate Appearances*, the number of completed chances to contribute offensively. For historical reasons, some stats use a restricted subset of plate appearances called AB (At Bats). You should generally prefer PA to AB, and can pretend they represent the same concept.
- H: *Hits*, either singles (`h1B`), doubles (`h2B`), triples (`h3B`) or home runs (`HR`)

- BB: *Walks*, pitcher presented too many unsuitable pitches
- HBP: *Hit by Pitch*, like a walk but more painful
- OBP: *On-base Percentage*, indicates effectiveness at becoming a potential run
- SLG: *Slugging Percentage*, indicates effectiveness at converting potential runs into runs
- OPS: *On-base-plus-Slugging*, a reasonable estimate of overall offensive contribution

For those who consider sporting events to be the dull province of jocks, holding no interest at all: when we say the “On-Base Percentage” is a simple matter of finding $(H + BB + HBP) / AB$, just trust us that (a) it’s a useful statistic; (b) that’s how you find its value; and then (c) pretend it’s the kind of numbers-in-a-table example abstracted from the real world that many books use.

The Rules and Goals

Major League Baseball teams play a game nearly every single day from the start of April to the end of September (currently, 162 per season). The team on offense sends its players to bat in order, with the goal of having its players reach base and advance the full way around the diamond. Each time a player makes it all the way to home, their team scores a run, and at the end of the game, the team with the most runs wins. We count these events as G (games), PA (plate appearances on offense) and R (runs).

The best way to reach base is by hitting the ball back to the fielders and reaching base safely before they can retrieve the ball and chase you down — a hit (H). You can also reach base on a *walk* (BB) if the pitcher presents too many unsuitable pitches, or from a *hit by pitch* (HBP) which is like a walk but more painful. You advance on the basepaths when your teammates hit the ball or reach base; the reason a hit is valuable is that you can advance as many bases as you can run in time. Most hits are singles (h1B), stopping safely at first base. Even better are doubles (h2B: two bases), triples (h3B: three bases, which are rare and require very fast running), or home runs (HR: reaching all the way home, usually by clobbering the ball out of the park).

Your goal as a batter is both becomes a potential run and helps to convert players on base into runs. If the batter does not reach base it counts as an out, and after three outs, all the players on base lose their chance to score and the other team comes to bat. (This threshold dynamic is what makes a baseball game exciting: the outcome of a single pitch could swing the score by several points and continue the offensive campaign, or it could squander the scoring potential of a brilliant offensive position.)

Performance Metrics

The above are all “counting stats”, and generally the more games the more hits and runs and so forth. For estimating performance and comparing players, it’s better to use “rate stats” normalized against plate appearances.

On-base percentage (OBP) indicates how well the player meets offensive goal #1: get on base, thus becoming a potential run and *not* consuming a precious out. It is given as the fraction of plate appearances that are successful: $((H + BB + HBP) / PA)$ ¹⁹. An OBP over 0.400 is very good (better than 95% of significant seasons).

Slugging Percentage (SLG) indicates how well the player meets offensive goal #2: advance the runners on base, thus converting potential runs into points towards victory. It is given by the total bases gained in hitting (one for a single, two for a double, etc) divided by the number of at bats: $((H + h2B + 2*h3B + 3*HR) / AB)$. An SLG over 0.500 is very good.

On-base-plus-slugging (OPS) combines on-base and slugging percentages to give a simple and useful estimate of overall offensive contribution. It's found by simply adding the figures: (OBP + SLG). Anything above 0.900 is very good.

Just as a professional mechanic has an assortment of specialized and powerful tools, modern baseball analysis uses statistics significantly more nuanced than these. But when it comes time to hang a picture, they use the same hammer as the rest of us. You might think that using the on-base, slugging, and OPS figures to estimate overall performance is a simplification we made for you. In fact, these are quite actionable metrics that analysts will reach for when they want to hang a sketch that anyone can interpret.

Operations that Break One Table Into Many

Directing Data Conditionally into Multiple Data Flows (**SPLIT**)

The careers table gives the number of times each player was elected to the All-Star game (indicating extraordinary performance during a season) and whether they were elected to the Hall of Fame (indicating a truly exceptional career).

Demonstration in Pig

Separating those records into different data flows isn't straightforward in map/reduce, but it's very natural using Pig's SPLIT operation.

```
SPLIT bat_career
    INTO hof      IF hofYear > 0, -- the '> 0' eliminates both NULLs and 0s
    INTO allstar  IF G_allstar > 0,
    INTO neither  OTHERWISE
;
STORE hof      INTO '/data/out/baseball/hof_careers';
```

19. Although known as percentages, OBP and SLG are always given as fractions to 3 decimal places. For OBP, we're also using a slightly modified formula to reduce the number of stats to learn. It gives nearly identical results but you will notice small discrepancies with official figures

```
STORE allstar INTO '/data/out/baseball/allstar_careers';
STORE neither INTO '/data/out/baseball/neither_careers';
```

The SPLIT operator does not short-circuit: every record is tested against every condition, and so a player who is both a hall-of-famer and an allstar will be written into both files.

The most natural use of the SPLIT operator is when you really do require divergent processing flows. In the next chapter, you'll use a JOIN LEFT OUTER to geolocate (derive longitude and latitude from place name) records. That method is susceptible to missing matches, and so in practice a next step might be to apply a fancier but more costly geolocation tool. This is a strategy that arises often in advanced machine learning applications: run a first pass with a cheap algorithm that can estimate its error rate; isolate the low-confidence results for harder processing; then reunite the whole dataset.

The syntax of the SPLIT command does not have an equals sign to the left of it; the new table aliases are created in its body.

```
SPLIT players_geoloced_some INTO
    players_non_geoloced_us IF ((IsNull(lng) OR IsNull(lat)) AND (country_id == "US")),
    players_non_geoloced_fo IF ((IsNull(lng) OR IsNull(lat)),
    players_geoloced_a OTHERWISE;

-- ... Pretend we're applying a more costly / higher quality geolocation tool, rather than just se
players_geoloced_b = FOREACH players_non_geoloced_us GENERATE
    player_id..country_id,
    FLATTEN((Disney,land)) as (lng, lat);
-- ... And again, pretend we are not just sending all non-us to the Eiffel Tower.
players_geoloced_c = FOREACH players_non_geoloced_us GENERATE
    player_id..country_id,
    FLATTEN((Eiffel,tower)) as (lng, lat);

Players_geoloced = UNION alloftheabove;
```

Splitting into files by key by using a Pig Storefunc UDF

One reason you might find yourself splitting a table is to create multiple files on disk according to some key.

If instead you're looking to partition directly into files named for a key, use the multi-storage storefunc from the Piggybank UDF collection. As opposed to SPLIT, each record goes into exactly one file. Here is how to partition player seasons by primary team:

There might be many reasons to do this splitting, but one of the best is to accomplish the equivalent of what traditional database admins call “vertical partitioning”. You are still free to access the table as a whole, but in cases where one field is over and over again used to subset the data, the filtering can be done without ever even accessing the excluded data. Modern databases have this feature built-in and will apply it on your behalf

based on the query, but our application of it here is purely ad-hoc. You will need to specify the subset of files yourself at load time to take advantage of the filtering.

```
bat_season = LOAD 'bat_season' AS (...);  
STORE bat_season INTO '/data/out/baseball/seasons_by_team' USING MultiStorage('/data/out/baseball/  
STORE ... multistorage;  
  
STORE events INTO '$out_dir/evs_away'  
    USING MultiStorage('$out_dir/evs_away','5'); -- field 5: away_team_id  
STORE events INTO '$out_dir/evs_home'  
    USING MultiStorage('$out_dir/evs_home','6'); -- field 6: home_team_id
```

This script will run a map-only job with 9 map tasks (assuming 1GB+ of data and a 128MB block size). With MultiStorage, all Boston Red Sox (team id BOS) home games that come from say the fifth map task will go into `$out_dir/evs_home/BOS/part-m-00004` (contrast that to the normal case of `$out_dir/evs_home/part-m-00004`). Each map task would write its records into the sub directory named for the team with the `part-m-` file named for its taskid index.

Since most teams appear within each input split, each subdirectory will have a full set of part-m-00000 through part-m-00008 files. In our runs, we ended up with XXX output files — not catastrophic, but (a) against best practices, (b) annoying to administer, (c) the cause of either nonlocal map tasks (if splits are combined) or proliferation of downstream map tasks (if splits are not combined). The methods of (REF) “Cleaning up Many Small Files” would work, but you’ll need to run a cleanup job per team. Better by far is to precede the `STORE USING MultiStorage` step with a `GROUP BY team_id`. We’ll learn all about grouping next chapter, but its use should be clear enough: all of each team’s events will be sent to a common reducer; as long as the Pig `pig.output.lazy` option is set, the other reducers will not output files.

```
events_by_away = FOREACH (GROUP events BY away_team_id) GENERATE FLATTEN(events);  
events_by_home = FOREACH (GROUP events BY home_team_id) GENERATE FLATTEN(events);  
STORE events_by_away INTO '$out_dir/evs_away-g'  
    USING MultiStorage('$out_dir/evs_away-g','5'); -- field 5: away_team_id  
STORE events_by_home INTO '$out_dir/evs_home-g'  
    USING MultiStorage('$out_dir/evs_home-g','6'); -- field 6: home_team_id
```

The output has a directory for each key, and within directory that the same part-NNNNN files of any map-reduce job.

This means the count of output files is the number of keys times the number of output slots, which can lead to severe many small files problem. As mentioned in Chapter 3 (REF), many small files is Not Good. If you precede the `STORE` operation by a `GROUP BY` on the key, the reducer guarantee provides that each subdirectory will only have one output file.

Splitting a Table into Uniform Chunks

We won't go into much detail, but one final set of patterns is to split a table into uniform chunks. If you don't need the chunks to be exactly sized, you can apply a final ORDER BY operation on a uniformly-distributed key — see the section on "Shuffling the Records in a Table" in the next chapter (REF).

To split into chunks with an exact number of lines, first use RANK to number each line, then prepare a chunk key using the line number modulo the chunk size, and store into chunks using MultiStorage. Since the rank operation's reducers number their records sequentially, only a few reducers are involved with each chunk, and so you won't hit the small files problem. Splitting a table into blocks of fixed *size* is naturally provided by the HDFS block size parameter, but we're not aware of a good way to do so explicitly.

An ORDER BY statement with parallelism forced to (output size / desired chunk size) will give you *roughly* uniform chunks,

```
SET DEFAULT_PARALLEL 3;
%DDEFAULT chunk_size 10000;

-- Supply enough keys to rank to ensure a stable sorting
bat_seasons_ranked = RANK bat_seasons BY (player_id, year_id)
bat_seasons_chunked = FOREACH (bat_seasons_ranked) GENERATE
    SPRINTF("%03d", FLOOR(rank/$chunk_size)) AS chunk_key, player_id...;

-- Writes the chunk key into the file, like it or not.
STORE bat_seasons_chunked INTO '$out_dir/bat_seasons_chunked'
    USING MultiStorage('$out_dir/bat_seasons_chunked','0'); -- field 0: chunk_key
```

Note that in current versions of Pig, the RANK operator forces parallelism one. If that's unacceptable, we'll quickly sketch a final alternative but send you to the sample code for details. You can instead use RANK on the map side modulo the *number* of chunks, group on that and store with MultiStorage. This will, however, have non-uniformity in actual chunk sizes of about the number of map-tasks — the final lines of each map task are more likely to short-change the higher-numbered chunks. On the upside, the final chunk isn't shorter than the rest (as it is with the prior method or the unix split command).

```
%DEFAULT n_chunks 8;

bats_ranked_m = FOREACH (RANK bat_seasons) GENERATE
    MOD(rank, $n_chunks) AS chunk_key, player_id...;
bats_chunked_m = FOREACH (GROUP bats_ranked_m BY chunk_key)
    GENERATE FLATTEN(bats_ranked_m);
STORE bats_chunked_m INTO '$out_dir/bats_chunked_m'
    USING MultiStorage('$out_dir/bat_seasons_chunked','0'); -- field 0: chunk_key
```

With no sort key fields, it's done on the map side (avoiding the single-reducer drawback of RANK)

Operations that Treat the Union of Several Tables as One

The counterpart to splitting a table into pieces is to treat many pieces as a single table. This really only makes sense when all those pieces have the same schema, so that's the only case we'll handle here.

Treat Several Pig Relation Tables as a Single Table (Stacking Rowsets)

In Pig, you can rejoin several pipelines using the UNION operation. The tables we've been using so far cover only batting stats; there are another set of tables covering stats for pitchers, and in rare cases a player may only appear in one or the other. To find the name and id of all players that appear in either table, we can project the fields we want (earning a uniform schema) and then unify the two streams:

Union Treats Several Tables as a Single Table.

```
bat_career = LOAD '/data/rawd/baseball/sports/bat_career AS (...);  
pit_career = LOAD '/data/rawd/baseball/sports/pit_career AS (...);  
bat_names = FOREACH bat_career GENERATE player_id, nameFirst, nameLast;  
pit_names = FOREACH pit_career GENERATE player_id, nameFirst, nameLast;  
names_in_both = UNION bat_names, pit_names;  
player_names = DISTINCT names_in_both;
```

Note that this is not a Join (which requires a reduce, and changes the schema of the records) — this is more like stacking one table atop another, making no changes to the records (schema or otherwise) and does not require a reduce.

A common use of the UNION statement comes in *symmetrizing* a relationship. For example, each line in the games table describes in a sense two game outcomes: one for the home team and one for the away team. We might reasonably want to prepare another table that listed game *outcomes*: game_id, team, opponent, team's home/away position, team's score, opponent's score. The game between BAL playing at BOS on XXX (final score BOS Y, BAL Z) would get two lines: GAMEIDXXX BOS BAL 1 Y Z and GAMEID BAL BOS 0 Z Y.

```
games_a = FOREACH games GENERATE  
    year_id, home_team_id AS team,  
    home_runs_ct AS runs_for, away_runs_ct AS runs_against, 1 AS is_home:int;  
  
games_b = FOREACH games GENERATE  
    away_team_id AS team,      year_id,  
    away_runs_ct AS runs_for, home_runs_ct AS runs_against, 0 AS is_home:int;  
  
team_scores = UNION games_a, games_b;  
  
DESCRIBE team_scores;  
--   team_scores: {team: chararray,year_id: int,runs_for: int,runs_against: int,is_home: int}
```

The `UNION` operation does not remove duplicate rows as a set-wise union would. It simply tacks one table onto the end of the other, and so the last line eliminates those duplicates — more on `DISTINCT` in the next chapter (REF). The `UNION` operation also does not provide any guarantees on ordering of rows. Some SQL users may fall into the trap of doing a `UNION`-then-`GROUP` to combine multiple tables. This is terrible in several ways, and you should instead use the `COGROUP` operation — see the Won-Loss Record example in the next chapter (REF).



The `UNION` operator is easy to over-use. For one example, in the next chapter we'll extend the first part of this code to prepare win-loss statistics by team. A plausible first guess would be to follow the `UNION` statement above with a `GROUP` statement, but a much better approach would use a `COGROUP` instead (both operators are explained in the next chapter). The `UNION` statement is mostly harmless but fairly rare in use; give it a second look any time you find yourself writing it in to a script.

Clean Up Many Small Files by Merging into Fewer Files

The Many Small Files problem is so pernicious because Hadoop natively assigns each mapper to only one file, and so a normal mapper-only job can only *increase* the number of files. We know of two ways to reorganize the records of a table into fewer files.

One is to perform a final `ORDER BY` operation ²⁰. Since this gives the side benefit of allowing certain optimized join operations, we like to do this for “gold” datasets that will be used by many future jobs.

Sorting is a fairly expensive operation, though; luckily, Pig can do this reasonably well with a mapper-only job by setting the `pig.splitCombination` configuration to true and setting `pig.maxCombinedSplitSize` to the size of the input divided by the number of files you'd like to produce.

```
set pig.splitCombination true;
set pig.maxCombinedSplitSize 2100100100;
```

The `maxCombinedSplitSize` should be much larger than the HDFS block size so that blocks are fully used. Also note the old sailor's trick in the last line — since there's no essential difference between 2 billion bytes, 2 gigabytes, or a number nearby, the value `2100100100` is much easier to read accurately than `2000000000` or `2147483648`.

²⁰. The tuning chapter (REF) tells you why you might want to increase the HDFS block size for truly huge dataset, and why you might not want to do so

The operations in this chapter (except where noted) do not require a reduce on their own, which makes them very efficient. The really interesting applications, however, come when we put data into context, which is the subject of the next chapter.

Analytic Patterns: Grouping Operations

Introduction

Grouping Records into a Bag by Key

The GROUP BY operation is at the heart of every structural operation. It's a one-liner in Pig to collect all the stadiums a team has played for in its history:

```
park_teams_g = GROUP park_teams BY team_id;
```

The result of a GROUP BY operation is always a field called *group*, followed by one field per grouped table, each named for the table it came from. The shape of the *group* field depends on whether you specified one or many group keys. If you specified a single key, the new *group* field is a scalar with the same schema. If you specified multiple keys, the new *group* field is a tuple whose schema corresponds to those keys. In this example, we grouped on the `team_id` chararray, and so the *group* field is a scalar chararray as well. In a moment we'll group on `year_id` and `team_id`, and so the *group* field would have schema `group:tuple(year_id:int, team_id:chararray)`.

Each of the following fields is a bag, holding tuples whose schema matches the corresponding table. Notice that the name we used to refer to the *table* is now also the name for a *field*. This will confuse you at first, but soon become natural, especially if you use DESCRIBE liberally:

```
DESCRIBE park_teams_g;
-- park_teams_g: {
--   group: chararray,
--   park_teams: {
--     ( park_id: chararray, team_id: chararray, year_id: long,
--       beg_date: chararray, end_date: chararray, n_games: long ) } }
```

Notice that the *full record* is kept, even including the keys:

```
(ALT,{{(ALT01,ALT,1884,1884-04-30,1884-05-31,18)})}  
(ANA,{{(ANA01,ANA,2001,2001-04-10,2001-10-07,81),(ANA01,ANA,2010,2010-04-05,2010-09-29,81),...}})
```

To eliminate the redundant data, you'll almost always immediately project using a FOREACH. Pig allows you to put the GROUP statement inline:

```
team_pkryr_pairs = FOREACH (GROUP park_teams BY team_id) GENERATE  
    group AS team_id, park_teams.(park_id, year_id);  
-- (ALT,{{(ALT01,1884)})}  
-- (ANA,{{(ANA01,2001),(ANA01,2010),(ANA01,2002),...}})
```

Notice the `park_teams.(park_id,year_id)` form, which gives us a bag of `(park_id,year_id)` pairs. Using `park_teams.park_id`, `park_teams.year_id` instead gives two bags, one with `park_id` tuples and one with `year_id` tuples:

```
team_pkryr_bags = FOREACH (GROUP park_teams BY team_id) GENERATE  
    group AS team_id, park_teams.park_id, park_teams.year_id;  
-- (ALT, {{(ALT01)}}, {{(1884)}})  
-- (ANA, {{(ANA01),(ANA01),(ANA01),...}}, {{(2001),(2010),(2002),...}})  
  
DESCRIBE team_pkryr_pairs;  
-- team_parks: { team_id: chararray, { (park_id: chararray, year_id: long) } }  
  
DESCRIBE team_pkryr_bags;  
-- team_parks: { team_id: chararray, { (park_id: chararray) }, { (year_id: long) } }
```

You can group on multiple fields. For each team and year, we can find the park(s) that team called home:

```
team_yr_parks_g = GROUP park_teams BY (year_id, team_id);
```

The first field is still called *group*, but it's now a tuple.

```
DESCRIBE team_yr_parks_g;  
-- team_yr_parks_g: {  
--     group: (year_id: long,team_id: chararray),  
--     park_teams: {(park_id: chararray, team_id: chararray, year_id: long, ...)}{}
```

Our FOREACH statement now looks a bit different:

```
team_yr_parks = FOREACH(GROUP park_teams BY (year_id, team_id)) GENERATE  
    group.team_id, park_teams.park_id;  
--  
=> LIMIT team_yr_parks 4; DUMP @;  
--   (BS1,{{(BOS01),(NYC01)}}  
--   (CH1,{{(NYC01),(CHI01)}}  
--   (CL1,{{(CIN01),(CLE01)}}  
--   (FW1,{{(FOR01)}})
```

Pattern in Use

- *Where You'll Use It* — Rarely on its own, but otherwise everywhere

- *Standard Snippet* — FOREACH (GROUP recs BY (key1, key2)) GENERATE group.key1, group.key2, recs AS bag_of_recs_records;
- *Hello, SQL Users* — Similar to the windowed functionality supplied by high-end SQL databases. MySQL, PostgreSQL, etc don't have similar functionality.
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values
- *Records* — Output is group, bag of records, with record contents within the bag unchanged.
- *Data Flow* — Map & Reduce

Counting Occurrences of a Key

The typical reason to group records is to operate on the bag of values it forms, and that's how we'll spend much of this chapter — the data bag is a very powerful concept. Let's take a quickie tour of what we can do to a group; afterwards we'll see the internals of how a group works before moving on to its broader applications.

You'll notice from the result of the last query that sometimes a team has more than one "home" stadium in a season. That's a bit unexpected, but on consideration teams occasionally face stadium repairs or late-season makeups for cancelled games. But cases where there were even three home parks should be quite rare. Let's confirm our feel for the data using COUNT_STAR, which counts all elements of a bag:

```
team_n_parks = FOREACH (GROUP park_teams BY (team_id,year_id)) GENERATE
    group.team_id, COUNT_STAR(park_teams) AS n_parks;
vagabonds = FILTER team_n_parks BY n_parks >= 3;

DUMP vagabonds;
(CL4,7)
(CLE,5)
(WS3,4)
(CLE,3)
(DET,3)
...
```

Always, always look through the data and seek *second stories*.

Our script is reporting that CL4 (the Cleveland Spiders) called seven (!) different stadiums home during a season. Is this some weirdness in the data? Could we possibly have messed up this three-line script? Or is it really the case that some teams have had four, five, even seven home stadiums? This demands a closer look.

Pattern in Use

- *Where You'll Use It* — Anywhere you're summarizing counts

- *Standard Snippet* — FOREACH (GROUP recs BY mykey) GENERATE group AS my key, COUNT_STAR(recs) AS ct;
- *Hello, SQL Users* — SELECT key, COUNT(*) as CT from recs GROUP BY key;. Remember: COUNT_STAR(recs), not COUNT(*) .
- *Important to Know* — See “Pattern in Use” for Aggregate Functions, below (REF)
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values
- *Records* — Output is mykey, ct:long
- *Data Flow* — Map, Combiner & Reduce; combiners very effective unless cardinality extremely high

Representing a Collection of Values with a Delimited String

Let’s keep the count of parks, but also list the parks themselves for inspection. We could keep dumping the values in Pig’s oddball output format, but this is a good opportunity to introduce a very useful pattern: denormalizing a collection of values into a single delimited field.

The format Pig uses to dump bags and tuples to disk wastes characters and is not safe to use in general: any string containing a comma or bracket will cause its record to be mis-interpreted. For simple data structures such as a list, we are better off concatenating the values together using a delimiter: a character with no other meaning that does not appear in any of the values. This preserves the rows-and-columns representation of the table that Pig handles best. It also lets us keep using the oh-so-simple TSV format for interchange with Excel, cut and other commandline tools, and later runs of Pig itself. Storing data this way means we do have to pack and unpack the value ourselves, which is an added burden when we need access to the array members. But if accessing the list contents is less frequent this can act as a positive feature: we can move the field around as a simple string and only pay the cost of constructing the full data structure when necessary.

The BagToString function will serialize a bag of values into a single delimited field as follows:

```
team_year_w_parks = FOREACH (GROUP park_teams BY (team_id, year_id)) GENERATE
    group.team_id,
    COUNT_STAR(park_teams) AS n_parks,
    BagToString(park_teams.park_id, '^') AS park_ids;
-- (ALT,1,ALT01)
-- (ANA,1,ANA01)
-- ...
-- (CL4,7,CHI08^CLE05^CLL01^PHI09^ROC02^ROC03^STL05)
```

This script outputs four fields — park_id, year, count of stadiums, and the names of the stadiums used separated by a ^ caret delimiter. Like colon :, comma ,, and slash /, it doesn't need to be escaped at the commandline; like those and semicolon ;, pipe |, and bang !, it is visually lightweight and can be avoided within a value. Don't use the wrong delimiter for addresses ("Fargo, ND"), dates ("2014-08-08T12:34:56+00:00"), paths (/tmp/foo) or unsanitized free text (It's a girl! ^_^ \m/ |:-)). If you are considering the use of quoting or escaping to make your strings delimiter safe, you're getting carried away. Stop, step away from the delimiter, and see "Representing a Complex Data Structure as a JSON-encoded String" (REF) below.

Since the park ids are formed from the first characters of the city name, we can recognize that the Spiders' home fields include two stadiums in Cleveland plus "home" stadiums in Philadelphia, Rochester, St. Louis, and Chicago. These aren't close enough to be alternatives in case of repairs, and 1898 baseball did not call for publicity tours. Were they rotating among these fields, or just spending a day or so at each? Let's see how many were played at each stadium.

Pattern in Use

- *Where You'll Use It* — Creating a URL for a batch request. Hiding a list you don't always want to deserialize. Writing a table in a format that will work everywhere.
- *Standard Snippet* — FOREACH (GROUP recs BY key) GENERATE group AS mykey, BagToString(recs, '|') AS recs_list;
- *Hello, SQL Users* — Similar to GROUP_CONCAT, but you prepare the input bag first; no fiddly in-line DISTINCT calls.
- *Important to Know* — Be careful with your choice of delimiter. Keep it simple. Don't stringify huge groups.
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values
- *Records* — Output is mykey, recs_list:chararray
- *Data Flow* — Map & Reduce; no real data reduction or explosion as assumedly you're turning all the data into strings.

Representing a Complex Data Structure with a Delimited String

Instead of serializing the simple list of park ids we had before, we'd now like to prepare and serialize the collection of (park id, number of games) pairs. We can handle this by using two delimiters: one for separating list elements and one for delimiting its contents. (This is also how you would handle an object with simple attribute-value pairs such as a hash map.)

```

team_year_w_pkgs = FOREACH (GROUP park_teams BY (team_id,year_id)) {
    pty_ordered      = ORDER park_teams BY n_games DESC;
    pk_ng_pairs      = FOREACH pty_ordered GENERATE
        CONCAT(park_id, ':', (chararray)n_games) AS pk_ng_pair;
    GENERATE group.team_id, group.year_id,
        COUNT_STAR(park_teams) AS n_parks,
        BagToString(pk_ng_pairs,'|') AS pk_ngs;
};

}

```

Whoa, a few new things going on here. We've snuck the `ORDER BY` statement into a few previous examples even though it won't be covered until later in the chapter (REF), but always as a full-table operator. Here we're using it within the body of a `FOREACH` to sort each bag locally, rather than as a total sort of the whole table. One nice thing about this `ORDER BY`: it's essentially free, as Pig just instructs Hadoop to do a secondary-sort on the data as it lands on the reducer. So there's no reason not to make the data easier to read.

After the `ORDER BY` statement, we use a *nested* `FOREACH` to staple each park onto the number of games at that park, delimited with a colon. (Along the way you'll see we also typecast the `n_games` value, since the `CONCAT` method expects a `chararray`.) The final `GENERATE` line creates records naming the team, the count of parks, and the list of park-usages pairs:

```

ALT 1 ALT01:18
ANA 1 ANA01:82
...
CL4 7 CLE05:40|PHI09:9|STL05:2|ROC02:2|CLL01:2|CHI08:1|ROC03:1

```

Out of 156 games that season, the Spiders played only 42 in Cleveland. Between the 15 “home” games in other cities, and their *ninety-nine* away games, they spent nearly three-quarters of their season on the road.

The [Baseball Library Chronology](#) sheds some light. It turns out that labor problems prevented play at their home or any other stadium in Cleveland for a stretch of time, and so they relocated to Philadelphia while that went on. What's more, on June 19th police *arrested the entire team* during a home game¹ for violating the Sunday “blue laws”². Little wonder the Spiders decided to take their talents away from Cleveland! The following year they played 50 straight on the road, won fewer than 13% of their games overall (20-134, the worst single-season record ever) and immediately disbanded at season's end.

1. The Baseball Library Chronology does note that “not so coincidentally, the Spiders had just scored to go ahead 4-3, so the arrests assured Cleveland of a victory.” Sounds like the officers, not devoid of hometown pride, might have enjoyed a few innings of the game first.
2. As late as 1967, selling cookery on Sunday in Ohio was still [enough to get you convicted](#)

Pattern in Use

See Previous.

Representing a Complex Data Structure with a JSON-encoded String

So the result for the Spiders isn't a mistake. Is the team a sole anomalous outlier, or are there other cases, less extreme but similar? The Spiders' season stands out for at least these three reasons: an unusual number of alternate parks; "home" games played in other cities; and a pre-modern (1898) setting. So let's include a field for the city (we'll take the first three characters of the park id to represent the city name) and not throw away the field for year.

```

-- Prepare the city field
pktm_city      = FOREACH park_teams GENERATE
  team_id, year_id, park_id, n_games,
  SUBSTRING(park_id, 0,3) AS city;

-- First grouping: stats about each city of residence
pktm_stats = FOREACH (GROUP pktm_city BY (team_id, year_id, city)) {
  pty_ordered    = ORDER   pktm_city BY n_games DESC;
  pk_ct_pairs   = FOREACH pty_ordered GENERATE CONCAT(park_id, ':', (chararray)n_games);
  GENERATE
    group.team_id,
    group.year_id,
    group.city              AS city,
    COUNT_STAR(pktm_city)   AS n_parks,
    SUM(pktm_city.n_games)  AS n_city_games,
    MAX(pktm_city.n_games)  AS max_in_city,
    BagToString(pk_ct_pairs, '|') AS parks
  ;
};


```

The records we're forming are significantly more complex this time. With fields of numbers or constrained categorical values, stapling together delimited values is a fine approach. But when fields become this complex, or when there's any danger of stray delimiters sneaking into the record, if you're going to stick with TSV you are better off using JSON encoding to serialize the field. It's a bit more heavyweight but nearly as portable, and it happily bundles complex structures and special characters to hide within TSV files.³

```
-- Next, assemble full picture:
farhome_gms = FOREACH (GROUP pktm_stats BY (team_id, year_id)) {
    pty ordered    = ORDER  pktm stats BY n city games DESC;
```

3. And if neither JSON nor simple-delimiter is appropriate, use Parquet or Trevni, big-data optimized formats that support complex data structures. As we'll explain in chapter (REF), those are your three choices: TSV with delimited fields; TSV with JSON fields or JSON lines on their own; or Parquet/Trevni. We don't recommend anything further.

```

city_pairs    = FOREACH pty_ordered GENERATE CONCAT(city, ':', (chararray)n_city_games);
n_home_gms   = SUM(pktd_stats.n_city_games);
n_main_city   = MAX(pktd_stats.n_city_games);
n_main_park   = MAX(pktd_stats.max_in_city);
-- a nice trick: a string vs a blank makes it easy to scan the data for patterns:
is_modern     = (group.year_id >= 1905 ? 'mod' : NULL);
--
GENERATE group.team_id, group.year_id,
        is_modern                  AS is_modern,
        n_home_gms                 AS n_home_gms,
        n_home_gms - n_main_city   AS n_farhome_gms,
        n_home_gms - n_main_park   AS n_althome_games,
        COUNT_STAR(pktd_stats)     AS n_cities,
        BagToString(city_pairs,'|') AS cities,
        BagToString(pktd_stats.parks,'|') AS parks
        ;
};

farhome_gms = ORDER farhome_gms BY n_cities DESC, n_farhome_gms DESC;

```

Here's a sample of the output:

JSON-formatted Values.

CL4 1898	57 17 17 6	CLE:40 PHI:9 ROC:3 STL:2 CLL:2 CHI:1	CLE05:40 PHI09:9 ROC02:2...
CLE 1902	65 5 5 5	CLE:60 FOR:2 COL:1 CAN:1 DAY:1	CLE05:60 FOR03:2 COL03:1...
...			
MON 2003 mod	81 22 22 2	MON:59 SJU:22	MON02:59 SJU01:22
MON 2004 mod	80 21 21 2	MON:59 SJU:21	MON02:59 SJU01:21
...			
CHA 1969 mod	81 11 11 2	CHI:70 MIL:11	CHI10:70 MIL05:11
CHA 1968 mod	81 9 9 2	CHI:72 MIL:9	CHI10:72 MIL05:9
BRO 1957 mod	77 8 8 2	NYC:69 JER:8	NYC15:69 JER02:8

Pattern in Use

- *Where You'll Use It* — Creating the POST body for a json batch request. Hiding a complex value you don't always want to deserialize. Writing a table in a format that will work everywhere. Creating a string free of non-keyboard characters.
- *Standard Snippet* — FOREACH (GROUP recs BY key) GENERATE group AS mykey, Jsonify(recs) AS recs_json;
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values
- *Records* — Output is mykey, recs_json:chararray
- *Data Flow* — Map & Reduce; mild data expansion as JSON repeats the sub-field names on each row.

Does God Hate Cleveland?

Probably. But are the Spiders a particularly anomalous exhibition? No. Considered against the teams of their era, they look much more normal. In the early days baseball was still literally getting its act together and teams hopped around frequently. Since 1905, no team has seen home bases in three cities, and the three cases where a team spent any significant time in an alternate city each tell a notable story.

In 2003 and 2004, les pauvres Montreal Expos were sentenced to play 22 “home” games in San Juan (Puerto Rico) and only 59 back in Montreal. The rudderless franchise had been sold back to the league itself and was being shopped around in preparation for a move to Washington, DC. With no real stars, no home-town enthusiasm, and no future in Montreal, MLB took the opportunity to build its burgeoning fanbase in Latin America and so deployed the team to Puerto Rico part-time. The 1968-1969 Chicago White Sox (CHA) were similarly nation-building in Milwaukee; the owner of the 1956-1957 Brooklyn Dodgers slipped them away for a stint in New Jersey in order to pressure Brooklyn for a new stadium.

You won’t always want to read a second story to the end as we have here, but it’s important to at least identify unusual features of your data set — they may turn out to explain more than you’d think.



In traditional analysis with sampled data, edge cases undermine the data, presenting the spectre of a non-representative sample or biased result. In big data analysis on comprehensive data, the edge cases *prove* the data. Here’s what we mean. Since 1904, only a very few teams have multiple home stadiums, and no team has had more than two home stadiums in a season. Home-field advantage gives a significant edge: the home team plays the deciding half of the final inning, their roster is constructed to take advantage of the ballpark’s layout, and players get to eat home-cooked meals, enjoy the cheers of encouraging fans, and spend a stretch of time in one location. The Spiders and Les Expos and a few others enjoyed only part of those advantages. XX % of our dataset is pre-modern and Y% had six or more home games in multiple cities.

With a data set this small there’s no good way to control for these unusual circumstances, and so they represent outliers that taint our results. With a large and comprehensive data set those small fractions would represent analyzable populations of their own. With millions of seasons, we could conceivably baseline the jet-powered computer-optimized schedules of the present against the night-train wanderjahr of Cleveland Spiders and other early teams.

Group and Aggregate

Some of the happiest moments you can have analyzing a massive data set come when you are able to make it a slightly less-massive data set. Aggregate functions — ones that turn the whole of a group into a scalar value — are the best path to this joy.

Aggregate Statistics of a Group

In the previous chapter, we used each player's seasonal counting stats — hits, home runs, and so forth — to estimate seasonal rate stats — how well they get on base (OPS), how well they clear the bases (SLG) and an overall estimate of offensive performance (OBP). But since we were focused on pipeline operations, we only did so on a season-by-season basis. The group-and-aggregate pattern lets us combine those seasonal stats in order to characterize each player's career.

Aggregate Statistics of a Group.

```
bat_careers = FOREACH (GROUP bat_seasons BY player_id) {
    totG  = SUM(bat_seasons.G);
    totPA = SUM(bat_seasons.PA);  totAB  = SUM(bat_seasons.AB);
    totHBP = SUM(bat_seasons.HBP); totSH  = SUM(bat_seasons.SH);
    totBB = SUM(bat_seasons.BB);  totH   = SUM(bat_seasons.H);
    toth1B = SUM(bat_seasons.h1B); toth2B = SUM(bat_seasons.h2B);
    toth3B = SUM(bat_seasons.h3B); tothR  = SUM(bat_seasons.HR);
    totR  = SUM(bat_seasons.R);   totRBI = SUM(bat_seasons.RBI);
    OBP   = 1.0*(totH + totBB + totHBP) / totPA;
    SLG   = 1.0*(toth1B + 2*totH2B + 3*totH3B + 4*totHR) / totAB;
    team_ids = DISTINCT bat_seasons.team_id;
    GENERATE
        group                  AS player_id,
        COUNT_STAR(bat_seasons) AS n_seasons,
        COUNT_STAR(team_ids)   AS card_teams,
        MIN(bat_seasons.year_id) AS beg_year,
        MAX(bat_seasons.year_id) AS end_year,
        totG AS G,
        totPA AS PA,  totAB AS AB,  totHBP AS HBP,      -- $6 - $8
        totSH AS SH,  totBB AS BB,  totH   AS H,        -- $9 - $11
        toth1B AS h1B, toth2B AS h2B, toth3B AS h3B,    -- $12 - $14
        tothR AS HR,  totR   AS R,   totRBI AS RBI,    -- $15 - $17
        OBP AS OBP,  SLG AS SLG,  (OBP + SLG) AS OPS    -- $18 - $20
    ;
};
```

We first gather together all seasons by a player by grouping on `player_id`, then throw a barrage of `SUM`, `COUNT_STAR`, `MIN` and `MAX` functions at the accumulated fields to find the career totals. Using the nested `FOREACH` form means we can use intermediate values such as `totPA` in both the calculation of `OBP` and as a field in the new table directly.

The nested FOREACH also lets us apply the DISTINCT bag operation, creating a new bag holding only the distinct team_id values across all seasons. That statement has, in principle, two steps: projection of a bag-with-just-team_id followed by DISTINCT to eliminate duplicates. But behind the scenes, Pig uses a special kind of bag (`DistinctData Bag`) that in all respects meets the data bag interface, but which uses an efficient internal data structure to eliminate duplicates as they're added. So rather than (list of seasons) → (list of team_ids) → (list of distinct team_ids) you only have to pay for (list of seasons) → (list of distinct team_ids)

We will use the `batcareers` table in several later demonstrations, so keep its output file around.

Pattern in Use

See the Pattern in Use for the next section too (REF).

- *Where You'll Use It* — Everywhere. Turning manufactured items into statistics about batches. Summarizing a cohort. Rolling up census block statistics to state-level statistics.
- *Standard Snippet* — FOREACH (GROUP recs BY key) GENERATE group AS mykey, AggregateFunction(recs), AggregateFunction(recs), ...;
- *Hello, SQL Users* — Directly comparable for the most part.
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values. Big decrease in output size from turning bags into scalars
- *Records* — Something like `mykey, aggregated_value, aggregated_value, ...`
- *Data Flow* — Map, Combiner & Reduce; combiners quite effective unless cardinality is very high.

Completely Summarizing a Field

In the preceding case, the aggregate functions were used to create an output table with similar structure to the input table, but at a coarser-grained relational level: career rather than season. The result was a new table to analyze, not a conceptual report.

Statistical aggregations also let you summarize groups and tables with well-understood descriptive statistics. By sketching their essential characteristics at dramatically smaller size, we make the data easier to work with but more importantly we make it possible to comprehend.

The following functions are built in to Pig:

- Count of all values: `COUNT_STAR(bag)`

- Count of non-null values: COUNT(bag)
- Minimum / Maximum non-null value: MIN(bag) / MAX(bag)
- Sum of non-null values: SUM(bag)
- Average of non-null values: AVG(bag)

There are a few additional summary functions that aren't native features of Pig, but are offered by LinkedIn's might-as-well-be-native DataFu package.⁴.

- Cardinality (i.e. the count of distinct values): combine the DISTINCT operation and the COUNT_STAR function as demonstrated below, or use the DataFu HyperLogLog PlusPlus UDF
- Variance of non-null values: VAR(bag), using the datafu.pig.stats.VAR UDF
- Standard Deviation of non-null values: SQRT(VAR(bag))
- Quantiles: Quantile(bag) or StreamingQuantile(bag)
- Median (50th Percentile Value) of a Bag: Median(bag) or StreamingMedian(bag)

The previous chapter (REF) has details on how to use UDFs, and so we're going to leave the details of that to the sample code. You'll also notice we list two functions for quantile and for median. Finding the exact median or other quantiles (as the Median/Quantile UDFs do) is costly at large scale, and so a good approximate algorithm (StreamingMedian/StreamingQuantile) is well appreciated. Since the point of this stanza is to characterize the values for our own sense-making, the approximate algorithms are appropriate. We'll have much more to say about why finding quantiles is costly, why finding averages isn't, and what to do about it in the Statistics chapter (REF).

```
weight_yr_stats = FOREACH (GROUP bat_seasons BY year_id) {
    dist      = DISTINCT bat_seasons.weight;
    sorted_a  = FILTER  bat_seasons.weight BY weight IS NOT NULL;
    sorted    = ORDER   sorted_a BY weight;
    some     = LIMIT   dist.weight 5;
    n_recs   = COUNT_STAR(bat_seasons);
    n_notnulls = COUNT(bat_seasons.weight);
    GENERATE
        group,
        AVG(bat_seasons.weight)      AS avg_val,
```

4. If you've forgotten/never quite learned what those functions mean, hang on for just a bit and we'll demonstrate them in context. If that still doesn't do it, set a copy of [Naked Statistics](#) or [Head First Statistics](#) next to this book. Both do a good job of efficiently imparting what these functions mean and how to use them without assuming prior expertise or interest in mathematics. This is important material though. Every painter of landscapes must know how to convey the essence of a [happy little tree](#) using a few deft strokes and not the prickly minutiae of its 500 branches; the above functions are your brushes footnote:[Artist/Educator Bob Ross: "Anyone can paint, all you need is a dream in your heart and a little bit of practice" — hopefully you're feeling the same way about Big Data analysis.

```

SQRT(VAR(bat_seasons.weight)) AS stddev_val,
MIN(bat_seasons.weight) AS min_val,
FLATTEN(ApproxEdgeile(sorted)) AS (p01, p05, p50, p95, p99),
MAX(bat_seasons.weight) AS max_val,
--
n_recs AS n_recs,
n_recs - n_notnulls AS n_nulls,
COUNT_STAR(dist) AS cardinality,
SUM(bat_seasons.weight) AS sum_val,
BagToString(some, '^') AS some_vals
;
};

```

Pattern in Use

- *Where You'll Use It* — Everywhere. Quality statistics on manufacturing batches. Response times of webserver requests. A/B testing in eCommerce.
- *Standard Snippet* — FOREACH (GROUP recs BY key) { ... ; GENERATE ...; };
- *Hello, SQL Users* — Directly comparable for the most part.
- *Important to Know*
 - Say COUNT_STAR(recs), not COUNT_STAR(recs.myfield) — the latter creates a new bag and interferes with combiner'ing.
 - Use COUNT_STAR and never SIZE on a bag.
 - Say SUM(recs.myfield), not SUM(myfield) (which isn't in scope).
 - Get in the habit of writing COUNT_STAR and never COUNT, unless you explicitly mean to only count non-'null's.
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values. Big decrease in output size from turning bags into scalars
- *Records* — Something like mykey, aggregated_value, aggregated_value, ...
- *Data Flow* — Map, Combiner & Reduce; combiners quite effective unless cardinality is very high.

Summarizing Aggregate Statistics of a Full Table

To summarize the statistics of a full table, we use a GROUP ALL statement. That is, instead of GROUP [table] BY [key], write GROUP [table] ALL. Everything else is as usual:

Summary of Weight Field.

```

weight_summary = FOREACH (GROUP bat_seasons ALL) {
    dist      = DISTINCT bat_seasons.weight;
    sorted_a  = FILTER  bat_seasons.weight BY weight IS NOT NULL;
    sorted    = ORDER   sorted_a BY weight;
}

```

```

some      = LIMIT    dist.weight 5;
n_recs   = COUNT_STAR(bat_seasons);
n_notnulls = COUNT(bat_seasons.weight);
GENERATE
  group,
  AVG(bat_seasons.weight)           AS avg_val,
  SQRT(VAR(bat_seasons.weight))     AS stddev_val,
  MIN(bat_seasons.weight)           AS min_val,
  FLATTEN(ApproxEdgeile(sorted))   AS (p01, p05, p50, p95, p99),
  MAX(bat_seasons.weight)           AS max_val,
  n_recs                          AS n_recs,
  n_recs - n_notnulls              AS n_nulls,
  COUNT_STAR(dist)                 AS cardinality,
  SUM(bat_seasons.weight)           AS sum_val,
  BagToString(some, '^')           AS some_vals
;
};


```

As we hope you readily recognize, using the GROUP ALL operation can be dangerous, as it requires bringing all the data onto a single reducer.

We're safe here, even on larger datasets, because all but one of the functions we supplied above are efficiently *algebraic*: they can be significantly performed in the map phase and combiner'ed. This eliminates most of the data before the reducer. The cardinality calculation, done here with a nested DISTINCT operation, is the only real contributor to reducer-side data size. For this dataset its size is manageable, and if it weren't there is a good approximate cardinality function. We'll explain the why and the how of algebraic functions and these approximate methods in the Statistics chapter. But you'll get a good feel for what is and isn't efficient through the examples in this chapter.)

Pattern in Use

Everything we said for “Completely Summarizing a Group” (REF), plus

- *Where You'll Use It* — Getting to know your data. Computing relative statistics or normalizing values. Topline totals and summaries.
- *Standard Snippet* — See the `summarizer_bot_9000.pig` macro
- *Hello, SQL Users* — Aggregate functions *without* a GROUP BY
- *Important to Know*
 - You're sending all the data to one reducer, so make sure the aggregate functions are highly reductive
 - Note the syntax of the full-table group statement. There's no I in TEAM, and no BY in GROUP ALL.
- *Output Count* — Single row

- *Data Flow* — Map, Combiner, and **single** reducer

(TODO-qem: should “Note the syntax of the full-table group statement. There’s no I in TEAM, and no BY in GROUP ALL.” be an “Important to Know” or a “NOTE\:”?)

Summarizing a String Field

We showed how to examine the constituents of a string field in the preceding chapter, under “Tokenizing a String” (REF). But for forensic purposes similar to the prior example, it’s useful to summarize their length distribution.

Summary of a String Field.

```

name_first_summary_0 = FOREACH (GROUP bat_seasons ALL) {
    dist      = DISTINCT bat_seasons.name_first;
    lens      = FOREACH bat_seasons GENERATE SIZE(name_first) AS len;
    --
    n_recs    = COUNT_STAR(bat_seasons);
    n_notnulls = COUNT(bat_seasons.name_first);
    --
    examples  = LIMIT dist.name_first 5;
    snippets   = FOREACH examples GENERATE (SIZE(name_first) > 15 ? CONCAT(SUBSTRING(name_first, 0,
    GENERATE
        group,
        'name_first'                  AS var:chararray,
        MIN(lens.len)                AS minlen,
        MAX(lens.len)                AS maxlen,
        --
        AVG(lens.len)                AS avglen,
        SQRT(VAR(lens.len))          AS stdvlen,
        SUM(lens.len)                AS sumlen,
        --
        n_recs                      AS n_recs,
        n_recs - n_notnulls          AS n_nulls,
        COUNT_STAR(dist)             AS cardinality,
        MIN(bat_seasons.name_first)  AS minval,
        MAX(bat_seasons.name_first)  AS maxval,
        BagToString(snippets, '^')    AS examples,
        lens  AS lens
    ;
};

name_first_summary = FOREACH name_first_summary_0 {
    sortlens  = ORDER lens BY len;
    pctiles   = ApproxEdgeile(sortlens);
    GENERATE
        var,
        minlen, FLATTEN(pctiles) AS (p01, p05, p10, p50, p90, p95, p99), maxlen,
        avglen, stdvlen, sumlen,
        n_recs, n_nulls, cardinality,
        minval, maxval, examples
}

```

```
    ;  
};
```

Pattern in Use

Everything we said for “Completely Summarizing a Group” (REF), plus

- *Where You’ll Use It* — Getting to know your data. Sizing string lengths for creating a database schema. Making sure there’s nothing ill-formed or outrageously huge. Making sure all values for a categorical field or string key is correct.
- *Standard Snippet* — See the `summarizer_bot_9000.pig` macro
- *Hello, SQL Users* — Corresponding functions *without* a `GROUP BY`
- *Important to Know*
 - You’re sending all the data to one reducer, so make sure the aggregate functions are highly reductive
 - Note the syntax of the full-table group statement. There’s no `I` in `TEAM`, and no `BY` in `GROUP ALL`.
- *Output Count* — Single row
- *Data Flow* — Map, Combiner, and `single` reducer

Calculating the Distribution of Numeric Values with a Histogram

One of the most common uses of a group-and-aggregate is to create a histogram showing how often each value (or range of values) of a field occur. This calculates the distribution of seasons played — that is, it counts the number of players whose career lasted only a single season; who played for two seasons; and so forth.

Histogram of Number of Seasons.

```
vals = FOREACH batcareers GENERATE n_seasons AS bin;  
seasons_hist = FOREACH (GROUP vals BY bin) GENERATE  
    group AS bin, COUNT_STAR(vals) AS ct;  
  
vals = FOREACH (GROUP bat_seasons BY (player_id, name_first, name_last)) GENERATE  
    COUNT_STAR(bat_seasons) AS bin, flatten(group);  
seasons_hist = FOREACH (GROUP vals BY bin) {  
    some_vals = LIMIT vals 3;  
    GENERATE group AS bin, COUNT_STAR(vals) AS ct, BagToString(some_vals, '|');  
};
```

So the pattern here is to:

- Project only the values,

- Group by the values,
- Produce the group as key and the count as value.

Pattern in Use

- *Where You'll Use It* — Anywhere you need a more detailed sketch of your data than average/standard deviation or simple quantiles can provide
- *Standard Snippet* — `vals = FOREACH recs GENERATE myfield AS bin; hist = FOREACH (GROUP vals BY bin) GENERATE group AS bin, COUNT_STAR(vals) AS ct;`. Or see `summarizer_bot_9000.pig` for a macro (REF).
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values
- *Records* — Output is `bin, ct:long`. You've turned records-with-values into values-with-counts
- *Data Flow* — Map, Combiner & Reduce; combiners very effective unless cardinality extremely high

Binning Data for a Histogram

Generating a histogram for games just as above produces mostly-useless output. There's no material difference between a career of 2000 games and one of 2001 games, but each value receives its own count — making it hard to distinguish the density of 1-, 2-, and 3-count bins near 1000 games from the 1-, 2-, and 3-count bins near 1500 games.

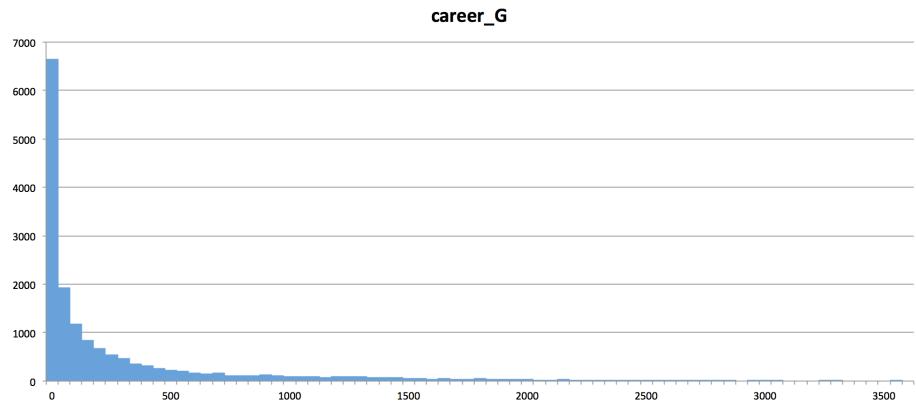
```
-- Meaningless
G_vals = FOREACH bat_seasons GENERATE G AS val;
G_hist = FOREACH (GROUP G_vals BY val) GENERATE
    group AS val, COUNT_STAR(G_vals) AS ct;
```

Instead, we will bin the data: divide by the bin size (50 in this case), and then multiply back by the bin size. The result of the division is an integer (since both the value and the bin size are of type `int`), and so the resulting value of `bin` is always an even multiple of the bin size. Values of 0, 12 and 49 all go to the 0 bin; 150 games goes to the 150 bin; and Pete Rose's total of 3,562 games played becomes the only occupant of bin 3550.

```
-- Binning makes it sensible
G_vals = FOREACH bat_seasons GENERATE 50*FLOOR(G/50) AS val;
G_hist = FOREACH (GROUP G_vals BY val) GENERATE
    group AS val, COUNT_STAR(G_vals) AS ct;
```

Histogram of Career Games Played

The histogram on the binned data is now quite clear:



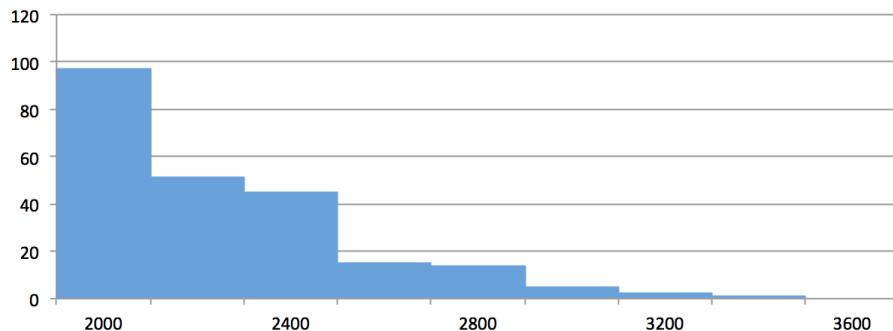
Choosing a Bin Size

How do you choose a binsize? The following three graphs zoom in on the tail (2000 or more games) to show bin sizes that are too large, too small, and just right.

Binsize too large

A bin size of 200 is too coarse, washing out legitimate gaps that tell a story.

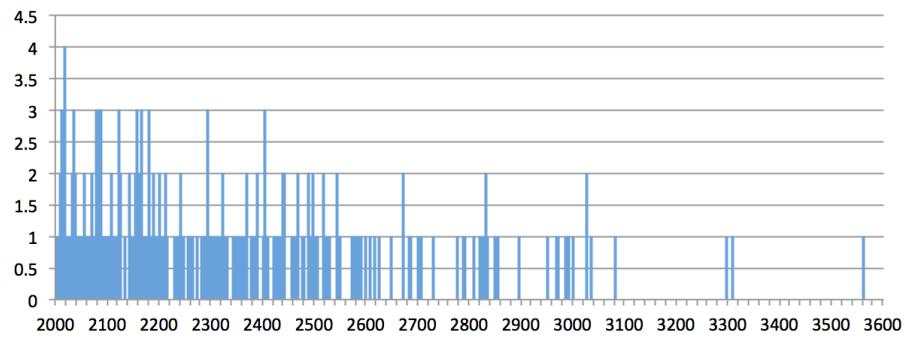
G binsize 200



Binsize too small

The bin size of 2 is too fine — the counts are small, there are many trivial gaps, and there is a lot of non-meaningful bin-to-bin variation.

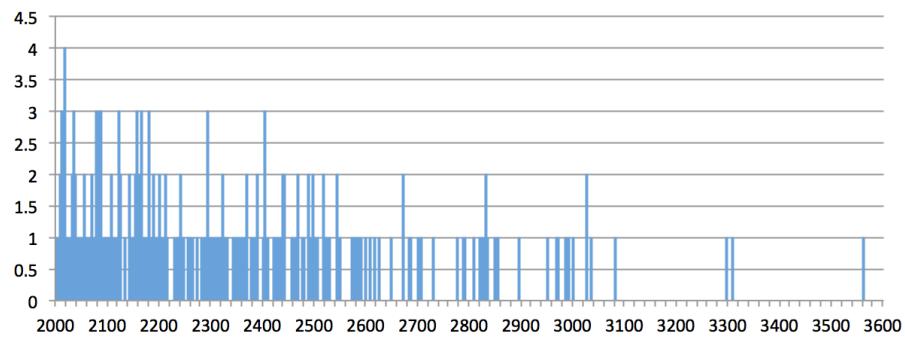
G binsize 2



Binsize too small

The bin size we chose, 50 games, works well. It's a meaningful number (50 games represents about 1/3 of a season), it gives meaty counts per bin even when the population starts to become sparse, and yet preserves the gaps that demonstrate the epic scope of Pete Rose and our other outliers' careers.

G binsize 2



Interpreting Histograms and Quantiles

Different underlying mechanics will give different distributions.

Games Played — linear

The histogram of career games shows that most players see only one game their whole career, and the counts drop off continuously at higher and higher career totals. You can't play 30 games unless you were good enough to make it in to 29 games; you can't play

100 games unless you continued to be good, didn't get injured, didn't get old, didn't go to war between the thirtieth and ninety-ninth game, and so on.

images::images/06-histograms-career_G-linear.png[Histogram of Career Games, Linear Axis]

Games Played — Log-Log plot

Distributions, such as this one, that span many orders of magnitude in value and count, are easier to understand using a *log-log graph*. The “log” is short for “logarithm,” in which successive values represent orders of magnitude difference. On a log-log graph, then, the axes arrange the displayed values so that the same distance separates 1 from 10 as separates 10 from 100 and so on, for any *ratio* of values.

Though the career games data shows a very sharp dropoff, it is *not* a long-tail distribution, as you can see by comparing a power-law fit (which is always a straight line on a log-log graph) to the actual histogram.

images::images/06-histograms-career_G-loglog.png[Histogram of Career Games, Log-Log plot]

Binning Data into Exponentially Sized Buckets

In contrast, webpage views known to be are one of many phenomena that obey the “long-tail” distribution, as we can see by generating a histogram of hourly pageview counts for each Wikipedia page⁵. Since the data is so sharply exponential, we are better off binning it *logarithmically*. To do so we take the log of the value, chunk it (using the multiply-floor-undo method again), and then take the exponential to restore a representative value for the bin. (You'll notice we avoid trouble taking the logarithm of zero by feeding it an insignificantly small number instead. This lets zero be included in the processing without materially altering the result)

```
pagecount_views = LOAD '/data/out/wikipedia/pagecount-views.tsv' AS (val:long);
SET eps 0.001
;

view_vals = FOREACH pagecount_views GENERATE
    (long)EXP( FLOOR(LOG((val == 0 ? $eps : val)) * 10)/10.0 ) AS bin;
hist_wp_view = FOREACH (GROUP view_vals BY bin) GENERATE
    group AS bin, COUNT_STAR(view_vals) AS ct;
```

images::06-histograms-pageviews-loglog.png[Histogram of Wikipedia Hourly Page-views, Log-Log plot]

5. For 11pm UTC on Oct 2nd, 2008, because that was what was nearby

The result indeed is a nice sharp line on the log-log plot, and the logarithmic bins did a nice job of accumulating robust counts while preserving detail. Logarithmic bins are generally a better choice any time you're using a logarithmic x-axis because it means that the span of each bin is visually the same size, aiding interpretation.

As you can see, you don't have to only bin linearly. Apply any function that takes piecewise segments of the domain and maps them sequentially to the integers, then undo that function to map those integers back to a central value of each segment. The Wikipedia webserver logs data also includes the total *bytes* transferred per page; this data spans such a large range that we end up binning both logarithmically (to tame the upper range of values) and linearly (to tame the lower range of values) — see the sample code for details.

Pattern in Use

See Pattern in Use for Histograms, above (REF)

- *Where You'll Use It* — Anywhere the values make sense exponentially; eg values make sense as 1, 100, 1000, ..., 10 million rather than 1 million, 2 million, ..., 10 million. Anywhere you will use a logarithmic X axis for displaying the bin values.
- *Important to Know* — The result is a representative value from the bin (eg 100), and not the log of that value (eg `log(100)`). Decide whether representative should be a central value from the bin or the minimum value in the bin.
- *Standard Snippet* — `(long)EXP(FLOOR(LOG((val == 0 ? $eps : val)) * bin_sf)/bin_sf)` for scale factor `bin_sf`. Instead of substituting `$eps` for zero you might prefer to filter them out.

Creating Pig Macros for Common Stanzas

Rather than continuing to write the histogram recipe over and over, let's take a moment and generalize. Pig allows you to create macros that parameterize multiple statements:

```
DEFINE histogram(table, key) RETURNS dist {
    vals = FOREACH $table GENERATE $key;
    $dist = FOREACH (GROUP vals BY $key) GENERATE
        group AS val, COUNT_STAR(vals) AS ct;
};

DEFINE binned_histogram(table, key, binsize, maxval) RETURNS dist {
    numbers = load_numbers_10k();
    vals = FOREACH $table GENERATE (long)(FLOOR($key / $binsize) * $binsize) AS bin;
    all_bins = FOREACH numbers GENERATE (num0 * $binsize) AS bin;
    all_bins = FILTER all_bins BY (bin <= $maxval);
    $dist = FOREACH (COGROUP vals BY bin, all_bins BY bin) GENERATE
        group AS bin, (COUNT_STAR(vals) == 0L ? null : COUNT_STAR(vals)) AS ct;
};
```

Distribution of Games Played

Call the histogram macro as follows:

```
career_G_hist      = binned_histogram(batcareers, 'G', 50, 3600);
career_G_hist_2   = binned_histogram(batcareers, 'G', 2, 3600);
career_G_hist_200 = binned_histogram(batcareers, 'G', 200, 3600);

height_hist       = binned_histogram(people, 'height_in', 40, 80);
weight_hist       = binned_histogram(people, 'weight_lb', 10, 300);

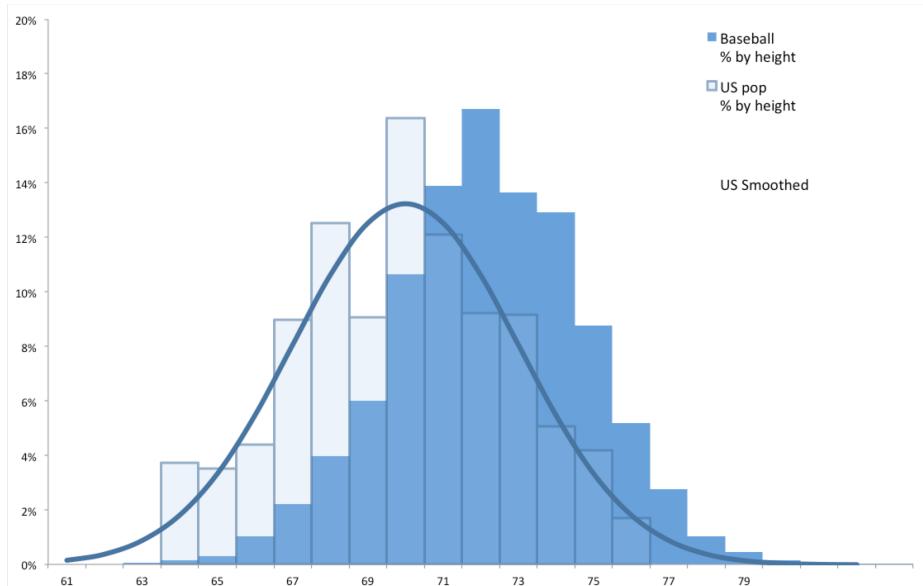
birthmo_hist      = histogram(people, 'birth_month');
deathmo_hist      = histogram(people, 'death_month');
```

Now that finding a histogram is effortless, let's examine more shapes of distributions.

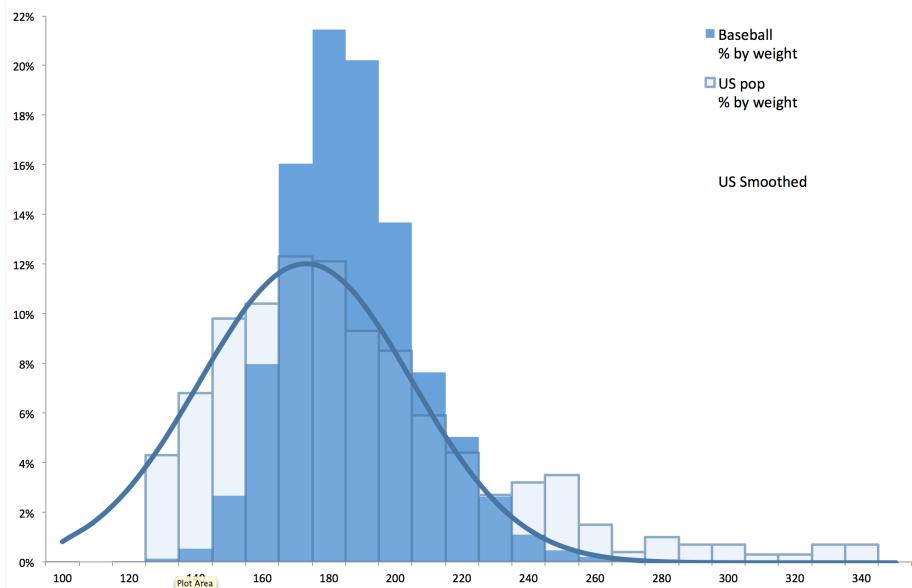
Extreme Populations and Confounding Factors

To reach the major leagues, a player must possess multiple extreme attributes: ones that are easy to measure, like being tall or being born in a country where baseball is popular; and ones that are not, like field vision, clutch performance, the drive to put in outlandishly many hours practicing skills. Any time you are working with extremes as we are, you must be very careful to assume their characteristics resemble the overall population's.

Height



Weight



Here again are the graphs for players' height and weight, but now graphed against (in light blue) the distribution of height/weight for US males aged 20-29⁶.

The overall-population distribution is shown with light blue bars, overlaid with a normal distribution curve for illustrative purposes. The population of baseball players deviates predictably from the overall population: it's an advantage to The distribution of player weights, meanwhile, is shifted somewhat but with a dramatically smaller spread.

Distribution of Birth and Death day of year

Surely at least baseball players are born and die like the rest of us, though?

With a little Pig action, we can generate some histograms to answer that question:

Vital Stats pt 1.

```
vitals = FOREACH peeps GENERATE
    height_in,
    10*CEIL(weight_lb/10.0) AS weight_lb,
    birth_month,
    death_month;

    birth_month_hist = histogram(vitals, 'birth_month');
```

6. US Census Department, Statistical Abstract of the United States. Tables 206 and 209, Cumulative Percent Distribution of Population by (Weight/Height) and Sex, 2007-2008; uses data from the U.S. National Center for Health Statistics

```

death_month_hist = histogram(vitals, 'death_month');
height_hist = histogram(vitals, 'height_in');
weight_hist = histogram(vitals, 'weight_lb');

Vital Stats pt 2.

peep_stats = FOREACH (GROUP attr_vals_nn ALL) GENERATE
    BagToMap(CountVals(attr_vals_nn.attr)) AS cts:map[long];
peep_hist = FOREACH (GROUP attr_vals BY (attr, val)) {
    ct = COUNT_STAR(attr_vals);
    GENERATE
        FLATTEN(group) AS (attr, val),
        ct          AS ct
        -- , (float)ct / ((float)peep_stats.ct) AS freq
    ;
};

peep_hist = ORDER peep_hist BY attr, val;
one = LOAD '$data_dir/stats/numbers/one.tsv' AS (num:int);
ht = FOREACH one GENERATE peep_stats.cts#'height';

```

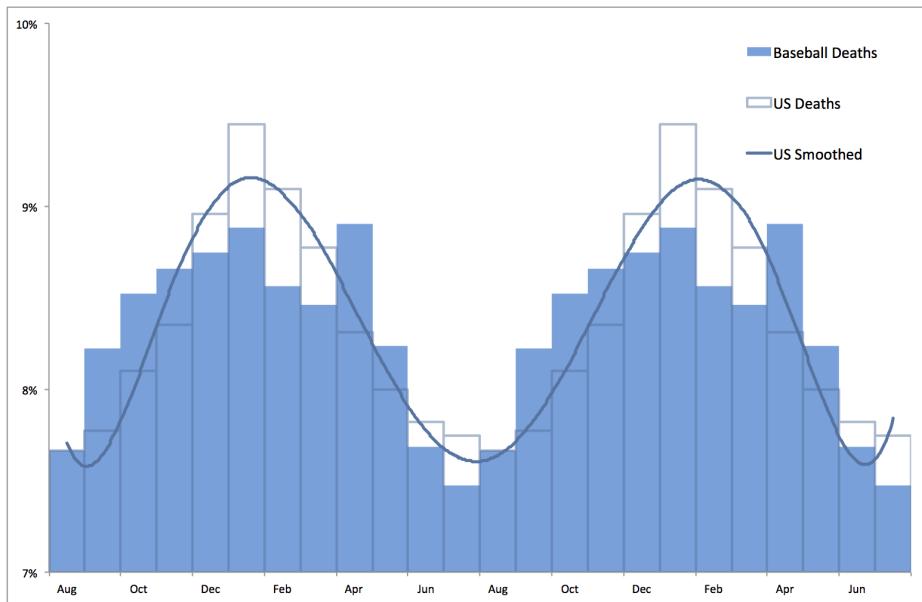
Deaths

These graphs show the relative seasonal distribution of death rates, with adjustment for the fact that there are fewer days in February than July and so forth. As above, the background US rates are shown as darker outlined bars and the results from our data set as solid blue bars.

We were surprised to see how seasonal the death rate is. We all probably have a feel there's more birthday party invitations in September than in March, but hopefully not so much for funerals. This pattern is quite consistent and as you might guess inverted in the Southern Hemisphere. Most surprisingly of all, it **persists even in places with a mild climate**. The most likely cause of fewer deaths in the summer is *not* fewer snow-covered driveways to shovel, it is that people take vacations — lowering stress, improving mood, and synthesizing vitamin D. (And there's no clear signal of “hanging on for Christmas” in the data).

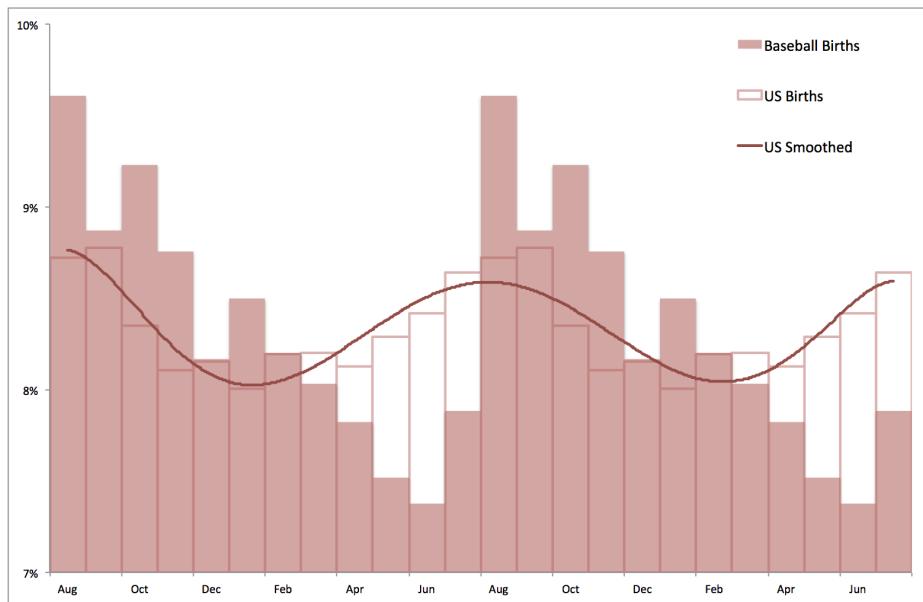
The baseball distribution is lumpier, as you'd expect from its smaller sample size⁷, but matches the background distribution. Death treats baseball players, at least in this regard, as it does us all.

7. We don't think the April spike is anything significant (“Hanging on for one more Opening Day celebration?”); sometimes lumpy data is lumpy



Births

That is not true for the birth data! The format of the graph is the same as above, and again we see a seasonal distribution — with a peak nine months after the cold winter temperatures induce people to stay home and find alternative recreations. But the baseball data does *not* match the background distribution at all. The sharp spike in August following the nadir in May and June appears nowhere in the background data, and its phase (where it crosses the centerline) is shifted later by several months. In this data set, a player born in August is about 25% more likely to make the major leagues than a player born in June; restricting it to players from the United States born after 1950 makes august babies 50% more likely to earn a baseball card than June babies.



The reason is that since the 1940s, American youth leagues have used July 31st as an age cutoff. If Augusta were born on August 1st, then four calendar years and 364 days later she would still technically be four years old. Julien, who showed up the day before her and thus has spent five years and no days orbiting the Sun, is permitted to join the league as a five-year-old. The Augustas may be initially disappointed, but when they do finally join the league as five-year-and-364-day-old kids, they have nearly an extra year of growth compared to the Juliens who sign up with them, which on the whole provides a huge advantage at young ages. This earns the Augustas extra attention from their coaches, extra validation of their skill, and extra investment of “I’m good at Baseball!” in their identity.

Don’t Trust Distributions at the Tails

A lot of big data analyses explore population extremes: manufacturing defects, security threats, disease carriers, peak performers. Elements arrive into these extremes exactly because multiple causative features drive them there (such as an advantageous height or birth month); and a host of other conflated features follow from those deviations (such as those stemming from the level of fitness athletes maintain).

So whenever you are examining populations of outliers, you cannot depend on their behavior resembling the universal population. Normal distributions may not remain normal and may not even retain a central tendency; independent features in the general population may become tightly coupled in the outlier group; and a host of other easy assumptions become invalid. Stay alert.

Calculating a Relative Distribution Histogram

The histograms we've calculated have results in terms of counts. The results do a better general job of enforcing comparisons if express them as relative frequencies: as fractions of the total count. You know how to find the total:

```
HR_stats = FOREACH (GROUP bats BY ALL) GENERATE COUNT_STAR(bats) AS n_players;
```

The problem is that HR_stats is a single-row table, and so not something we can use directly in a FOREACH expression. Pig gives you a piece of syntactic sugar for this specific case of a one-row table⁸: project the value as tablename.field as if it were an inner bag, but slap the field's type (in parentheses) in front of it like a typecast expression:

```
HR_stats = FOREACH (GROUP bats BY ALL) GENERATE COUNT_STAR(bats) AS ct;
HR_hist = FOREACH (GROUP bats BY HR) {
    ct = COUNT_STAR(bats);
    GENERATE HR as val,
        ct/((long)HR_stats.ct) AS freq,
        ct;
}
```

Typecasting the projected field as if you were simply converting the schema of a field from one scalar type to another acts as a promise to Pig that what looks like column of possibly many values will turn out to have only row. In return, Pig will understand that you want a sort of über-typecast of the projected column into what is effectively its literal value.

Pattern in Use

See Pattern in Use for “Histograms”, above (REF), and “Re-injecting Global Values”, following (REF).

- *Where You'll Use It* — Histograms on sampled populations. Whenever you want frequencies rather than counts, i.e. proportions rather than absolute values.
- *Standard Snippet* — Same as for a histogram, but with COUNT_STAR(vals)/((long)recs_info.ct) AS freq. See `summarizer_bot_9000.pig` for a macro (REF).

Re-injecting Global Values

Sometimes things are more complicated, and what you'd like to do is perform light synthesis of the results of some initial Hadoop jobs, then bring them back into your script as if they were some sort of “global variable”. But a pig script just orchestrates the top-level motion of data: there's no good intrinsic ways to bring the result of a step into

8. called *scalar projection* in Pig terminology

the declaration of following steps. You can use a backhoe to tear open the trunk of your car, but it's not really set up to push the trunk latch button. The proper recourse is to split the script into two parts, and run it within a workflow tool like Rake, Drake or Oozie. The workflow layer can fish those values out of the HDFS and inject them as runtime parameters into the next stage of the script.

In the case of global counts, it would be so much faster if we could sum the group counts to get the global totals; but that would mean a job to get the counts, a job to get the totals, and a job to get the relative frequencies. Ugh.

If the global statistic is relatively static, there are occasions where we prefer to cheat. Write the portion of the script that finds the global count and stores it, then comment that part out and inject the values statically — the sample code shows you how to do it with a templating runner, as runtime parameters, by copy/pasting, or using the `cat` Grunt shell statement. Then, to ensure your time-traveling shenanigans remain valid, add an `ASSERT` statement comparing the memoized values to the actual totals. Pig will not only run the little checkup stage in parallel if possible, it will realize that the data size is small enough to run as a local mode job — cutting the turnaround time of a tiny job like that in half or better.

```
-- cheat mode:  
-- HR_stats = FOREACH (GROUP bats BY ALL) GENERATE COUNT_STAR(bats) AS n_total;  
SET HR_stats_n_total = `cat $out_dir/HR_stats_n_total`;  
  
HR_hist = FOREACH (GROUP bats BY HR) {  
    ct = COUNT_STAR(bats);  
    GENERATE HR as val, ct AS ct,  
    -- ct/( (long)HR_stats.n_total ) AS freq,  
    ct/( (long)HR_stats_n_total ) AS freq,  
    ct;  
};  
-- the much-much-smaller histogram is used to find the total after the fact  
--  
ASSERT (GROUP HR_hist ALL)  
    IsEqualish( SUM(freq), 1.0 ),  
    (HR_stats_n_total == SUM(ct));
```

As we said, this is a cheat-to-win scenario: using it to knock three minutes off an eight minute job is canny when used to make better use of a human data scientist's time, foolish when applied as a production performance optimization.

Calculating a Histogram Within a Group

As long as the groups in question do not rival the available memory, counting how often each value occurs within a group is easily done using the DataFu `CountEach` UDF. There's been a trend over baseball's history for increased specialization

<http://datafu.incubator.apache.org/docs/datafu/guide/bag-operations.html>

You'll see the

```
DEFINE CountVals          datafu.pig.bags.CountEach('flatten');
binned = FOREACH sig_seasons GENERATE
  ( 5 * ROUND(year_id/ 5.0f)) AS year_bin,
  (20 * ROUND(H      /20.0f)) AS H_bin;

hist_by_year_bags = FOREACH (GROUP binned BY year_bin) {
  H_hist_cts = CountVals(binned.H_bin);
  GENERATE group AS year_bin, H_hist_cts AS H_hist_cts;
};
```

We want to normalize this to be a relative-fraction histogram, so that we can make comparisons across eras even as the number of active players grows. Finding the total count to divide by is a straightforward COUNT_STAR on the group, but a peccadillo of Pig's syntax makes using it a bit frustrating. Annoyingly, a nested FOREACH can only "see" values from the bag it's operating on, so there's no natural way to reference the calculated total from the FOREACH statement.

```
-- Won't work:
hist_by_year_bags = FOREACH (GROUP binned BY year_bin) {
  H_hist_cts = CountVals(binned.H_bin);
  tot        = 1.0f*COUNT_STAR(binned);
  H_hist_rel = FOREACH H_hist_cts GENERATE H_bin, (float)count/tot;
  GENERATE group AS year_bin, H_hist_cts AS H_hist_cts, tot AS tot;
};
```

The best current workaround is to generate the whole-group total in the form of a bag having just that one value. Then we use the CROSS operator to graft it onto each (bin,count) tuple, giving us a bag with (bin,count,total) tuples — yes, every tuple in the bag will have the same group-wide value. Finally, iterate across the tuples to find the relative frequency.

It's more verbose than we'd like, but the performance hit is limited to the CPU and GC overhead of creating three bags `\{(result,count)\}`, `\{(result,count,total)\}`, `\{(result,count,freq)\}` in quick order.

Histogram within a Group.

```
hist_by_year_bags = FOREACH (GROUP binned BY year_bin) {
  H_hist_cts = CountVals(binned.H_bin);
  tot        = COUNT_STAR(binned);
  GENERATE
    group      AS year_bin,
    H_hist_cts AS H_hist,
    {(tot)}   AS info:bag{(tot:long)}; -- single-tuple bag we can feed to CROSS
  };
hist_by_year = FOREACH hist_by_year_bags {
  -- Combines H_hist bag \{(100,93),(120,198)\...} and dummy tot bag \{(882.0)\}
  -- to make new (bin,count,total) bag: \{(100,93,882.0),(120,198,882.0)\...}
  H_hist_with_tot = CROSS H_hist, info;
```

```

-- Then turn the (bin,count,total) bag into the (bin,count,freq) bag we want
H_hist_rel      = FOREACH H_hist_with_tot
    GENERATE H_bin, count AS ct, count/((float)tot) AS freq;
    GENERATE year_bin, H_hist_rel;
};


```

Pattern in Use

- *Where You'll Use It* — Summarizing Cohorts. Comparatively plotting histograms as a small multiples plot (REF) or animation
- *Standard Snippet* — `DEFINE CountVals datafu.pig.bags.CountEach('flatten');` FOREACH (GROUP recs BY bin) GENERATE group, CountVals(recs.bin);. Must download and enable the DataFu package (REF)
- *Important to Know* — This is done largely in-memory at the reducer, so watch your data sizes
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values
- *Records* — Output is group, bag of (count, bin) tuples. You've turned bags of records-with-values into bags of values-with-counts
- *Data Flow* — Map & Reduce. As you'll learn in "Advanced Pig" (REF), `CountEach` is not an algebraic, but is an accumulator

Dumping Readable Results

We are of course terribly anxious to find out the results, so much so that having to switch over to R to graph our totals is more delay than we can bear. It's also often nice to have production jobs dump a visual summary of the results that an operator can easily scan and sanity-check. And so let's apply the "Formatting a String According to a Template" (REF) pattern to dump a readable summary of our results to the screen.

```

year_hists_HH = FOREACH year_hists {
    HH_hist_rel_o = ORDER HH_hist_rel BY bin ASC;
    HH_hist_rel_x = FILTER HH_hist_rel_o BY (bin >= 90);
    HH_hist_vis   = FOREACH HH_hist_rel_x GENERATE
        SPRINTF('%1$3d: %3$4.0f', bin, ct, ROUND_TO(100*freq, 1));
    GENERATE year_bin, BagToString(HH_hist_vis, ' ');
};


```

TODO-reviewers: previous version without comments, or following? In practice I would write it without comments.

```

year_hists_HH = FOREACH year_hists {
    -- put all bins in regular order
    HH_hist_rel_o = ORDER HH_hist_rel BY bin ASC;
    -- The PA threshold makes the lower bins ragged, exclude them
};


```

```

HH_hist_rel_x = FILTER HH_hist_rel_o BY (bin >= 90);
-- Format each bin/freq into a readable string
HH_hist_vis = FOREACH HH_hist_rel_x GENERATE
  SPRINTF('%1$3d: %3$4.0f', bin, ct, ROUND_TO(100*freq, 1));
-- Combine those strings into readable table
GENERATE year_bin, BagToString(HH_hist_vis, ' ');
};

```

In this snippet, we first put all bins in regular order and exclude the lower bins (the minimum-plate appearances threshold makes them ragged). Next, we transform each bin-count-frequency triple into a readable string using `SPRINTF`. Since we used positional specifiers (the `1$` part of `%1$3d`), it's easy to insert or remove fields in the display depending on what question you're asking. Here, we've omitted the count as it wasn't helpful for the main question we have: "What are the long-term trends in offensive production?". Finally, we use `BagToString` to format the row. We first met that combination of formatting-elements-formatting-bag in "Representing a Complex Data Structure with a Delimited String" (REF) above. (We hope you're starting to feel like Daniel-san in Karate Kid when all his work polishing cars comes together as deadly martial arts moves.)

Relative Distribution of Total Hits per Season by Five-Year Period, 1900-.

1900	100:	21	125:	38	150:	27	175:	9	200:	2	225:	1
1905	100:	30	125:	37	150:	20	175:	4	200:	2		
1910	100:	22	125:	40	150:	25	175:	9	200:	1	225:	1
1915	100:	25	125:	38	150:	20	175:	6	200:	1	225:	0
1920	100:	12	125:	26	150:	29	175:	21	200:	9	225:	1
1925	100:	13	125:	29	150:	26	175:	19	200:	9	225:	2
1930	100:	12	125:	30	150:	26	175:	20	200:	9	225:	1
1935	100:	13	125:	29	150:	29	175:	19	200:	8	225:	1
1940	100:	20	125:	35	150:	29	175:	11	200:	2		
1945	100:	26	125:	36	150:	22	175:	11	200:	2	225:	1
1950	100:	21	125:	29	150:	32	175:	12	200:	3		
1955	100:	27	125:	31	150:	22	175:	14	200:	2		
1960	100:	24	125:	29	150:	29	175:	12	200:	3	225:	0
1965	100:	26	125:	34	150:	24	175:	8	200:	2	225:	0
1970	100:	26	125:	35	150:	23	175:	9	200:	2	225:	0
1975	100:	23	125:	33	150:	26	175:	11	200:	3	225:	0
1980	100:	22	125:	34	150:	25	175:	11	200:	3	225:	0
1985	100:	27	125:	31	150:	26	175:	9	200:	3	225:	0
1990	100:	29	125:	33	150:	24	175:	10	200:	1		
1995	100:	20	125:	31	150:	29	175:	14	200:	3	225:	0
2000	100:	22	125:	30	150:	29	175:	13	200:	3	225:	0
2005	100:	19	125:	32	150:	28	175:	15	200:	3	225:	0
2010	100:	22	125:	36	150:	26	175:	11	200:	2		

Relative Distribution of Total Home Runs per Season by Five-Year Period, 1900-.

1900	0:	97	10:	3		
1905	0:	99	10:	1		
1910	0:	93	10:	6	20:	0

1915	0:	96	10:	3	20:	1													
1920	0:	77	10:	18	20:	3	30:	1	40:	1	50:	0							
1925	0:	71	10:	20	20:	4	30:	3	40:	1	50:	0	60:	0					
1930	0:	62	10:	25	20:	6	30:	5	40:	2	50:	0							
1935	0:	57	10:	27	20:	10	30:	4	40:	1	50:	0							
1940	0:	64	10:	24	20:	8	30:	3	40:	0									
1945	0:	58	10:	27	20:	10	30:	4	40:	1	50:	1							
1950	0:	39	10:	33	20:	18	30:	7	40:	3									
1955	0:	34	10:	32	20:	23	30:	8	40:	4	50:	1							
1960	0:	33	10:	34	20:	22	30:	8	40:	3	50:	0	60:	0					
1965	0:	38	10:	34	20:	19	30:	8	40:	2	50:	0							
1970	0:	39	10:	34	20:	20	30:	5	40:	2									
1975	0:	42	10:	33	20:	19	30:	6	40:	1	50:	0							
1980	0:	41	10:	34	20:	18	30:	6	40:	1									
1985	0:	33	10:	34	20:	25	30:	8	40:	1									
1990	0:	36	10:	35	20:	20	30:	7	40:	2	50:	0							
1995	0:	24	10:	32	20:	25	30:	13	40:	6	50:	1	60:	0	70:	0			
2000	0:	19	10:	35	20:	26	30:	14	40:	5	50:	1	60:	0	70:	0			
2005	0:	22	10:	34	20:	28	30:	12	40:	3	50:	1							
2010	0:	24	10:	37	20:	27	30:	11	40:	2	50:	0							

We'll need to draw graphs to get any nuanced insight, but the long-term trends in production of Hits and Home Runs is strong enough that this chart tells a clear story. Baseball has seen two offensive booms: one in the 1920-1939 period, and one in the 1990-2009 period. However, the first was an on-base boom, with a larger proportion of players crossing the 200-hit mark than ever have since. The recent one was decidedly a power-hitting boom. There is an increase in the fraction of players reaching high seasonal hit totals, but the chart above shouts how large the increase in the proportion of players hitting 30-, 40-, and 50-home runs per year is.

Pattern in Use

- *Where You'll Use It* — Production jobs, to give the operator a readable summary that the job not only ran to completion but gave meaningful results. In development, to Know Thy Data.
- *Standard Snippet* — A mashup of the Format with a Template, Represent Complex Data Structures, and Group-and-Aggregate patterns
- *Important to Know* — This is more valuable, and more used by experts, than you might think. You'll see.
- *Records* — Up to you; enough for your brain, not too much for your eyes.
- *Exercises for you:* Create a macro to generate such a table. It should accept parameters for sprintf template, filter limits and sort key. Consult the `summarizer_bot_9000.pig` file in the example code repository for other examples of macros.

The Summing Trick

There's a pattern-of-patterns we like to call the "Summing trick", a frequently useful way to act on subsets of a group without having to perform multiple GROUP BY or FILTER operations. Call it to mind every time you find yourself thinking "gosh, this sure seems like a lot of reduce steps on the same key". Before we describe its generic nature, it will help to see an example

Counting Conditional Subsets of a Group — The Summing Trick

Whenever you are exploring a dataset, you should determine figures of merit for each of the key statistics — easy-to-remember values that separate qualitatively distinct behaviors. You probably have a feel for the way that 30 C / 85 deg F reasonably divides a "warm" day from a "hot" one; and if I tell you that a sub-three-hour marathon distinguishes "really impress your friends" from "really impress other runners", you are equipped to recognize how ludicrously fast a 2:15 (the pace of a world-class runner) marathon is.

For our purposes, we can adopt 180 hits (H), 30 home runs (HR), 100 runs batted in (RBI), a 0.400 on-base percentage (OBP) and a 0.500 slugging percentage (SLG) each as the dividing line between a good and a great performance.

One reasonable way to define a great career is to ask how many great seasons a player had. We can answer that by counting how often a player's season totals exceeded each figure of merit. The obvious tactic would seem to involve filtering and counting each bag of seasonal stats for a player's career; that is cumbersome to write, brings most of the data down to the reducer, and exerts GC pressure materializing multiple bags.

```
-- Create indicator fields on each figure of merit for the season
standards = FOREACH mod_seasons {
    OBP      = 1.0*(H + BB + HBP) / PA;
    SLG      = 1.0*(h1B + 2*h2B + 3*h3B + 4*HR) / AB;
    GENERATE
        player_id,
        (H    >= 180 ? 1 : 0) AS hi_H,
        (HR   >= 30  ? 1 : 0) AS hi_HR,
        (RBI  >= 100 ? 1 : 0) AS hi_RBI,
        (OBP  >= 0.400 ? 1 : 0) AS hi_OBP,
        (SLG  >= 0.500 ? 1 : 0) AS hi_SLG
    ;
};
```

Next, count the seasons that pass the threshold by summing the indicator value

```
career_standards = FOREACH (GROUP standards BY player_id) GENERATE
    group AS player_id,
    COUNT_STAR(standards) AS n_seasons,
    SUM(standards.hi_H)   AS hi_H,
    SUM(standards.hi_HR)  AS hi_HR,
```

```
SUM(standards.hi_RBI) AS hi_RBI,  
SUM(standards.hi_OBP) AS hi_OBP,  
SUM(standards.hi_SLG) AS hi_SLG  
;
```

The summing trick involves projecting a new field whose value is based on whether it's in the desired set, forming the desired groups, and aggregating on those new fields. Irrelevant records are assigned a value that will be ignored by the aggregate function (typically zero or `null`), and so although we operate on the group as a whole, only the relevant records contribute.

In this case, instead of sending all the hit, home run, etc figures directly to the reducer to be bagged and filtered, we send a 1 for seasons above the threshold and 0 otherwise. After the group, we find the *count* of values meeting our condition by simply *summing* the values in the indicator field. This approach allows Pig to use combiners (and so less data to the reducer); and more importantly it doesn't cause a bag of values to be collected, only a running sum (and so way less garbage-collector pressure).

Another example will help you see what we mean — next, we'll use one GROUP operation to summarize multiple subsets of a table at the same time.

First, though, a side note on these figures of merit. As it stands, this isn't a terribly sophisticated analysis: the numbers were chosen to be easy-to-remember, and not based on the data. For actual conclusion-drawing, we should use the z-score (REF) or quantile (REF) figures (we'll describe both later on, and use them for our performance analysis instead). And yet, for the exploratory phase we prefer the ad-hoc figures. A 0.400 OBP is a number you can hold in your hand and your head; you can go click around [ESPN](#) and see that it selects about the top 10-15 players in most seasons; you can use paper-and-pencil to feed it to the run expectancy table (REF) we'll develop later and see what it says a 0.400-on-base hitter would produce. We've shown you how useful it is to identify exemplar records; learn to identify these touchstone values as well.

Summarizing Multiple Subsets of a Group Simultaneously

We can use the summing trick to apply even more sophisticated aggregations to conditional subsets. How did each player's career evolve — a brief brilliant flame? a rise to greatness? sustained quality? Let's classify a player's seasons by whether they are "young" (age 21 and below), "prime" (22-29 inclusive) or "older" (30 and older). We can then tell the story of their career by finding their OPS (our overall performance metric) both overall and for the subsets of seasons in each age range⁹.

9. these breakpoints are based on where www.fangraphs.com/blogs/how-do-star-hitters-age research by fangraphs.com showed a performance drop-off by 10% from peak.

The complication here over the previous exercise is that we are forming compound aggregates on the group. To apply the formula `career SLG = (career TB) / (career AB)`, we need to separately determine the career values for TB and AB and then form the combined SLG statistic.

Project the numerator and denominator of each offensive stat into the field for that age bucket. Only one of the subset fields will be filled in; as an example, an age-25 season will have values for PA_all and PA_prime and zeros for PA_young and PA_older.

```
age_seasons = FOREACH mod_seasons {
    young = (age <= 21 ? true : false);
    prime = (age >= 22 AND age <= 29 ? true : false);
    older = (age >= 30 ? true : false);
    OB = H + BB + HBP;
    TB = h1B + 2*h2B + 3*h3B + 4*HR;
    GENERATE
        player_id, year_id,
        PA AS PA_all, AB AS AB_all, OB AS OB_all, TB AS TB_all,
        (young ? 1 : 0) AS is_young,
        (young ? PA : 0) AS PA_young, (young ? AB : 0) AS AB_young,
        (young ? OB : 0) AS OB_young, (young ? TB : 0) AS TB_young,
        (prime ? 1 : 0) AS is_prime,
        (prime ? PA : 0) AS PA_prime, (prime ? AB : 0) AS AB_prime,
        (prime ? OB : 0) AS OB_prime, (prime ? TB : 0) AS TB_prime,
        (older ? 1 : 0) AS is_older,
        (older ? PA : 0) AS PA_older, (older ? AB : 0) AS AB_older,
        (older ? OB : 0) AS OB_older, (older ? TB : 0) AS TB_older
    ;
}
```

After the group, we can sum across all the records to find the plate-appearances-in-prime-seasons even though only some of the records belong to the prime-seasons subset. The irrelevant seasons show a zero value in the projected field and so don't contribute to the total.

Career Epochs.

```
career_epochs = FOREACH (GROUP age_seasons BY player_id) {
    PA_all      = SUM(age_seasons.PA_all );
    PA_young   = SUM(age_seasons.PA_young);
    PA_prime    = SUM(age_seasons.PA_prime);
    PA_older    = SUM(age_seasons.PA_older);
    -- OBP = (H + BB + HBP) / PA
    OBP_all    = 1.0f*SUM(age_seasons.OB_all) / PA_all ;
    OBP_young = 1.0f*SUM(age_seasons.OB_young) / PA_young;
    OBP_prime  = 1.0f*SUM(age_seasons.OB_prime) / PA_prime;
    OBP_older  = 1.0f*SUM(age_seasons.OB_older) / PA_older;
    -- SLG = TB / AB
    SLG_all    = 1.0f*SUM(age_seasons.TB_all) / SUM(age_seasons.AB_all);
    SLG_prime  = 1.0f*SUM(age_seasons.TB_prime) / SUM(age_seasons.AB_prime);
    SLG_older  = 1.0f*SUM(age_seasons.TB_older) / SUM(age_seasons.AB_older);
```

```

SLG_young = 1.0f*SUM(age_seasons.TB_young) / SUM(age_seasons.AB_young);
--
GENERATE
    group AS player_id,
    MIN(age_seasons.year_id) AS beg_year,
    MAX(age_seasons.year_id) AS end_year,
    --
    OBP_all + SLG_all AS OPS_all:float,
    (PA_young >= 700 ? OBP_young + SLG_young : null) AS OPS_young:float,
    (PA_prime >= 700 ? OBP_prime + SLG_prime : null) AS OPS_prime:float,
    (PA_older >= 700 ? OBP_older + SLG_older : null) AS OPS_older:float,
    --
    COUNT_STAR(age_seasons) AS n_seasons,
    SUM(age_seasons.is_young) AS n_young,
    SUM(age_seasons.is_prime) AS n_prime,
    SUM(age_seasons.is_older) AS n_older
;
};

```

If you do a sort on the different OPS fields, you'll spot Ted Williams (player ID willite01) as one of the top three young players, top three prime players, and top three old players. He's pretty awesome.

Pattern in Use

- *Where You'll Use It* — Summarizing the whole and a small number of discrete subsets: all/true/false, country/region/region/.., all visitors/cohort A/cohort B.
- *Standard Snippet* — Project dummy fields for each subset you'll track, having an ignorable value for records not in that subset. Aggregating over the whole then aggregates only over that subset
- *Hello, SQL Users* — This is a common trick in SQL cookbooks. Thanks y'all!
- *Important to Know* — You have to manufacture one field per subset. At some point you should use finer-grained grouping instead — see “Group-Flatten-Decorate” (REF) and “Cube and Rollup” (REF).
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values. Data size should decrease greatly.
- *Data Flow* — Similar to any group-and-aggregate. Combiners become highly effective as most of the values will be ignorable

Testing for Absence of a Value Within a Group

We don't need a trick to answer “which players have ever played for the Red Sox” — just select seasons with team id BOS and eliminate duplicate player ids:

```
-- Players who were on the Red Sox at some time
onetimetime_soxx_ids = FOREACH (FILTER bat_seasons BY (team_id == 'BOS')) GENERATE player_id;
onetimetime_soxx      = DISTINCT onetimetime_soxx_ids;
```

The summing trick is useful for the complementary question “which players have *never* played for the Red Sox?” You might think to repeat the above but filter for `team_id != 'BOS'` instead, but what that gives you is “which players have ever played for a non-Red Sox team?”. The right approach is to generate a field with the value 1 for a Red Sox season and the irrelevant value 0 otherwise. The never-Sox are those with zeroes for every year.

```
player_soxness     = FOREACH bat_seasons GENERATE
    player_id, (team_id == 'BOS' ? 1 : 0) AS is_sooxy;

player_soxness_g = FILTER
    (GROUP player_soxness BY player_id)
    BY MAX(is_sooxy) == 0;

never_sox = FOREACH player_soxness_g GENERATE group AS player_id;
```

Pattern in Use

- *Where You'll Use It* — Security: badges that have “entered reactor core” but no “signed in at front desk” events. Users that clicked on three or more pages but never bought an item. Devices that missed QA screening.
- *Standard Snippet* — create indicator field: `mt_f = FOREACH recs GENERATE ... , (test_of_foiness ? 1 : 0) is_foo;`; find the non-foos: `non_foos = FILTER (GROUP mt_f BY mykey) BY MAX(is_foo) == 0;`; then project just the keys: `non_foos = FOREACH non_foos GENERATE group AS mykey.`
- *Hello, SQL Users* — Another classic pattern from the lore
- *Important to Know* — If you’re thinking “gosh, once I’ve got that indicator field I could not only test its non-zeroness but sum it and average it and ...” then you’re thinking along the right lines.
- *Output Count* — As many records as the cardinality of its key, i.e. the number of distinct values. Data size should decrease dramatically.
- *Records* — List of keys
- *Data Flow* — Map, Combiner & Reducer. Combiners should be extremely effective.

Refs

- Born at the Wrong Time: Selection Bias in the NHL Draft by Robert O. Deaner, Aaron Lowen, Stephen Cobley. February 27, 2013 DOI: 10.1371/journal.pone.0057753
- The Impact Of Baseball Age-Cutoff Date Rules, waswatching.com, May 23rd, 2013
- The Boys of Late Summer, Greg Spira, April 16 2008.

Analytic Patterns: Joining Tables

In database terminology, a *join* combines the rows of two or more tables based on some matching information, known as a *key*. For example, you could join a table of names and a table of mailing addresses, so long as both tables had a common field for the user ID. You could also join a table of prices to a table of items, given an item ID column in both tables. Joins are useful because they permit people to *normalize* data — that is to say, eliminate redundant content between multiple tables — yet still bring several tables' content to a single view on-the-fly.

Joins are pedestrian fare in relational databases. Far less so for Hadoop, since MapReduce wasn't really created with joins in mind, and you have to go through acrobatics to make it work.¹ Pig's JOIN operator provides the syntactical ease of a SQL query. While Pig will shield you from hand-writing joins in MapReduce, it's still all MapReduce behind the scenes, so your joins are still subject to certain performance considerations. This section will dig into the basics of Pig joins, then explain how to avoid certain mishaps.

Matching Records Between Tables (Inner Join)

An *inner join* drops records that don't have matching keys in both tables. This means that the result of an inner join may have fewer rows than either of the original tables.

Joining Records in a Table with Directly Matching Records from Another Table (Direct Inner Join)

There is a stereotypical picture in baseball of a “slugger”: a big fat man who comes to plate challenging your notion of what an athlete looks like, and challenging the pitcher

1. Hence why you may see Hadoop joins on data scientist tech interviews.

to prevent him from knocking the ball for multiple bases (or at least far enough away to lumber up to first base). To examine the correspondence from body type to ability to hit for power (i.e. high SLG), we will need to join the people table (listing height and weight) with their hitting stats.

```
fatness = FOREACH peeps GENERATE
    player_id, name_first, name_last,
    height_in, weight_lb;
-- only players w/ statistically significant careers (about two regular-player-season's worth)
slugging_stats = FOREACH (FILTER bat_careers BY (PA > 1000))
    GENERATE player_id, SLG;
```

The syntax of the join statement itself shouldn't be much of a surprise:

```
slugging_fatness_join = JOIN
    fatness      BY player_id,
    slugging_stats BY player_id;

LIMIT slugging_fatness_join 20; DUMP @;
...
```

Disambiguating Field Names With ::

As a consequence of flattening records from the fatness table next to records from the slugging_stats table, the two tables each contribute a field named `player_id`. Although we privately know that both fields have the same value, Pig is right to insist on an unambiguous reference. The schema helpfully prefixes the field names with a slug, separated by `::`, to make it unambiguous. You'll need to run a FOREACH across the joined data, specifying the qualified names of the fields you want to keep.

Body Type vs Slugging Average

So having done the join, we finish by preparing the output:

```
FOREACH(JOIN fatness BY player_id, slugging_stats BY player_id) {
    BMI = ROUND_TO(703.0*weight_lb/(height_in*height_in),1) AS BMI;
    GENERATE bat_careers::player_id, name_first, name_last,
        SLG, height_in, weight_lb, BMI;
};
```

We added a field for BMI (Body Mass Index), a simple measure of body type. It is found by diving a person's weight by their height squared, and, since we're stuck with english units, multiplying by 703 to convert to metric. Though BMI can't distinguish between 180 pounds of muscle and 180 pounds of flab, it reasonably controls for weight-due-to-tallness vs weight-due-to-bulkiness: beanpole Randy Johnson (6'10"/2.1m, 225lb/102kg) and pocket rocket Tim Raines (5'8"/1.7m, 160lb/73kb) both have a low BMI of 23; Babe Ruth (who in his later days was 6'2"/1.88m 260lb/118kb) and Cecil Fielder (of whom Bill James wrote "...his reported weight of 261 leaves unanswered the question

of what he might weigh if he put his other foot on the scale") both have high BMIs well above 30²

```
SELECT bat.player_id, peep.nameCommon, begYear,
       peep.weight, peep.height,
       703*peep.weight/(peep.height*peep.height) AS BMI, -- measure of body type
       PA, OPS, ISO
  FROM bat_career bat
 JOIN people peep ON bat.player_id = peep.player_id
 WHERE PA > 500 AND begYear > 1910
 ORDER BY BMI DESC
;
```

A Join is a Map/Reduce Job with a secondary sort on the Table Name

The way to perform a join in map-reduce is similarly a particular application of the COGROUP we stepped through above. Even still, we'll walk through it mostly on its own. The mapper receives its set of input splits either from the bat_careers table or from the peep table and makes the appropriate transformations. Just as above (REF), the mapper knows which file it is receiving via either framework metadata or environment variable in Hadoop Streaming. The records it emits follow the COGROUP pattern: the join fields, anointed as the partition fields; then the index labeling the origin file, anointed as the secondary sort fields; then the remainder of the fields. So far this is just a transform (FOREACH) inlined into a cogroup.

```
mapper do
  self.processes_models
  config.partition_fields 1 # player_id
  config.sort_fields      2 # player_id, origin_key
RECORD_ORIGINS = [
  /bat_careers/ => ['A', Baseball::BatCareer],
  /players/      => ['B', Baseball::Player],
]
def set_record_origin!
  RECORD_ORIGINS.each do |origin_name_re, (origin_index, record_klass)|
    if config[:input_file]
      [@origin_key, @record_klass] = [origin_index, record_klass]
      return
    end
  end
  # no match, fail
  raise RuntimeError, "The input file name #{config[:input_file]} must match one of #{RECORD_ORIGINS}"
end
def start(*) set_record_origin! ; end
def recordize(vals) @record_klass.receive(vals)
def process(record)
```

2. The dataset we're using unfortunately only records players' weights at the start of their career, so you will see different values listed for Mr. Fielder and Mr. Ruth.

```

case record
when CareerStats
    yield [rec.player_id, @origin_idx, rec.slg]
when Player
    yield [rec.player_id, @origin_key, rec.height_in, rec.weight_lb]
else raise "Impossible record type #{rec.class}"
end
end
end

```

For each key, the reducer spools all the records matching that key from the initial table into an array in memory. The lastmost-named table, however, does not need to be accumulated. As the framework streams in each record in the current group, you simply compose it with each record in the accumulated array

Join in Wukong: Reducer Part.

```

reducer do
  def gather_records(group, origin_key)
    records = []
    group.each do |*vals|
      if vals[1] != origin_key # We hit start of next table's keys
        group.shift(vals)      # put it back before Mom notices
        break                  # and stop gathering records
      end
      records << vals
    end
    return records
  end

  BMI_ENGLISH_TO_METRIC = 0.453592 / (0.0254 * 0.254)
  def bmi(ht, wt)
    BMI_ENGLISH_TO_METRIC * wt / (ht * ht)
  end

  def process_group(group)
    # remainder are slugging stats
    group.each do |player_id, _, slg|
      players = gather_records(group, 'A')
      players.each do |player_id, _, height_in, weight_lb|
        # The result of the JOIN in Pig would be all the fields, keys and not, in order by origin
        # after_the_join =
        #   player_id, slg,           # fields from 'A'
        #   player_id, height_in, weight_lb # fields from 'B'
        #   ]

        # But Pig then pipelines the post-join FOREACH into the reducer, and so do we:
        yield [player_id, slg, height_in, weight_lb, bmi(height_in, weight_lb)]
      end
    end
  end
end

```

The output of the Join job will have one record for each discrete combination of A and B. As you will notice in our Wukong version of the Join, the secondary sort ensures that for each key the reducer receives all the records for table A strictly followed by all records for table B. We gather all the A records in to an array, then on each B record emit the A records stapled to the B records. All the A records have to be held in memory at the same time, while all the B records simply flutter by; this means that if you have two datasets of wildly different sizes or distribution, it is worth ensuring the Reducer receives the smaller group first. In map/reduce, the table with the largest number of records per key should be assigned the last-occurring field group label; in Pig, that table should be named last in the JOIN statement.

```
stats_and_fatness = FOREACH (JOIN fatness BY player_id, stats BY player_id)
    GENERATE fatness::player_id..BMI, stats::n_seasons..OPS;
```

Pattern in Use

- *Exercise* — Explore the correspondence of weight, height and BMI to SLG using a medium-data tool such as R, Pandas or Excel. Spoiler alert: the stereotypes of the big fat slugger is quite true.

Handling Nulls and Non-matches in Joins and Groups

It's important to understand how Null keys are handled in Join and Group operations. Briefly:

- In map-reduce, Nulls are respected as keys:
- In a single-table Pig GROUP, Nulls are also respected as keys.
- In a multi-table COGROUP, Nulls are respected as keys, *but not grouped together*
- In a JOIN operation, rows with Nulls *do not take place in the join* at all, but are *processed anyway*
- If you have a lot of Null keys, watch out: it is somewhere between costly and foolish.

When we say *null key*, we mean that if the group or join key is a scalar expression, that it has a null result; and if the key is a tuple, that all elements of the tuple are null. So

- these are null keys: Null, (Null,Null,Null), ("hi",Null,"howareyou") (even one non-null field)
- these are not: "" (empty string), 0 (zero); An empty bag {} and a bag with a tuple holding null {{}} are both not-null, but a bag cannot be used as a join or group key.

In the base Hadoop infrastructure, there's not much to understand: a key is a key, and Hadoop doesn't treat nulls specially in any way. Anything different is up to your program, and Pig does in fact supply something different.

A single-table GROUP statement does treat Nulls as keys. It's pretty easy to come up with a table having many Null values for the key you're grouping on; and if you do, all of them will be sent to the same reducer. If you actually need those keys, well, whaddaya-gonnado: sounds like one of the reducers will have to endure a bad day at work. But if you don't need the groups having Null keys, get rid of them as early as possible.

A COGROUP statement with multiple tables also treats Nulls as keys (so get rid of them if unwanted). But take note! Multi-table groups treat *each table's Nulls as distinct*. That is, if table A had 4 records with null keys, and table B had 2 records with null keys, COGROUP A by key, B by key would produce

- a row whose three fields are the null key; a bag holding the four associated records from A, and an empty bag; and
- a row whose three fields are the null key; an empty bag; and a bag holding the two associated records from B.

What do you do if you want null keys treated like any other tuple? Add an indicator field saying whether the value is null, and coalesce the actual key to non-null value. So instead of JOIN aa BY has_nulls, bb BY has_nulls, write

Join on NULL Fields.

```
JOIN
  aa BY ( (has_nulls IS NULL ? 'x' : 'Y'), (has_nulls IS NULL ? -999 : has_nulls) ),
  bb BY ( (has_nulls IS NULL ? 'x' : 'Y'), (has_nulls IS NULL ? -999 : has_nulls) );
```

Even if there are records whose value is -999, they will have 'Y' for the indicator, while the null-keyed records will have 'x', and so they will not meet up. (For your sanity, if it's possible to choose a replacement value that can't occur in the data set do so). The file `j-important_notes_about_joins.pig` in the sample code repo has a bunch more demonstrations of edge cases in groups and joins.

Pattern in Use: Inner Join

- *Where You'll Use It* — Any time you need to match records among tables. Reattaching metadata about a record to the record. Combining incidences of defective products with the manufacturing devices that made them.
- *Standard Snippet* — JOIN aa BY key, bb BY key;
- *Hello, SQL Users* — The only join that Hadoop admits is the "equi-join" — equality of values. Much more on this to follow.
- *Important to Know*

- List the tables in the statement from smallest to largest (largest table last)
- You can do a multi-way join; see the documentation
- The key does not appear in the output
- :: is for disambiguation, . is for projecting tuples in a bag. JOIN doesn't create new bags, so :: is probably what you want.
- *Output Count* — For each key that matches, the number of pairings among keys. This can be anywhere from much smaller to explosively bigger.
- *Records* — Schema of the result is the schema from each table stapled end-to-end. Values are unchanged from their input.
- *Data Flow* — Pipelinable: it's composed onto the end of the preceding map or reduce, and if it stands alone becomes a map-only job.
- *See Also*
 - DataFu's bag left outer join;
 - Left outer join on three tables: <http://datafu.incubator.apache.org/docs/datafu/guide/more-tips-and-tricks.html>
 - Time-series chapter: Range query using cross
 - Time-series chapter: Range query using prefix and UDFs (the ip-to-geo example)
 - Time-series chapter: Self-join for successive row differences
 - Advanced Pig: Sparse joins for filtering, with a HashMap (replicated)
 - The internet, for information on Bitmap index or Bloom filter joins

Enumerating a Many-to-Many Relationship

In the previous examples there's been a direct pairing of each line in the main table with the unique line from the other table that decorates it. Therefore, the output had exactly the same number of rows as the larger input table. When there are multiple records per key, however, the output will have one row for each *pairing* of records from each table. A key with two records from the left table and 3 records from the right table yields six output records.

```

player_team_years = FOREACH bat_seasons GENERATE year_id, team_id, player_id;
park_team_years  = FOREACH park_teams  GENERATE year_id, team_id, park_id;

player_stadia = FOREACH (JOIN
    player_team_years BY (year_id, team_id),
    park_team_years  BY (year_id, team_id)
) GENERATE

```

```
player_team_years::year_id AS year_id, player_team_years::team_id AS team_id,  
player_id, park_id;
```

By consulting the Jobtracker counters (map input records vs reduce output records) or by explicitly using Pig to count records, you'll see that the 77939 batting_seasons became 80565 home stadium-player pairings. The cross-product behavior didn't cause a big explosion in counts — as opposed to our next example, which will generate much more data.

Joining a Table with Itself (self-join)

Joining a table with itself is very common when you are analyzing relationships of elements within the table (when analyzing graphs or working with datasets represented as attribute-value lists it becomes predominant.) Our example here will be to identify all teammates pairs: players listed as having played for the same team in the same year. The only annoying part about doing a self-join in Pig is that you can't, at least not directly. Pig won't let you list the same table in multiple slots of a JOIN statement, and also won't let you just write something like "mytable_dup = mytable;" to assign a new alias³. Instead you have to use a FOREACH or somesuch to create a duplicate representative. If you don't have any other excuse, use a project-star expression: p2 = FOREACH p1 GENERATE *;. In this case, we already need to do a projection; we feel the most readable choice is to repeat the statement twice.

```
-- Pig disallows self-joins so this won't work:  
wont_work = JOIN bat_seasons BY (team_id, year_id), bat_seasons BY (team_id, year_id);  
"ERROR ... Pig does not accept same alias as input for JOIN operation : bat_seasons"
```

That's OK, we didn't want all those stupid fields anyway; we'll just make two copies and then join the table copies to find all teammate pairs. We're going to say a player isn't their own teammate, and so we also reject the self-pairs.

```
p1 = FOREACH bat_seasons GENERATE player_id, team_id, year_id;  
p2 = FOREACH bat_seasons GENERATE player_id, team_id, year_id;  
  
teammate_pairs = FOREACH (JOIN  
    p1 BY (team_id, year_id),  
    p2 by (team_id, year_id)  
) GENERATE  
    p1::player_id AS pl1,  
    p2::player_id AS pl2;  
teammate_pairs = FILTER teammate_pairs BY NOT (pl1 == pl2);
```

As opposed to the slight many-to-many expansion of the previous section, there are on average ZZZ players per roster to be paired. The result set here is explosively larger:

3. If it didn't cause such a surprisingly hairy set of internal complications, it would have long ago been fixed

YYY pairings from the original XXX player seasons, an expansion of QQQ⁴. Now you might have reasonably expected the expansion factor to be very close to the average number of players per team, thinking “QQQ average players per team, so QQQ times as many pairings as players.” But a join creates as many rows as the product of the records in each tables’ bag — the square of the roster size in this case — and the sum of the squares necessarily exceeds the direct sum.

The 78,000 player seasons we joined onto the team-parks-years table In contrast, a similar JOIN expression turned 78,000 seasons into 2,292,658 player-player pairs, an expansion of nearly thirty times

(A simplification was made)⁵

Joining Records Without Discarding Non-Matches (Outer Join)

The Baseball Hall of Fame is meant to honor the very best in the game, and each year a very small number of players are added to its rolls. It’s a significantly subjective indicator, which is its cardinal virtue and its cardinal flaw — it represents the consensus judgement of experts, but colored to some small extent by emotion, nostalgia, and imperfect quantitative measures. But as you’ll see over and over again, the best basis for decisions is the judgement of human experts backed by data-driven analysis. What we’re assembling as we go along this tour of analytic patterns isn’t a mathematical answer to who the highest performers are, it’s a basis for centering discussion around the right mixture of objective measures based on evidence and human judgement where the data is imperfect.

So we’d like to augment the career stats table we assembled earlier with columns showing, for hall-of-famers, the year they were admitted, and a Null value for the rest. (This allows that column to also serve as a boolean indicator of whether the players were inducted). If you tried to use the JOIN operator in the form we have been, you’ll find

4. See the example code for details

5. (or, what started as a footnote but should probably become a sidebar or section in the timeseries chapter — QEM advice please) Our bat_seasons table ignores mid-season trades and only lists a single team the player played the most games for, so in infrequent cases this will identify some teammate pairs that didn’t actually overlap. There’s no simple option that lets you join on players’ intervals of service on a team: joins must be based on testing key equality, and we would need an “overlaps” test. In the time-series chapter you’ll meet tools for handling such cases, but it’s a big jump in complexity for a small number of renegades. You’d be better off handling it by first listing every stint on a team for each player in a season, with separate fields for the year and for the start/end dates. Doing the self-join on the season (just as we have here) would then give you every *possible* teammate pair, with some fraction of false pairings. Lastly, use a FILTER to reject the cases where they don’t overlap. Any time you’re looking at a situation where 5% of records are causing 150% of complexity, look to see whether this approach of “handle the regular case, then fix up the edge cases” can apply.

that it doesn't work. A plain JOIN operation keeps only rows that have a match in all tables, and so all of the non-hall-of-famers will be excluded from the result. (This differs from COGROUP, which retains rows even when some of its inputs lack a match for a key). The answer is to use an *outer join*

```
career_stats = FOREACH (
    JOIN
        batcareers BY player_id LEFT OUTER,
        batting_hof BY player_id) GENERATE
    batcareers::player_id..batcareers::OPS, allstars::year_id AS hof_year;
```

Since the batting_hof table has exactly one row per player, the output has exactly as many rows as the career stats table, and exactly as many non-null rows as the hall of fame table.

6

```
-- (sample data)
-- (Hank Aaron)... Year
```

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

In a Pig JOIN it is important to order the tables by size — putting the smallest table first and the largest table last. (You'll learn why in the "Map/Reduce Patterns" (REF) chapter.) So while a right join is not terribly common in traditional SQL, it's quite valuable in Pig. If you look back at the previous examples, you will see we took care to always put the smaller table first. For small tables or tables of similar size, it is not a big deal — but in some cases, it can have a huge impact, so get in the habit of always following this best practice.

6. Please note that the `batting_hof` table excludes players admitted to the Hall of Fame based on their pitching record. With the exception of Babe Ruth — who would likely have made the Hall of Fame as a pitcher if he hadn't been the most dominant hitter of all time — most pitchers have very poor offensive skills and so are relegated back with the rest of the crowd



A Pig join is outwardly similar to the join portion of a SQL SELECT statement, but notice that although you can place simple expressions in the join expression, you can make no further manipulations to the data whatsoever in that statement. Pig's design philosophy is that each statement corresponds to a specific data transformation, making it very easy to reason about how the script will run; this makes the typical Pig script more long-winded than corresponding SQL statements but clearer for both human and robot to understand.

Pattern in Use

- *Where You'll Use It* — Any time only some records have matches but you want to preserve the whole. All products from the manufacturing line paired with each incident report about a product (keeping products with no incident report). All customers that took a test drive matched with the past cars they bought from you (but not discarding the new customer records)
- *Standard Snippet* — FOREACH (JOIN aa BY key LEFT OUTER, bb BY key) GENERATE a::key...a::last_field,b::second_field...;
- *Hello, SQL Users* — Right joins are much more common in Pig, because you want the table size to determine the order they're listed in
- *Important to Know* — Records with NULL keys are dropped even in an outer join
- *Output Count* — At least as many records as the OUTER table has, expanded by the number of ways to pair records from each table for a key. Like any join, output size can be explosively higher
- *Data Flow* — Pipelinable: it's composed onto the end of the preceding map or reduce, and if it stands alone becomes a map-only job.

Joining Tables that do not have a Foreign-Key Relationship

All of the joins we've done so far have been on nice clean values designed in advance to match records among tables. In SQL parlance, the career_stats and batting_hof tables both had player_id as a primary key (a column of unique, non-null values tied to each record's identity). The team_id field in the bat_seasons and park_team_years tables points into the teams table as a foreign key: an indexable column whose only values are primary keys in another table, and which may have nulls or duplicates. But sometimes you must match records among tables that do not have a polished mapping of values. In that case, it can be useful to use an outer join as the first pass to unify what records you can before you bring out the brass knuckles or big guns for what remains.

Suppose we wanted to plot where each major-league player grew up — perhaps as an answer in itself as a browsable map, or to allocate territories for talent scouts, or to see

whether the quiet wide spaces of country living or the fast competition of growing up in the city better fosters the future career of a high performer. While the people table lists the city, state and country of birth for most players, we must geolocate those place names — determine their longitude and latitude — in order to plot or analyze them.

There are geolocation services on the web, but they are imperfect, rate-limited and costly for commercial use ⁷. Meanwhile the freely-available geonames database gives geo-coordinates and other information on more than seven million points of interest across the globe, so for informal work it can make a lot of sense to opportunistically decorate whatever records match and then decide what to do with the rest.

```
geolocated_somewhat = JOIN
    people BY (birth_city, birth_state, birth_country),
    places BY (city, admin_1, country_id)
```

In the important sense, this worked quite well: XXX% of records found a match. (Question do we talk about the problems of multiple matches on name here, or do we quietly handle it?)

Experienced database hands might now suggest doing a join using some sort of fuzzy-match or some sort of other fuzzy equality. However, in map-reduce the only kind of join you can do is an “equi-join” — one that uses key equality to match records. Unless an operation is *transitive* — that is, unless `a joinsto b` and `b joinsto c` guarantees `a joinsto c`, a plain join won’t work, which rules out approximate string matches; joins on range criteria (where keys are related through inequalities ($x < y$)); graph distance; geographic nearness; and edit distance. You also can’t use a plain join on an *OR* condition: “match stadiums and places if the placename and state are equal or the city and state are equal”, “match records if the postal code from table A matches any of the component zip codes of place B”. Much of the middle part of this book centers on what to do when there *is* a clear way to group related records in context, but which is more complicated than key equality.

Exercise: are either city dwellers or country folk over-represented among major leaguers? Selecting only places with very high or very low population in the geonames table might serve as a sufficient measure of urban-ness; or you could use census data and the methods we cover in the geographic data analysis chapter to form a more nuanced indicator. The hard part will be to baseline the data for population: the question is how the urban vs rural proportion of ballplayers compares to the proportion of the general populace, but that distribution has changed dramatically over our period of interest. The US has seen a steady increase from a rural majority pre-1920 to a four-fifths majority of city dwellers today.

7. Put another way, “Accurate, cheap, fast: choose any two

Pattern in Use

- *Where You'll Use It* — Any time you're geolocating records, sure, but the lessons here hold any time you're combining messy data with canonical records
- *Hello, SQL Users* — No fuzzy matches, no string distance, no inequalities. There's no built-in SOUNDEX UDF, but that would be legal as it produces a scalar value to test with equality
- *Important to Know* — Watch out for an embarrassment of riches — there are many towns named "Springfield".

Joining on an Integer Table to Fill Holes in a List

In some cases you want to ensure that there is an output row for each potential value of a key. For example, a histogram of career hits will show that Pete Rose (4256 hits) and Ty Cobb (4189 hits) have so many more hits than the third-most player (Hank Aaron, 3771 hits) there are gaps in the output bins.

To fill the gaps, generate a list of all the potential keys, then generate your (possibly holey) result table, and do a join of the keys list (LEFT OUTER) with results. In some cases, this requires one job to enumerate the keys and a separate job to calculate the results. For our purposes here, we can simply use the integer table. (We told you it was surprisingly useful!)

If we prepare a histogram of career hits, similar to the one above for seasons, you'll find that Pete Rose (4256 hits) and Ty Cobb (4189 hits) have so many more hits than the third-most player (Hank Aaron, 3771 hits) there are gaps in the output bins. To make it so that every bin has an entry, do an outer join on the integer table. (See, we told you the integers table was surprisingly useful.)

```
-- SQL Equivalent:  
SET @H_binsize = 10;  
SELECT bin, H, IFNULL(n_H,0)  
  FROM (SELECT @H_binsize * idx AS bin FROM numbers WHERE idx <= 430) nums  
  LEFT JOIN (SELECT @H_binsize*CEIL(H/@H_binsize) AS H, COUNT(*) AS n_H  
             FROM bat_career bat GROUP BY H) hist  
    ON hist.H = nums.bin  
 ORDER BY bin DESC  
;
```

Regular old histogram of career hits, bin size 100

```
H_vals = FOREACH (GROUP bat_seasons BY player_id) GENERATE  
  100*ROUND(SUM(bat_seasons.H)/100.0) AS bin;  
H_hist_0 = FOREACH (GROUP H_vals BY bin) GENERATE  
  group AS bin, COUNT_STAR(H_vals) AS ct;
```

Generate a list of all the bins we want to keep, then perform a LEFT JOIN of bins with histogram counts. Missing rows will have a null `ct` value, which we can convert to zero.

```
H_bins = FOREACH (FILTER numbers_10k BY num0 <= 43) GENERATE 100*num0 AS bin;

H_hist = FOREACH (JOIN H_bins BY bin LEFT OUTER, H_hist_0 BY bin) GENERATE
    H_bins::bin,
    ct,           -- leaves missing values as null
    (ct IS NULL ? 0 : ct) -- converts missing values to zero
;
```

Pattern in Use

- *Where You'll Use It* — Whenever you know the values you want (whether they're integers, model numbers, dates, etc) and always want a corresponding row in the output table

Selecting Only Records That Lack a Match in Another Table (anti-join)

A common use of a JOIN is to perform an effective filter on a large number of values — the big brother of the pattern in section (REF). In this case (known as an *anti-join*), we don't want to keep the selection table around afterwards

```
-- Project just the fields we need
allstars_p = FOREACH allstars GENERATE player_id, year_id;

-- An outer join of the two will leave both matches and non-matches.
scrub_seasons_jn = JOIN
    bat_seasons BY (player_id, year_id) LEFT OUTER,
    allstars_p BY (player_id, year_id);

-- ...and the non-matches will have Nulls in all the allstars slots
scrub_seasons_jn_f = FILTER scrub_seasons_jn
    BY allstars_p::player_id IS NULL;
```

Once the matches have been eliminated, pick off the first table's fields. The double-colon in `bat_seasons::` makes clear which table's field we mean. The fieldname-ellipsis `bat_seasons::player_id..bat_seasons::RBI` selects all the fields in `bat_seasons` from `player_id` to `RBI`, which is to say all of them.

```
scrub_seasons_jn_f = FOREACH scrub_seasons_jn_f
    GENERATE bat_seasons::player_id..bat_seasons::RBI;
```

Selecting Only Records That Possess a Match in Another Table (semi-join)

A semi-join is the counterpart to an anti-join: you want to find records that *do* have a match in another table, but not keep the fields from that table around.

Let's use the same example — player seasons where they made the all-star team — but only look for seasons that *were* all-stars. You might think you could do this with a join:

```
-- Don't do this... produces duplicates!
bats_g    = JOIN allstar BY (player_id, year_id), bats BY (player_id, year_id);
badness   = FOREACH bats_g GENERATE bats::player_id .. bats::HR;
```

The result is wrong, and even a diligent spot-check will probably fail to notice. You see, from 1959-1962 there were multiple All-Star games (!), and so players who appeared in both have two rows in the All-Star table. In turn, each singular row in the bat_season table became two rows in the result for players in those years. We've broken the contract of leaving the original table unchanged.

This is the biggest thing people coming from a SQL background need to change about their thinking. In SQL, the JOIN rules all. In Pig, GROUP and COGROUP rule the land, and nearly every other structural operation is some piece of syntactic sugar on top of those. So when going gets rough with a JOIN, remember that it's just a convenience and ask yourself whether a COGROUP would work better. In this case it does:

```
-- Players with no entry in the allstars_p table have an empty allstars_p bag
allstar_seasons_cg = COGROUP
    bat_seasons BY (player_id, year_id),
    allstars_p BY (player_id, year_id);
```

Now select all cogrouped rows where there was an all-star record, and project just the records from the original table.

```
-- One row in the batting table => One row in the result
allstar_seasons = FOREACH
    (FILTER allstar_seasons_cg BY (COUNT_STAR(allstars_p) > 0L))
    GENERATE FLATTEN(bat_seasons);
```

The JOIN version was equivalent to flattening both bags (GENERATE FLATTEN(bat_seasons), FLATTEN(allstars_p)) and then removing the fields we had just flattened. In the COGROUP version, neither the incorrect duplicate rows nor the unnecessary columns are created.

An Alternative to Anti-Join: use a COGROUP

As a lesson on the virtues of JOINs and COGROUPs, let's examine an alternate version of the anti-join introduced above (REF).

```
-- Players with no entry in the allstars_p table have an empty allstars_p bag
bats_ast_cg = COGROUP
    bat_seasons BY (player_id, year_id),
    allstars_p BY (player_id, year_id);
```

Select all cogrouped rows where there were no all-star records, and project the batting table fields.

```
scrub_seasons_cg = FOREACH
    (FILTER bats_ast_cg BY (COUNT_STAR(allstars_p) == 0L))
    GENERATE FLATTEN(bat_seasons);
```

There are three opportunities for optimization here. Though these tables are far to small to warrant optimization, it's a good teachable moment for when to (not) optimize.

- You'll notice that we projected off the extraneous fields from the allstars table before the map. Pig is sometimes smart enough to eliminate fields we don't need early. There's two ways to see if it did so. The surest way is to consult the tree that EXPLAIN produces. If you make the program use `allstars` and not `allstars_p`, you'll see that the extra fields are present. The other way is to look at how much data comes to the reducer with and without the projection. If there is less data using `allstars_p` than `allstars`, the explicit projection is required.
- The EXPLAIN output also shows that co-group version has a simpler map-reduce plan, raising the question of whether it's more performant.
- Usually we put the smaller table (`allstars`) on the right in a join or cogroup. However, although the `allstars` table is smaller, it has larger cardinality (barely): `(player_id, team_id)` is a primary key for the `bat_seasons` table. So the order is likely to be irrelevant.

But “more performant” or “possibly more performant” doesn’t mean “use it instead”.

Eliminating extra fields is almost always worth it, but the explicit projection means extra lines of code and it means an extra alias for the reader to understand. On the other hand, the explicit projection reassures the experienced reader that the projection is for-sure-no-doubt-about-it taking place. That’s actually why we chose to be explicit here: we find that the more-complicated script gives the reader less to think about.

In contrast, any SQL user will immediately recognize the join formulation of this as an anti-join. Introducing a RIGHT OUTER join or choosing the cogroup version disrupts that familiarity. Choose the version you find most readable, and then find out if you care whether it's more performant; the simpler explain graph or the smaller left-hand join table *do not* necessarily imply a faster dataflow. For this particular shape of data, even at much larger scale we'd be surprised to learn that either of the latter two optimizations mattered.

Analytic Patterns: Ordering Operations

Sorting All Records in Total Order

We're only going to look at players able to make solid contributions over several years, which we'll define as playing for five or more seasons and 2000 or more plate appearances (enough to show statistical significance), and a OPS of 0.650 (an acceptable-but-not-allstar level) or better.

Career Epochs.

```
career_epochs = FILTER career_epochs BY
  ((PA_all >= 2000) AND (n_seasons >= 5) AND (OPS_all >= 0.650));

career_young = ORDER career_epochs BY OPS_young DESC;
career_prime = ORDER career_epochs BY OPS_prime DESC;
career_older = ORDER career_epochs BY OPS_older DESC;
```

You'll spot Ted Williams (willite01) as one of the top three young players, top three prime players, and top three old players. Ted Williams was pretty awesome.

Sorting by Multiple Fields

Sorting on Multiple fields is as easy as adding them in order with commas. Sort by number of older seasons, breaking ties by number of prime seasons:

```
career_older = ORDER career_epochs
  BY n_older DESC, n_prime DESC;
```

Wherever reasonable, “stabilize” your sorts by adding enough columns to make the ordering unique. This ensures the output will remain the same from run to run, a best practice for testing and maintainability.

```
career_older = ORDER career_epochs
  BY n_older DESC, n_prime DESC, player_id ASC; -- makes sure that ties are always broken the same
```

Sorting on an Expression (You Can't)

Which players have aged the best — made the biggest leap in performance from their prime years to their older years? You might think the following would work, but you cannot use an expression in an ORDER..BY statement:

```
by_diff_older = ORDER career_epochs BY (OPS_older-OPS_prime) DESC; -- fails!
```

Instead, generate a new field, sort on it, then project it away. Though it's cumbersome to type, there's no significant performance impact.

```
by_diff_older = FOREACH career_epochs
    GENERATE OPS_older - OPS_prime AS diff, player_id..;
by_diff_older = FOREACH (ORDER by_diff_older BY diff DESC, player_id)
    GENERATE player_id..;
```

If you browse through that table, you'll get a sense that current-era players seem to be over-represented. This is just a simple whiff of a question, but [more nuanced analyses](#) do show an increase in longevity of peak performance. Part of that is due to better training, nutrition, and medical care — and part of that is likely due to systemic abuse of performance-enhancing drugs.

Sorting Case-insensitive Strings

There's no intrinsic way to sort case-insensitive; instead, just force a lower-case field to sort on. We don't have an interesting table with mixed-case records in the baseball dataset, but most UNIX-based computers come with a dictionary in the /usr/share directory tree. Here's how to sort that ignoring case:

Case-insensitive Sort.

```
dict      = LOAD '/usr/share/dict/words' AS (word:chararray);
sortable  = FOREACH dict GENERATE LOWER(word) AS l_word, *;
dict_nocase = FOREACH (ORDER sortable BY l_word, word) GENERATE word;
dict_case  = ORDER dict BY word DESC;
```

Note that we sorted on `l_word` and `word`: this stabilizes the sort, ensuring that even though Polish and polish tie in case-insensitivity those ties will always be resolved the same way.

Dealing with Nulls When Sorting

When the sort field has nulls, Pig sorts them as least-most by default: they will appear as the first rows for DESC order and as the last rows for ASC order. To float Nulls to the front or back, project a dummy field having the favoritism you want to impose, and name it first in the ORDER..BY clause.

Handling Nulls When Sorting.

```

nulls_sort_demo = FOREACH career_epochs
  GENERATE (OPS_older IS NULL ? 0 : 1) AS has_older_epoch, player_id..;
nulls_then_vals = FOREACH (ORDER nulls_sort_demo BY
  has_older_epoch ASC, OPS_all DESC, player_id)
  GENERATE player_id..;
vals_then_nulls = FOREACH (ORDER nulls_sort_demo BY
  has_older_epoch DESC, OPS_all DESC, player_id)
  GENERATE player_id..;

```

Floating Values to the Top or Bottom of the Sort Order

Use the dummy field trick any time you want to float records to the top or bottom of the sort order based on a criterion. This moves all players whose careers start in 1985 or later to the top, but otherwise sorts on number of older seasons:

Floating Values to the Top of the Sort Order.

```

post1985_vs_earlier = FOREACH career_epochs
  GENERATE (beg_year >= 1985 ? 1 : 0) AS is_1985, player_id..;
post1985_vs_earlier = FOREACH (ORDER post1985_vs_earlier BY is_1985 DESC, n_older DESC, player_id)
  GENERATE player_id..;

```

Pattern in Use

- *Standard Snippet* — ORDER tbl BY mykey;
- *Hello, SQL Users*
 - Usually this is part of a SELECT statement; in Pig it stands alone
 - You can't put an expression in the BY clause
- *Important to Know* — Pound-for-pound, unless followed by a LIMIT statement this is one of the most expensive operations you can perform, requiring two to three jobs and a full reduce
- *Output Count* — Unchanged
- *Records* — Unchanged
- *Data Flow* — Map-only on a sample of the data; Map and Reduce to perform the sort. In some cases, if Pig isn't confident that it will sample correctly, an extra Map-only to perform the pipelinable operations before the sample

Sorting Records within a Group

This operation is straightforward enough and so useful we've been applying it all this chapter, but it's time to be properly introduced and clarify a couple points.

Sort records within a group using ORDER BY within a nested FOREACH. Here's a snippet to list the top four players for each team-season, in decreasing order by plate appearances.

```
players_PA = FOREACH bat_seasons GENERATE team_id, year_id, player_id, name_first, name_last, PA;
team_playerslist_by_PA = FOREACH (GROUP players_PA BY (team_id, year_id)) {
    players_o_1 = ORDER players_PA BY PA DESC, player_id;
    players_o = LIMIT players_o_1 4;
    GENERATE group.team_id, group.year_id,
        players_o.(player_id, name_first, name_last, PA) AS players_o;
};
```

Ordering a group in the nested block immediately following a structural operation does not require extra operations, since Pig is able to simply specify those fields as secondary sort keys. Basically, as long as it happens first in the reduce operation it's free (though if you're nervous, look for the line "Secondary sort: true" in the output of EXPLAIN). Messing with a bag before the ORDER..BY causes Pig to instead sort it in-memory using quicksort, but will not cause another map-reduce job. That's good news unless some bags are so huge they challenge available RAM or CPU, which won't be subtle.

If you depend on having a certain sorting, specify it explicitly, even when you notice that a GROUP ..BY or some other operation seems to leave it in that desired order. It gives a valuable signal to anyone reading your code, and a necessary defense against some future optimization deranging that order¹

Once sorted, the bag's order is preserved by projections, by most functions that iterate over a bag, and by the nested pipeline operations FILTER, FOREACH, and LIMIT. The return values of nested structural operations CROSS, ORDER..BY and DISTINCT do not follow the same order as their input; neither do structural functions such as CountEach (in-bag histogram) or the set operations (REF) described at the end of the chapter. (Note that though their outputs are dis-arranged these of course don't mess with the order of their inputs: everything in Pig is immutable once created.)

```
team_playerslist_by_PA_2 = FOREACH team_playerslist_by_PA {
    -- will not have same order, even though contents will be identical
    disordered      = DISTINCT players_o;
    -- this ORDER BY does _not_ come for free, though it's not terribly costly
    alt_order       = ORDER players_o BY player_id;
    -- these are all iterative and so will share the same order of descending PA
    still_ordered  = FILTER players_o BY PA > 10;
    pa_only        = players_o.PA;
    pretty         = FOREACH players_o GENERATE
        CONCAT((chararray)PA, ':', name_first, ' ', name_last);
    GENERATE team_id, year_id,
        disordered, alt_order,
```

1. That's not too hypothetical: there are cases where you could more efficiently group by binning the items directly in a Map rather than sorting

```

    still_ordered, pa_only, BagToString(pretty, '|');
};


```

The lines `Global sort: false // Secondary sort: true` in the explain output indicate that pig is indeed relying on the free secondary sort, rather than quicksorting the bag itself in the reducer. Lastly: in current versions of Pig this does *not* extend gracefully to COGROUP — you only get one free application of ORDER BY. Here's the dump of an EXPLAIN statement for a COGROUP with a sort on each bag:

```

cogroup_tnstaaf = FOREACH (COGROUP in_a by client_ip, in_b BY client_ip) {
    ordered_by_hadoop = ORDER in_a BY ts;
    ordered_painfully = ORDER in_b BY qk;
    GENERATE ordered_by_hadoop, ordered_painfully;
};

EXPLAIN cogroup_tnstaaf;

Reduce Plan
cogroup_tnstaaf: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-17
|
|---cogroup_tnstaaf: New For Each(false,false)[bag] - scope-16
|   | RelationToExpressionProject[bag][*] - scope-8
|   | ---Project[bag][1] - scope-9
|   |
|   | RelationToExpressionProject[bag][*] - scope-12
|   | ---ordered_painfully: POSort[bag]() - scope-15
|   |   | Project[bytearray][1] - scope-14
|   | ---Project[bag][2] - scope-13
|   ...
|
Global sort: false
Secondary sort: true

```

The key things to see are that the `ordered_painfully` bag has a POSort operator — it is being sorted in-memory and with CPU cost — while the combination of Secondary sort: true and lack of POSort in front of the first field means that `ordered_by_hadoop` is being sorted for free.

Pattern in Use

- *Where You'll Use It* — Extracting top records from a group (see next). Preceding many UDFs that depend on ordering. To make your output readable. To stabilize results.
- *Hello, SQL Users* — This is not directly analogous to the ORDER BY part of a SELECT statement, as it is done to the inner bag. For users of Oracle and other databases, this is similar to a sort within a windowed query.
- *Important to Know* — If it can be applied to the records coming from the mapper, it's free. Verify by looking for Secondary sort: true in the output of EXPLAIN
- *Output Count* — Unchanged

- *Records* — Unchanged

Select Rows with the Top-K Values for a Field

On its own, `LIMIT` will return the first records it finds. What if you want to *rank* the records — sort by some criteria — so you don't just return the first ones, but the *top* ones?

Use the `ORDER` operator before a `LIMIT` to guarantee this “top K ” ordering. This technique also applies a clever optimization (reservoir sampling, see TODO ref) that sharply limits the amount of data sent to the reducers.

Let's say you wanted to select the top 20 seasons by number of hits:

TODO: Pig code

In SQL, this would be:

```
SELECT H FROM bat_season WHERE PA > 60 AND year_id > 1900 ORDER BY H DESC LIMIT 10
```

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



We've cheated on the theme of this chapter (pipeline-only operations) — sharp eyes will note that `ORDER ... LIMIT` will in fact trigger a reduce operation. We still feel that top- K belongs with the other data elimination pattern, though, so we've included it here.

Finding Records Associated with Maximum Values

For each player, find their best significant season by OPS:

```
-- For each season by a player, select the team they played the most games for.  
-- In SQL, this is fairly clumsy (involving a self-join and then elimination of  
-- ties) In Pig, we can ORDER BY within a foreach and then pluck the first  
-- element of the bag.
```

```
SELECT bat.player_id, bat.year_id, bat.team_id, MAX(batmax.Gmax), MAX(batmax.stints), MAX(team_ids)
  FROM      batting bat
INNER JOIN (SELECT player_id, year_id, COUNT(*) AS stints, MAX(G) AS Gmax, GROUP_CONCAT(team_id)
    ON bat.player_id = batmax.player_id AND bat.year_id = batmax.year_id AND bat.G = batmax.Gmax
   GROUP BY player_id, year_id
  -- WHERE stints > 1
;
```

```
-- About 7% of seasons have more than one stint; only about 2% of seasons have  
-- more than one stint and more than a half-season's worth of games  
SELECT COUNT(*), SUM(mt1stint), SUM(mt1stint)/COUNT(*) FROM (SELECT player_id, year_id, IF(COUNT(*
```

TOP(topN, sort_column_idx, bag_of_tuples) must have an explicit field — can't use an expression

Leaderboard By Season-and-league

GROUP BY year_id, lg_id

There is no good way to find the tuples associated with the minimum value. EXERCISE: make a “BTM” UDF, having the same signature as the “TOP” operation, to return the lowest-n tuples from a bag.

Top K Records within a table using ORDER..LIMIT

Most hr in a season Describe pigs optimization of order..limit

- Pulling a Section from the Middle of a Result Set: rank and filter? Modify the quantile/median code?
- Hard in SQL but easy in Pig: Finding Rows Containing Per-Group Minimum or Maximum Value, Displaying One Set of Values While Sorting by Another: - can only ORDER BY an explicit field. In SQL you can omit the sort expression from the table (use expression to sort by)
- Sorting a Result Set (when can you count on reducer order?)

Shuffle a set of records

To shuffle a set of records, we're going to apply the *Assign a unique ID* pattern to generate an arbitrary key (one that is decoupled from the records' content), and then use that to order the records.

```
DEFINE Hasher datafu.pig.hash.Hasher('sip24-32', 'rand');

vals = LOAD '$rawd/geo/census/us_city_pops.tsv' USING PigStorage('\t', '-tagSplit')
    AS (split_info:chararray, city:chararray, state:chararray, pop:int);

vals_rk = RANK vals;
vals_ided = FOREACH vals_rk {
    line_info = CONCAT((chararray)split_info, '#', (chararray)rank_vals);
    GENERATE Hasher((chararray)line_info) AS rand_id, *; -- $2..;
};

vals_shuffled = FOREACH (ORDER vals_ided BY rand_id) GENERATE $1..;
DESCRIBE vals_shuffled;

STORE_TABLE('vals_shuffled', vals_shuffled);
```

This follows the general plot of *Assign a Unique ID*: enable a hash function UDF; load the files so that each input split has a stable handle; and number each line within the split. The important difference here is that the hash function we generated accepts a seed that we can mix in to each record. If you supply a constant to the constructor (see the documentation) then the records will be put into an effectively random order, but the same random order each time. By supplying the string '`rand`' as the argument, the UDF will use a different seed on each run. What's nice about this approach is that although the ordering is different from run to run, it does not exhibit the anti-pattern of changing from task attempt to task attempt. The seed is generated once and then used everywhere. Rather than creating a new random number for each row, you use the hash to define an effectively random ordering, and the seed to choose which random ordering to apply.

Analytic Patterns: Finding Duplicate and Unique Records

Handling Duplicates

Eliminating Duplicate Records from a Table

The park_teams table has a row for every season. To find every distinct pair of team and home ballpark, use the DISTINCT operator. This is equivalent to the SQL statement
SELECT DISTINCT player_id, team_id from batting;.

```
tm_pk_pairs_many = FOREACH park_teams GENERATE team_id, park_id;
tm_pk_pairs = DISTINCT tm_pk_pairs_many;
-- -- ALT      ALT01
-- -- ANA      ANA01
-- -- ARI      PH001
-- -- ATL      ATL01
-- -- ATL      ATL02
```

Don't fall in the trap of using a GROUP statement to find distinct values:

```
dont_do_this = FOREACH (GROUP tm_pk_pairs_many BY (team_id, park_id)) GENERATE
group.team_id, group.park_id;
```

The DISTINCT operation is able to use a combiner, eliminating duplicates at the mapper before shipping them to the reducer. This is a big win when there are frequent duplicates, especially if duplicates are likely to occur near each other. For example, duplicates in web logs (from refreshes, callbacks, etc) will be sparse globally, but found often in the same log file.

The combiner may impose a minor penalty when there are very few or very sparse duplicates. In that case, you should still use DISTINCT, but disable combiners with the `pig.exec.nocombiner=true` setting.

Eliminating Duplicate Records from a Group

Eliminate duplicates from a group with the DISTINCT operator inside a nested foreach. Instead of finding every distinct (team, home ballpark) pair as we just did, let's find the list of distinct home ballparks for each team:

```
team_parkslist = FOREACH (GROUP park_teams BY team_id) {  
    parks = DISTINCT park_teams.park_id;  
    GENERATE group AS team_id, BagToString(parks, '|');  
};  
  
EXPLAIN team_parkslist;  
  
-- -- CL1      CHI02|CIN01|CLE01  
-- -- CL2      CLE02  
-- -- CL3      CLE03|CLE09|GEA01|NEW03  
-- -- CL4      CHI08|CLE03|CLE05|CLL01|DET01|IND06|PHI09|ROC02|ROC03|STL05  
  
SELECT team_id, GROUP_CONCAT(DISTINCT park_id ORDER BY park_id) AS park_ids  
    FROM park_team_years  
    GROUP BY team_id  
    ORDER BY team_id, park_id DESC  
;
```

Eliminating All But One Duplicate Based on a Key

The DataFu DistinctBy UDF selects a single record for each key in a bag.

It has the nice feature of being order-preserving: only the first record for a key is output, and all records that make it to the output follow the same relative ordering they had in the input bag,

This gives us a clean way to retrieve the distinct teams a player served in, along with the first and last year of their tenure: define DistinctBy

```
DEFINE DistinctByYear datafu.pig.bags.DistinctBy('0');  
  
pltmyrs = FOREACH bat_seasons GENERATE player_id, year_id, team_id;  
player_teams = FOREACH (GROUP pltmyrs BY player_id) {  
    -- TODO does this use secondary sort, or cause a POSort?  
    pltmyrs_o = ORDER pltmyrs.(team_id, year_id) BY team_id;  
    pltmyrs = DistinctByYear(pltmyrs);  
    GENERATE player_id, BagToString(pltmyrs, '|');  
};
```

The key is specified with a string argument in the DEFINE statement, naming the positional index(es) of the key's fields as a comma-separated list.

Selecting Records with Unique (or with Duplicate) Values for a Key

The DISTINCT operation is useful when you want to eliminate duplicates based on the whole record. But to instead find only rows having a unique record for its key, or to find only rows having multiple records for its key, do a GROUP BY and then filter on the size of the resulting bag.

On a broadcast a couple years ago, announcer Tim McCarver paused from his regular delivery of the obvious and the officious to note that second baseman Asdrubal Cabrera “is the only player in the majors with that first name”. This raises the question: how many other people in the history of baseball similarly are uniquely yclept¹? Let’s create a table for the biography site awarding players the “Asdrubal” badge if they are the only one in possession of their first name.

```
uniquely_yclept_g = GROUP    people BY name_first;
uniquely_yclept_f = FILTER  uniquely_yclept_g BY COUNT_STAR(people) == 1;
uniquely_yclept  = FOREACH uniquely_yclept_f GENERATE
    group AS name_first,
    FLATTEN(people.(name_last, player_id, beg_date, end_date));
```

This approach should be getting familiar. We group on the key (name_first) and eliminate all rows posessing more than one record for the key. Since there is only one element in the bag, the FLATTEN statement just acts to push the bag’s fields up into the record itself.

There are some amazing names in this list. You might be familiar with Honus Wagner, Eppa Rixey, Boog Powell or Yogi Berra, some of the more famous in the list. But have you heard recounted the diamond exploits of Firpo Mayberry, Zoilo Versalles, Pi Schwert or Bevo LeBourveau? Mul Holland, Sixto Lezcano, Welcome Gaston and Mox McQuery are names that really should come attached to a film noir detective; the villains could choose among Mysterious Walker, The Only Nolan, or Phenomenal Smith for their name. For a good night’s sleep on the couch, tell your spouse that your next child must be named for Urban Shocker, Twink Twining, Pussy Tebeau, Bris Lord, Boob Fowler, Crazy Schmit, Creepy Crespi, Cuddles Marshall, Vinegar Bend Mizell, or Butttercup Dickerson.

Set Operations

Set operations — intersection, union, set difference and so forth — are a valuable strategic formulation for the structural operations we’ve been learning. In terms of set operations, “Which users both clicked on ad for shirts and bought a shirt?” becomes “find the intersection of shirt-ad-clickers set with the shirt-buyers set”. “What patients either were ill but did not test positive, or tested positive but were not ill?” becomes “find the

1. yclept /iklept/: by the name of; called.

symmetric difference of the actually-ill patients and the tested-positive patients". The relational logic that powers traditional database engines is, at its core, the algebra of sets. We've actually met many of the set operations in certain alternate guises, but set operations are so important it's worth calling them out specifically.

When we say *set*, we mean an unordered collection of distinct elements. Those elements could be full records, or they could be key fields in a record — allowing us to intersect the shirt-ad-clickers and the shirt-buyers while carrying along information about the ad they clicked on and the shirt they bought.

In the next several sections, you'll learn how to combine sets in the following ways:

- *Distinct Union* ($A \cup B$) — all distinct elements that are in A or in B .
- *Set Intersection* ($A \cap B$) — all distinct elements that are in A and also in B .
- *Set Difference* ($A - B$) — all distinct elements that are in A but are *not* in B .
- *Symmetric Difference* ($a \Delta b$) — all distinct elements that are in A or in B but not both. Put another way, it's all distinct elements that are in A but not B as well as all distinct elements that are in B but not A .
- *Set Equality* ($A == B$) — every element in A is also in B . The result of the set equality operation is a boolean true or false, as opposed to a set as in the above operations.

The following table may help. The rows correspond to the kind of elements that are in both A and B ; A but not B ; and B but not A . Under the column for each operator, only the kinds of elements marked *T* will be present in the result.

Set Operation Membership.

			Union	Inters	Diff	Diff	Sym.Diff
	A	B	AB	AB	a-b	b-a	$a \Delta b$
A	T	T	T	T	-	-	-
A -	T	-	T	-	T	-	T
- B	-	T	T	-	-	T	T

The mechanics of working with sets depends on whether the set elements are represented as records in a bag or as rows in a full table. Set operations on bags are particularly straightforward thanks to the purpose-built UDFs in the Datafu package. Set operations on tables are done using a certain COGROUP-and-FILTER combination — wordier, but no more difficult. Let's start with the patterns that implement set operations on full tables.

Set Operations on Full Tables

To demonstrate full-table set operations, we can relate the set of major US cities² with the set of US cities that have hosted a significant number (more than 50) of major-league games. To prove a point about set operations with duplicates, we will leave in the duplicates from the team cities (the Mets and Yankees both claim NY).

Preparation for Set Operations on Full Tables.

```
parks      = load_parks();
main_parks = FILTER parks BY n_games >= 50 AND country_id == 'US';
major_cities = load_us_city_pops();
--
bball_city_names = FOREACH main_parks GENERATE city;
major_city_names = FOREACH major_cities GENERATE city;
```

Distinct Union

If the only contents of the tables are the set membership keys, finding the distinct union is done how it sounds: apply union, then distinct.

```
major_or_bball = DISTINCT (UNION bball_city_names, major_city_names);
```

Distinct Union (alternative method)

For all the other set operations, or when the elements are keys within a record (rather than the full record), we will use some variation on a COGROUP to generate the result.

```
combined      = COGROUP major_cities BY city, main_parks BY city;
major_or_parks = FOREACH combined GENERATE
  group AS city,
  FLATTEN(FirstTupleFromBag(major_cities.(state, pop_2011), ((chararray)NULL,(int)NULL))),
  main_parks.park_id AS park_ids;
```

The DataFu `FirstTupleFromBag` UDF is immensely simplifying. Since the city value is a unique key for the `major_cities` table, we know that the `major_cities` bag has only a single element. Applying `FirstTupleFromBag` turns the bag-of-one-tuple into a tuple-of-two-fields, and applying `FLATTEN` lifts the tuple-of-two-fields into top-level fields for state and for population. When the `city` key has no match in the `major_cities` table, the second argument to `FirstTupleFromBag` forces those fields to have `NUL` values.

As we mentioned, there are potentially many park records for each city, and so the `main_parks` bag can have zero, one or many records. Above, we keep the list of parks around as a single field.

2. We'll take "major city" to mean one of the top 60 incorporated places in the United States or Puerto Rico; see the "Overview of Datasets" (REF) for source information

Set Intersection

Records lie in the set intersection when neither bag is empty.

```
major_and_parks_f = FILTER combined BY
  (COUNT_STAR(major_cities) > 0L) AND (COUNT_STAR(main_parks) > 0L);
major_and_parks   = FOREACH major_and_parks_f GENERATE
  group AS city,
  FLATTEN(FirstTupleFromBag(major_cities.(state, pop_2011), ((chararray)NULL,(int)NULL))),
  main_parks.park_id AS park_ids;
```

Two notes. First, we test against COUNT_STAR(bag), and not SIZE(bag) or IsEmpty(bag). Those latter two require actually materializing the bag — all the data is sent to the reducer, and no combiners can be used. Second, since COUNT_STAR returns a value of type long, it's best to do the comparison against 0L (a long) and not 0 (an int).

Set Difference

Records lie in A minus B when the second bag is empty, and they lie in B minus A when the first bag is empty.

```
major_minus_parks_f = FILTER combined BY (COUNT_STAR(main_parks) == 0L);
major_minus_parks   = FOREACH major_minus_parks_f GENERATE
  group AS city,
  FLATTEN(FirstTupleFromBag(major_cities.(state, pop_2011), ((chararray)NULL,(int)NULL))),
  main_parks.park_id AS park_ids;

parks_minus_major_f = FILTER combined BY (COUNT_STAR(major_cities) == 0L);
parks_minus_major   = FOREACH parks_minus_major_f GENERATE
  group AS city,
  FLATTEN(FirstTupleFromBag(major_cities.(state, pop_2011), ((chararray)NULL,(int)NULL))),
  main_parks.park_id AS park_ids;
```

Symmetric Set Difference: (A-B)+(B-A)

Records lie in the symmetric difference when one or the other bag is empty. (We don't have to test for them both being empty — there wouldn't be a row if that were the case.)

```
major_xor_parks_f   = FILTER combined BY
  (COUNT_STAR(major_cities) == 0L) OR (COUNT_STAR(main_parks) == 0L);

major_xor_parks     = FOREACH major_xor_parks_f GENERATE
  group AS city,
  FLATTEN(FirstTupleFromBag(major_cities.(state, pop_2011), ((chararray)NULL,(int)NULL))),
  main_parks.park_id AS park_ids;
```

Set Equality

Set Equality indicates whether the elements of each set are identical — here, would tell us whether the set of keys in the major_cities table and the set of keys in the main_parks table were identical.

There are several ways to determine full-table set equality, but likely the most efficient is to see whether the two sets' symmetric difference is empty. An empty symmetric difference implies that every element of A is in B , and that every element of B is in A — which is exactly what it means for two sets to be equal.

Properly testing whether a table is empty so is a bit more fiddly than you'd think. To illustrate the problem, first whip up a set that should compare as equal to the major_cities table, run the symmetric difference stanza from above, and then test whether the table is empty:

```
major_city_names_also = FOREACH major_cities GENERATE city;
major_xor_major = FILTER
  (COGROUP major_city_names BY city, major_city_names_also BY city)
  BY ((COUNT_STAR(major_city_names) == 0L) OR (COUNT_STAR(major_city_names_also) == 0L));

-- Does not work
major_equals_major_fail = FOREACH (GROUP major_xor_major ALL) GENERATE
  (COUNT_STAR(major_xor_major) == 0L ? 1 : 0) AS is_equal;
```

The last statement of the code block attempts to measure whether the count of records in major_xor_major is zero. And if the two tables were unequal, this would have worked. But major_xor_major is empty and so *the FOREACH has no lines to operate on*. The output file is not a 1 as you'd expect, it's an empty file.

Our integer table to the rescue! Actually we'll use her baby brother *one_line.tsv*: it has one record, with fields uno (value 1) and zilch (value 0). Instead of a GROUP..ALL, do a COGROUP of one_line on a constant value 1. Since there is exactly one possible value for the group key, there will be exactly one row in the output.

```
one_line = LOAD '$data_dir/stats/numbers/one_line.tsv' AS (uno:int, zilch:int);

-- will be `1` (true)
major_equals_major = FOREACH (COGROUP one_line BY 1, major_xor_major BY 1)
  GENERATE (COUNT_STAR(major_xor_major) == 0L ? 1 : 0) AS is_equal;

-- will be `0` (false)
major_equals_parks = FOREACH (COGROUP one_line BY 1, major_xor_parks BY 1)
  GENERATE (COUNT_STAR(major_xor_parks) == 0L ? 1 : 0) AS is_equal;
```

TODO: clean up transition to set ops on groups

To demonstrate set operations on grouped records, let's look at the year-to-year churn of mainstay players³ on each team.

Other applications of the procedure we follow here would include analyzing how the top-10 products on a website change over time, or identifying sensors that report values over threshold in N consecutive hours (by using an N-way COGROUP).

Constructing a Sequence of Sets

To construct a sequence of sets, perform a self-cogroup that collects the elements from each sequence key into one bag and the elements from the next key into another bag. Here, we group together the roster of players for a team's season (that is, players with a particular `team_id` and `year_id`) together with the roster of players from the following season (players with the same `team_id` and the subsequent `year_id`).

Since it's a self-cogroup, we must do a dummy projection to make new aliases (see the earlier section on self-join for details).

```
y1 = FOREACH sig_seasons GENERATE player_id, team_id, year_id;
y2 = FOREACH sig_seasons GENERATE player_id, team_id, year_id;

-- Put each team of players in context with the next year's team of players
year_to_year_players = COGROUP
    y1 BY (team_id, year_id),
    y2 BY (team_id, year_id-1)
;
-- Clear away the grouped-on fields
rosters = FOREACH year_to_year_players GENERATE
    group.team_id AS team_id,
    group.year_id AS year_id,
    y1.player_id AS pl1,
    y2.player_id AS pl2
;
-- The first and last years of existence don't have anything interesting to compare, so reject them
rosters = FILTER rosters BY (COUNT_STAR(pl1) == 0L OR COUNT_STAR(pl2) == 0L);
```

Set Operations Within a Group

The content of `rosters` is a table with two key columns: team and year; and two bags: the set of players from that year and the set of players from the following year.

Applying the set operations lets us describe the evolution of the team from year to year.

```
roster_changes_y2y = FOREACH rosters {
    -- Distinct Union (doesn't need pre-sorting)
    either_year = SetUnion(pl1, pl2);
    -- The other operations require sorted bags.
```

3. using our definition of a significant season: post-1900 and 450 or more plate appearances

```

pl1_o = ORDER pl1 BY player_id;
pl2_o = ORDER pl2 BY player_id;

-- Set Intersection
stayed      = SetIntersect(pl1_o, pl2_o);
-- Set Difference
y1_departed = SetDifference(pl1_o, pl2_o);
y2_arrived  = SetDifference(pl2_o, pl1_o);
-- Symmetric Difference
non_stayed   = DIFF(y1_departed, y2_arrived);
-- Set Equality
is_equal     = ( (COUNT_STAR(non_stayed) == 0L) ? 1 : 0);

GENERATE year_id, team_id,
either_year, stayed, y1_departed, y2_arrived, non_stayed, is_equal;
};

```

The Distinct Union (A union B, which we'll find using the DataFu `SetUnion` UDF) describes players on the roster in either year of our two-year span.

```
either_year = SetUnion(pl1, pl2);
```

All the DataFu set operations here tolerate inputs containing duplicates, and all of them return bags that contain no duplicates. They also each accept two or more bags, enabling you to track sequences longer than two adjacent elements.

As opposed to `SetUnion`, the other set operations require sorted inputs. That's not as big a deal as if we were operating on a full table, since a nested `ORDER BY` makes use of Hadoop's secondary sort. As long as the input and output bags fit efficiently in memory, these operations are efficient.

```

pl1_o = ORDER pl1 BY player_id;
pl2_o = ORDER pl2 BY player_id;

```

The Set Intersection (A intersect B, which we'll find using the DataFu `SetIntersect` UDF) describes the players that played in the first year and also stayed to play in the second year.

```
stayed      = SetIntersect(pl1_o, pl2_o);
```

The Set Difference (A minus B, using the `SetDifference` UDF) contains the elements in the first bag that are not present in the remaining bags. The first line therefore describes players that did *not* stay for the next year, and the second describes players that newly arrived in the next year.

```

y1_departed = SetDifference(pl1_o, pl2_o);
y2_arrived  = SetDifference(pl2_o, pl1_o);

```

The Symmetric Difference contains all elements that are in one set or the other but not both. You can find this using either `(A minus B) union (B minus A)` — players who either departed after the first year or newly arrived in the next year — or `((A union B)`

`minus (A intersect B))` — players who were present in either season but not both seasons.

```
non_stayed = SetUnion(y1_departed, y2_arrived);
```

Set Equality indicates whether the elements of each set are identical — here, it selects seasons where the core set of players remained the same. There's no direct function for set equality, but you can repurpose any of the set operations to serve.

If A and B each have no duplicate records, then A and B are equal if and only if

- `size(A) == size(B)` AND `size(A union B) == size(A)`
- `size(A) == size(B)` AND `size(A intersect B) == size(A)`
- `size(A) == size(B)` AND `size(A minus B) == 0`
- `size(symmetric difference(A,B)) == 0`

For multiple sets of distinct elements, A, B, C... are equal if and only if all the sets and their intersection have the same size: `size(intersect(A,B,C,...)) == size(A) == size(B) == size(C) == ...`

If you're already calculating one of the functions, use the test that reuses its result. Otherwise, prefer the A minus B test if most rows will have equal sets, and the A intersect B test if most will not or if there are multiple sets.

```
is_equal = ( (COUNT_STAR(non_stayed) == 0L) ? 1 : 0);
```

TODO also show and op that labels by (a, b, both)

PART III

Strategy: Practical Data Explorations

Spatial and Geographic Analysis

Having conquered time (or, at least, learned basics of timeseries analysis), let's move on to space. Spatial analysis works with *data attached to points, paths and regions*, where the *fundamentally interesting relationships are among nearby objects*. This is some of our favorite material in the book for two reasons: because it's really useful, and because it will extend your understanding of map-reduce in a significant way.

Spatial Analytics is a Useful Tool

Problems with a directly geographic aspect appear naturally in all sorts of human endeavors: "Where should I put this cell tower / oil well / coffee shop?". Analysis across billions of GPS paths on millions of routes not only allows a driving directions app to direct you to the correct lane when changing roads, it will power the self-driving cars of the (near?) future. **Farmers are improving crop yields and reducing environmental impact** by combining data from suppliers, other growers, satellite feeds, and government agencies to fine-tune what strains they plant and what pests and diseases they act to prevent. Wind farm companies use petabytes of weather data to predict **optimize turbine locations and predict their yield**. It is an increasingly essential piece of any high-stakes effort in marketing, agriculture, security, politics, and any other field where location is a key variable.

Geospatial analysis is also incredibly useful because geography is such a good proxy for many key features you can't easily measure. You wouldn't think the 32 bits of an IP address described the weather. But as you saw in the last chapter an IP address can (imperfectly but actionably) be related to a geolocation, and a geolocation can (imperfectly but actionably) be related to a weather report. Without violating a person's privacy, you can know when to switch from promoting sunscreen to Sydneysiders and galoshes to Glaswegians to quite profitably doing the reverse. With the right auxilliary dataset you can treat geographic coordinates as an imperfect but often actionable stand-in for

census demographics, average daily temperature, political preference, crime rate — you get the idea.

To keep things concrete and relevant to the typical reader we're going to focus on geo-spatial analysis, where the data is specifically located on the surface of our Earth. But the methods we'll demonstrate here extend naturally to anything with a spatial aspect, like factory floor optimization, customer flow through a mall. What's more, it's surprisingly easy to extend these methods to three and higher dimensions (see the exercise at the end of the chapter (REF)), good news for anyone looking to analyze ocean currents, electronic chip layout or brain activity. And even beyond explicitly spatial problems, every reader looking to master machine learning techniques will find that this chapter provides an important reference point for thinking in the highly-dimensional spaces those algorithms work in.

Spatial Analytics is Good for Your Brain

The essential element of spatial analytics is to operate on shapes within context of what is spatially nearby. We can do so by coaching Hadoop to:

1. partition space into coarse-grained tiles
2. assign each tile to a reducer (equivalently, group each tile's objects into a bag)
3. ensure that everything potentially relevant for the objects on a tile finds its way there
4. eliminate any potentially-but-not-actually relevant objects and any duplicated results

Since a point has no extent, nominating its context is straightforward: send it to the tile it lives on. Spatially aggregating points on a tile requires exactly and only the occupants of that tile. As you'll see in our first example, that means it's no harder than the grouping operations you mastered back in Chapter 6 (REF).

In contrast, a multi-point, a line, or any other shape having spatial extent might cross one or many tiles, or even wholly contain them. It might have gaps and holes, might cross from 180 degrees longitude to -180 degrees longitude (or by covering one of the poles) might even span all 360 degrees of longitude. And while it's cheap to determine whether pairs of points or rectangles intersect, touch, lie within a given distance, or whatever other definition of "spatially relevant" is in play, you need to perform the corresponding operations on the complex shapes eventually, which can be quite expensive. That leads to why bullet point number 4 appears above, and why we think this chapter is so important for a deep understanding of orchestrating big data operations.

What we need to do is scatter each object to all the groups where it *might* be relevant, knowing that (a) it might be grouped with objects that are not actually relevant, and (b) relevant operations may be duplicated within multiple groups. A person standing in

Lesotho is within the South Africa's bounding box (and so potentially relevant) but is not actually within South Africa (and so is not relevant for the "Within" relationship). Territories of the UK and France lie within 60 km of each other not only from Dover to Calais in Europe but also fromMontserrat to Guadeloupe in the Caribbean.

The core spatial operations allow us to segment and . The key new skill as map/reduce coaches is to do so correctly and efficiently: without expensive processing, without requiring context that might reside on some other machine, and without an explosion of midstream data, or proliferation of not-actually-relevant objects to consider, or an infection of indeterminately duplicate output records.

Those strategies aren't unique to the spatial case, and so what we're really trying to do in this chapter is equip you do deal with situations where determining which objects should be related in context is complex and can't be done locally. For example, one way to find pairs of similar documents involves deriving each document's most prominent keywords, then matching each document to all others that share a significant fraction of its keywords. In the spatial analytics case, our cascading love triangles from London to Paris to Lyon to Milan (REF (to E&C preamble)) threatened to crowd the whole world into a single dance hall. The document similarity case presents the same problem. A document mentioning Paris, Lyon, and Milan is a candidate match for one mentioning London+Paris+Lyon and for one mentioning Lyon+Milan+Rome — but London+Paris+Lyon and Lyon+Milan+Rome don't need to be compared. We want to ensure the necessary pairings are considered, without allowing the chain of candidate matches from London+Paris+Lyon, through Lyon+Milan+Rome, on over to Busan+Osaka+Tokyo to land on the same library table. It's the same problem in different guise, and the essential intuition you build here will carry over.

What We'll Cover

We'll start by demonstrating spatial aggregations on points: for example, counting the number of UFO sightings per area over a regular grid. This type of aggregation is a frontline tool of spatial analysis, a great way to summarize a large data set, and one of the first things you'll do to Know Thy Data. It draws on methods you've already learned, giving us a chance to introduce some terminology and necessary details. Then we'll go from spatial grouping of a single dataset to demonstrate a point-wise spatial join: matching points in a table with all nearby points in another table.

Next is a gallery tour of the basic spatial operations: how to find a shape's area or bounding box, its intersection with another shape, and so forth. We won't spend much of your time here, but it's worth having something to refer to. The real fun comes as the notion of "nearby" becomes less and less predictable in advance.

Spatial aggregations on regions — e.g. smoothing country-by-country crop production data onto a uniform grid — demands more subtlety than aggregations of points. In places where multiple regions appear, you must weight their contributions correctly,

and in places where only one region is present you'd like to avoid extra work. We handle that using a quad-tree tiling (aka "quadtiles"), a superbly elegant tool for partitioning spatially-nearby objects onto common reducers with minimal waste.

The quadtile scheme not only helps to partition the data efficiently but also orders it so that spatially-nearby objects are generally nearby in quadtile order. That lets us perform a spatial join of points with *regions* using nothing more than Hadoop's native sort and a low-memory-overhead data structure. This is the key material in the chapter, and we'll step through it in detail.

Lastly, we'll demonstrate how to handle the case where it's difficult in advance to even know how to spatially partition data. You see, one way to determine which records are spatially relevant — the one all the operations to that point in the chapter will have used — is to define a fixed distance and only relate objects within that nearby-ness threshold. When you're matching shapes with objects less than a certain distance away, You can set an upper bound in advance on what "nearby" means: the specified distance. But when you want to map against not "*any nearby*" objects but against "*the nearest*" object — which might be a three-minute walk or might be a three-day sail — things get more complicated. There's a wonderful trick for doing these kinds of "nearest object" queries without overwhelming your cluster. We'll use it to combine our record of every baseball game played against the historical weather data, and learn an important truth about truth and error along the way.

Let's start off in the best way possible: with a tool for turning lots of data into manageable insight.

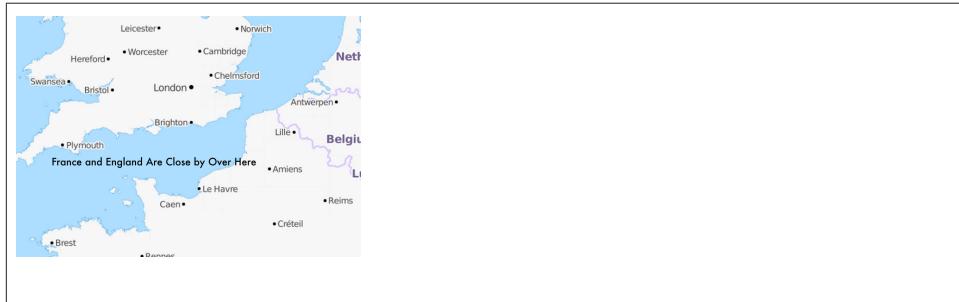


Figure 9-1. The territories of France and the UK are close by in Europe



Figure 9-2. And also in the Caribbean



Figure 9-3. South Africa contains Lesotho: Politics trumps Topology



Geographic Analysis is a well developed field. The number of hard problems with valuable solutions in intelligence, petroleum, natural resources and other such industries gave rise to highly sophisticated products capable of processing massive quantities of data well before these kids with their Hadoops and their JSONs came along. As you might expect, however these products possess a correspondingly steep price tag and learning curve and often integrate poorly outside their domain. Most importantly, they are bound by the limits of traditional database technology in many aspects. So even users of these powerful GIS tools can benefit by augmenting them with Big Data analytics. But our focus will be on the reader who needs to get stuff done with geographic data, and the less they have to learn a new field the better. We're going to use GeoJSON, a newly-developed standard for geographic data (friendly to general purpose analytic tools and to rendering data in the browser), rather than Arcview shapefiles or other geospatial formats (seamless integration with industry products). We've simplified concepts and minimized jargon to keep the focus on readable, powerful scripts that give actionable insight.

Smoothing Pointwise Data Locally (Spatial Aggregation of Points)

We will start, as we always do, by applying patterns that turn Big Data into Much a Less Data. In particular, A great tool for visualizing a large spatial data set

The straightforward approach we'll take is to divide the world up into a grid of tiles and map the position of each point onto the unique grid tile it occupies. We can then group on each tile

Area of a spherical segment is $2\pi R^2 h$ — so for lat from equator to 60

```
%default binsz 2.0
-- place into half-degree bins -- ~ 120x50 cells for US
gridded = FOREACH sightings GENERATE
    FLOOR(lng * $binsz) / $binsz AS bin_x,
    FLOOR(lat * $binsz) / $binsz AS bin_y;
-- number density
grid_cts = FOREACH (GROUP gridded BY (bin_x, bin_y))
GENERATE
    group.bin_x, group.bin_y,
    COUNT_STAR(gridded) AS ct;
```

- US: -125 24 to -66, 50 (-124.7625, 24.5210, -66.9326, 49.3845) — about 60 x 26

Creating a Spatial Density Map

Map points to quad cells, plot number density of airports as a heat map

Then geonames places — show lakes and streams (or something nature-y) vs something urban-y

(just call out that rollup, summing trick, or group-decorate-flatten would work: do no pursue)

Do that again, but for a variable: airport flight volume — researching epidemiology

We can plot the number of air flights handled by every airport

```
%default binsz 2.0
-- place into half-degree bins -- ~ 120x50 cells for US
gridded = FOREACH sightings GENERATE
    FLOOR(lng * $binsz) / $binsz AS bin_x,
    FLOOR(lat * $binsz) / $binsz AS bin_y,
    n_flights;
-- number density
grid_cts = FOREACH (GROUP gridded BY (bin_x, bin_y))
    GENERATE
        group.bin_x, group.bin_y,
        COUNT_STAR(gridded) AS ct,
        SUM(n_flights) AS tot_flights;
```

An epidemiologist or transportation analyst interested in knowing the large-scale flux of people could throughout the global transportation network

Pattern Recap: Spatial Aggregation of Points

- *Generic Example* — group on tile cell, then apply the appropriate aggregation function
- *When You'll Use It* — as mentioned above: summarizing data; converting point samples into a continuous value; smoothing out spatial variation; reassigning spatial data to grid-aligned regions
- *Exercises* --
- *Important to Know* --
 - A **Dot Distribution Map** is in some sense the counterpart to a spatial average — turning data over a region into data at synthesized points

Matching Points within a Given Distance (Pointwise Spatial Join)

Now that you've learned the spatial equivalent of a GROUP BY, you'll probably be interested to learn the spatial equivalent of COGROUP and JOIN. In particular, let's demonstrate how to match all points in one table with every point in another table that are less than a given fixed distance apart.

Our reindeer friends would like us to help determine what UFO pilots do while visiting Earth. Let's combine the NUFORC data set of 60,000+ documented UFO sightings with the 7 million points of interest from the Geonames dataset to determine whether UFOs are more frequently sighted nearby airports, parks, schools, or churches.

It's up to us to define what "nearby" means. Let's start by casting a fairly wide 16km (10-mile) net. It's the distance of visibility on a pretty clear day, and seems like a reasonable upper bound on how far I'd travel to hide my UFO while sightseeing.

What we're going to do is this:

- For every UFO sighting, figure out all tiles that potentially contain points within 16 km of the sighting
- For every point of interest, figure out the single tile it belongs to
- Join the points of interest by tile with all UFO sightings potentially relevant to the tile
- Select only the UFO sighting - point of interest matches that are actually within the 16km threshold.

Distance on a Sphere

To send each UFO sighting to all relevant grid cells, we need to find the bounding box of its circle of distance — the highest and lowest excursions in longitude and latitude. That purpose is answered by the `TilesForGeoCircle` UDF, which generates the list of relevant tiles given the point, distance and grid scheme. So far, we've been casually pretending that our coordinates live on a uniform Cartesian grid, but that's not the case. Our longitude and latitude coordinates exist on a sphere, which can greatly complicate some operations and betray your intuition. Finding the bounding box in longitude and latitude is a cardinal example of how spherical geometry can complicate matters, and so it's worth going into how the answer is calculated.

Finding the latitude values is easy: the fraction of the earth's circumference that our given distance covers is the same as the fraction of 360 degrees that our range of latitudes cover.

```
earth_radius = 6378.1370
earth_circum = earth_radius * 2 * pi # pretty close to 40,000 km

# arc distance in degrees, along north-south arc
arc_dist_degs = 360 * (distance in km / earth_circum)
```

As an example, the drawing below shows a circle of 1400 km¹ centered right on the equator near Libreville, Gabon. Our 1400 km is about 3.5% of the earth's circumference, pretty near 12.5 degrees of arc: our bounding box will extend from -12.58 on the south to 12.58 on the north. You can count off grid cells in our diagram below to see that we've drawn it correctly so far.

If we were on a regular flat Cartesian grid, finding the westmost extent would be obvious: hold the latitude constant and walk west for the given distance. (Or in this case, paddle.) Since we are at the equator, this works correctly, and in fact the extent of arc is the same east-west as it is north-south. At every place other than along the equator, however, this answer is wrong.

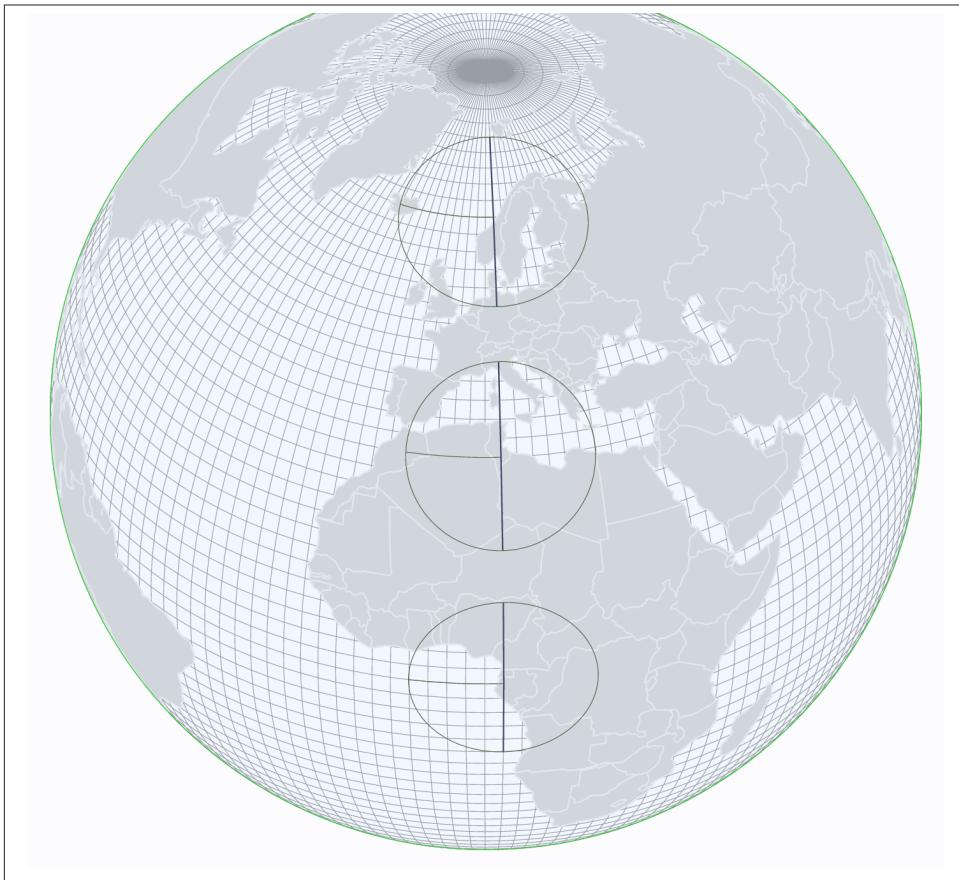


Figure 9-4. Two circles, both alike in radius, in Libreville and Reykjavik we lay our scene

1. the equivalent distance of Boston-Chicago or Paris-Warsaw

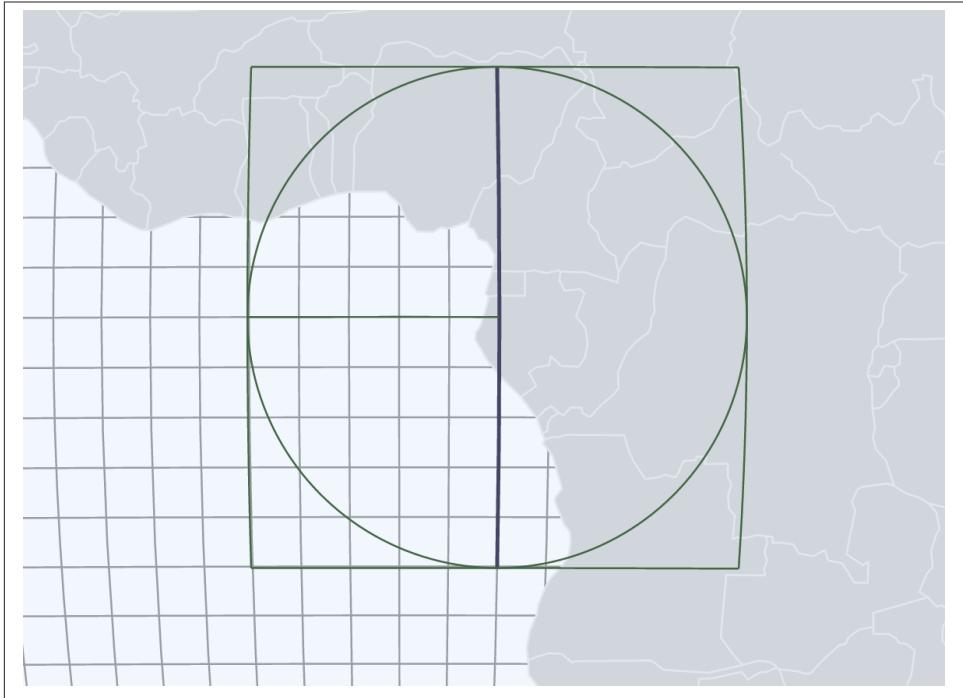


Figure 9-5. Everything is greatly easier on the Equator because everything is a Great Circle

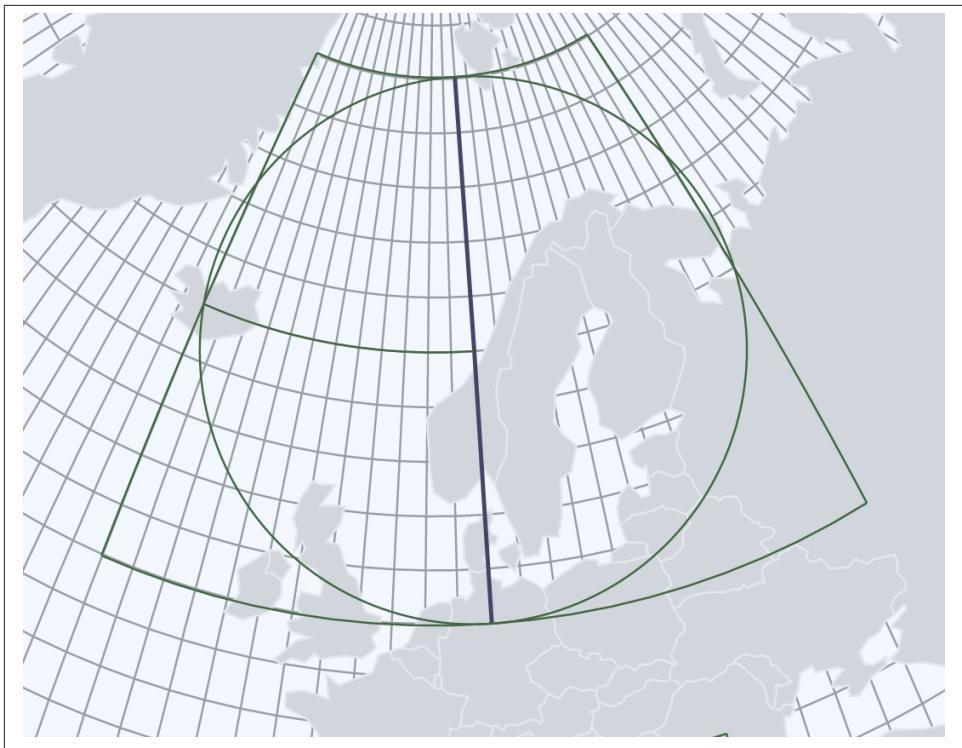


Figure 9-6. The point of westmost longitude is not parallel to the center

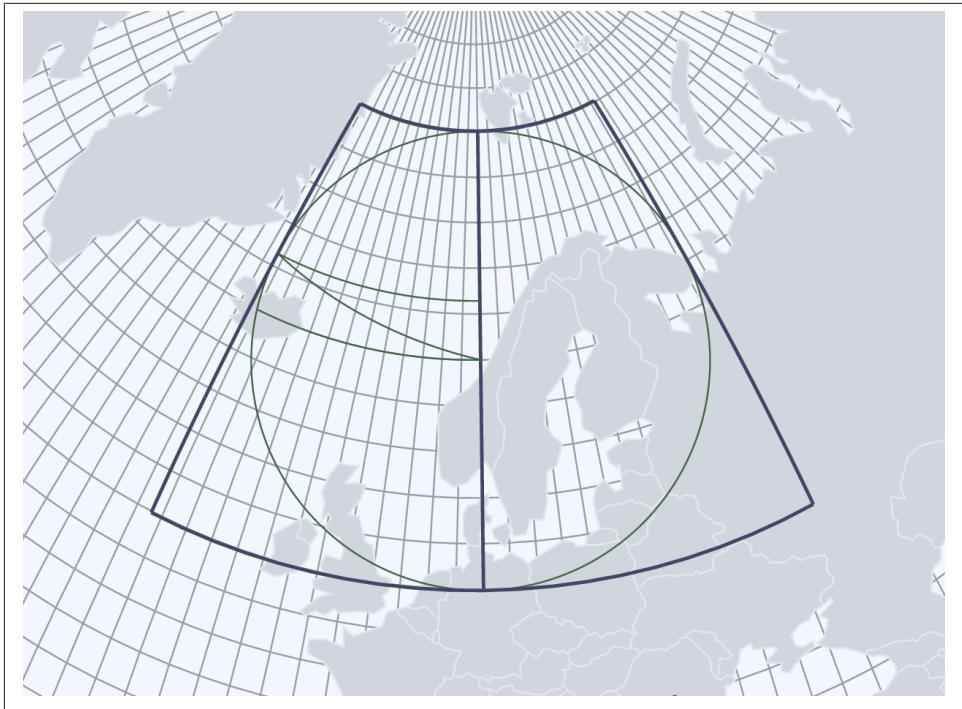


Figure 9-7. Gridline is tangent at the actual westmost longitude

To see why, let's look at another 1400km-radius circle, centered on the same longitude but this time at 65 degrees north latitude — extending roughly from Reykjavik in Iceland to Arkhangelsk in Russia.

Applying the logic that worked at the equator, we've drawn a "horizontal" arc following the parallel of latitude from the center of our circle west towards Reykjavik. The great-circle distance of that arc is exactly 1400-km² and its endpoint is 30.042 degrees of longitude west of the center.

But notice what happens when we apply those offsets ($[\pm 30.042, \pm 12.58]$) to the circle's center ($[10^{\circ}\text{E}, 65^{\circ}\text{N}]$) to construct the bounding box from $[10^{\circ} - 30.042, 65^{\circ} - 12.58]$ to $[10^{\circ} + 30.042, 65^{\circ} + 12.58]$. Parts of the circle jut outside of the box!

If you look carefully, you'll see that the lines of constant longitude "come together" faster than the curve of the circle does.

2. There's another potential pitfall which we won't go into. If you actually flew strictly along the horizontal arc we drew from the center to its end, you would travel a bit over 1413 kilometers. The shortest route from the center to the endpoint has you deviate slightly north along a "great circle" path, and takes the expected 1400 km

Travelling 1400 km (12.58 degrees of arc) north from the center along a meridian brought us to a point where the constant-latitude gridlines were perfectly tangent to the circle. That meant that travelling along the circle in either direction necessarily departed from that farthest gridline, making it the maximum gridline touched. In contrast, travelling 1400 km from the center along the 65th parallel did not bring us to a point where the constant-longitude gridlines were tangent to the circle. That location is given by the following equations:

```
# the arc distance
arc_dist_rad      = 2 * PI * distance_in_km / earth_circum
lat_rad           = lat * PI / 180
tangent_lat       = arcsin( sin(lat_rad)      / cos(arc_dist_rad) ) * PI / 180
delta_lng         = arcsin( sin(arc_dist_rad) / cos(lat_rad) ) * PI / 180
bounding_box       = [ [lng-delta_lng, lat-arc_dist_degs], [lng+delta_lng, lat+arc_dist_degs] ]
```

The `tangent_lat` statement provides the latitude where the constant-latitude gridlines are actually tangent, making it the location of farthest longitude. // It always lies towards the nearest pole by some amount. The `delta_lng` statement finds proper the arc distance west and east for our bounding box, which the last statement calculates explicitly.

There are still other details to attend to — the box could cross over the antimeridian from 180 to -180 degrees longitude, causing it to split into “two” bounding boxes; and it could extend north over the pole, causing it to extend through all 360 degrees(!) of longitude. We’ve written a UDF that finds the bounding box correctly and handles all those edge case, so that’s what we’ll use. But please learn the lesson from this particularly mild instance: spatial geometry operations can get astonishingly thorny. Going beyond what the libraries provide may cause you to learn more mathematics than you’d prefer.

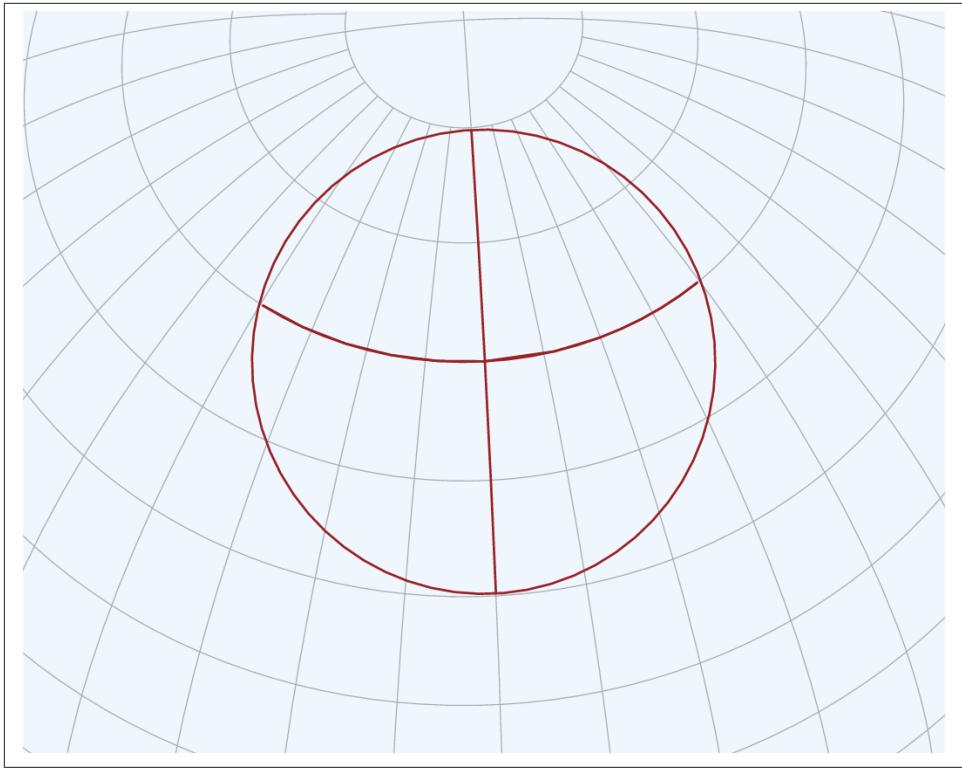


Figure 9-8. Min/Max Longitudes are not at the same latitude as the center

Spatial Aggregations on Regions

While spatially aggregating pointwise data required nothing terribly sophisticated, doing the same for objects with spatial extent will motivate a couple new tricks. As an example, let's take agricultural production for each country³ and smooth it onto a uniform grid. (We'll find some interesting connections to this data in the final chapter).

Fair warning: though the first version of this script we demonstrate for you is correct as it stands, it will be exceptionally inefficient. But we want to demonstrate the problem and then discuss its elegant solution.

The key first step is to assemble a table giving the weight of each region's contribution to the tile.

3. downloaded from the website of FAOSTAT, The Statistical Division of the Food and Agriculture Organization of the United Nations

- Break up the geographic shapes onto tiles. Luxembourg breaks up into two tiles representing XXX% and 1-XXX% of its area. Russia breaks into XXX tiles (!).
- Bring all fragments one each tile together. For example, the southern of the tiles holding Luxembourg also holds parts of Belgium, France, and Germany.
- Find the area fraction of each segment on its tile along with the combined shapes. Our example four-country tile is covered by XXX% Luxembourg, XXX% Belgium, XXX% France, and XXX% Germany. Meanwhile, most of Russia's tiles lie 100% within its interior.

```
(... code ...)
quad_area_frac: fragment's pull / sum of fragments' pulls
(id) (name) (area on quad) (area of whole) (fraction of occupied area on quad) (shape)
```

With this table in hand, a simple join of the each region's statistics with the table of weights gives the aggregated fraction. For rate or aggregated statistics, such as commodity price index or proportion of imports to exports, we only need to weight the contribution by fraction-on-tile. The relative price index of eggs in Luxembourg in 2011 was 118, 135 in France 170 in Germany and 144 in Belgium, and so our four-country exemplar scale has an averaged price index of XXX (EQUATION).

For counting statistics such as “number of head of cattle”, we need to first use the fraction-of-country figure to find the contributed count, and then use the Fraction-on-tile to generate the aggregate. It isn't fair to pit all of Germany's XXX head against Luxembourg's XXX on that tile; we should consider that Germany weighed in with a proportional XXX head (XXX%) against Luxembourg's XXX head (XXX%).



Yes, it's problematic that we are smearing all of each country's production uniformly throughout its interior. Florida grows far more cane sugar than Alaska. Your options? First, it might not matter. Predictive analytics algorithms are often quite happy to accept imperfect data if it means more signal. The audience for an "Every possible statistic about where I'm Standing" mobile app would have to accept that we don't quite have *everything* yet; just be upfront about the data's imperfections. If that's not acceptable, you can try to find better data: through commercial vendors, or through each country's agricultural statistics website, if any. A big expense either way. Too often, people are reluctant to pursue the middle ground of supplementing the data with their own expertise. Torturing insufficient data into a misleading result is a mortal sin, but acting on only that which can be quantified will send you to perdition just the same. We all just agreed that there are no sugar cane farms in Alaska — finding out what things Alaska, Greenland, Siberia and the Australian Outback *do* produce, and excluding those regions from your analysis otherwise, would yield a large improvement at modest effort. Even if we don't know exactly how production is distributed within a country, we know that (a) this only matters for countries so large they span different ecologies; (b) the smaller countries give you a good sense of where those ecologies are. You could hold out the too-large countries from a first pass and use that background distribution to allocate. This is only justifiable as a way to turn a bad estimate into a serviceable one, but sometimes that's enough. The best way to get good at this is to watch masters at work. Besides peer-reviewed articles in your field, one of the best things you can do to advance along your path is to regularly read the data-journalism articles on the FiveThirtyEight blog. Nate Silver and Emily Oster especially excel at supplementing what they can measure with what they know, without either muddying the plot or corrupting the outcome.⁴

Running the job will produce output like the graph at the right (REF). It should also produce a sense of frustration at the run time and output size.

It's all those tiny trivial tiles from Russia, the US, and other countries with a large interior. As our approach stands right now, 200-something countries produced XXX number of output tiles. Joining that to the 2.3 million rows of crop production data produces big data from small, which is just the opposite of what we like to do.

Instead of breaking down

4. Some other recommendations include [Steven Levitt's](#) journal articles and the Freakonomics blog; OKCupid's OKStats (REF); and [Analytics Made Skeezy](#) (REF). On the data visualization front, see Flowing Data (REF) and The Why Axis (REF)

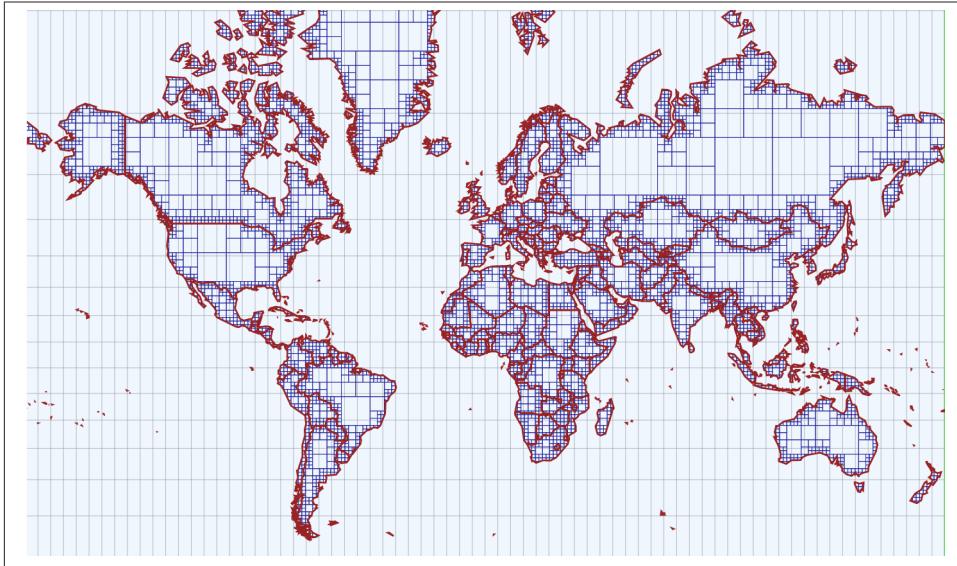


Figure 9-9. Variable Level of Detail prevents Inefficient Tiny Tiles

The QuadDecompose UDF accepts an outer and an inner zoom level. Every region will be broken down to at least the coarser zoom level; you'll use this grid size as the partition key (in map/reduce) or group key (in Pig). No region will be broken into tiles smaller than the inner zoom level, which is important for both managing the data volume and for ensuring we don't group data into finer bins than it can support (see the section on “Choosing a Histogram Bin Size” (REF)).

In the graphic (REF), you can see the result: the top two images, showing the tiles decomposed naively (on the left) and hierarchically (on the right), are identical. The lower row has the same data but with a light border around each tile in the heatmap. The large interior portions of Russia, China, the USA, Brazil and others are now represented by coarser zoom-level blocks. Even France and Argentina manage to line up their borders fortuitously for a zoom-level 6 (TODO check size) block. In all, we reduced the number of tiles from XXX down to XXX without impacting accuracy.

Since every point on the map is covered by at most one region for the current data set, you'll find that every tile is either (a) wholly in the interior of a country, or (b) at the finest zoom level: there's no way for Russia to send data off to a huge zoom-level 5 tile while Latvia sends data to one of its zoom-level 7 children. The zoom-level chosen for each tile in the FOREACH...QuadDecompose pass was the correct granularity for the result.

Quadtree Decomposition and Numbering

Let's look closer at the quadtree scheme.

Our Reindeer friends, with their ample free time and bizarre sense of humor, sometimes vacation at a quaint vacation estate known as Spatial Manor. On the one hand, every once in a while somebody attacks a fellow guest with candlestick or rope or whatever's close at hand. On the other hand, though, most people *aren't* killed and get to enjoy the rousing sport of solving a mystery!

We can apply our spatial analysis tools to its simple geometry and help the investigators make a data-backed decision. Here is a map of the grounds.

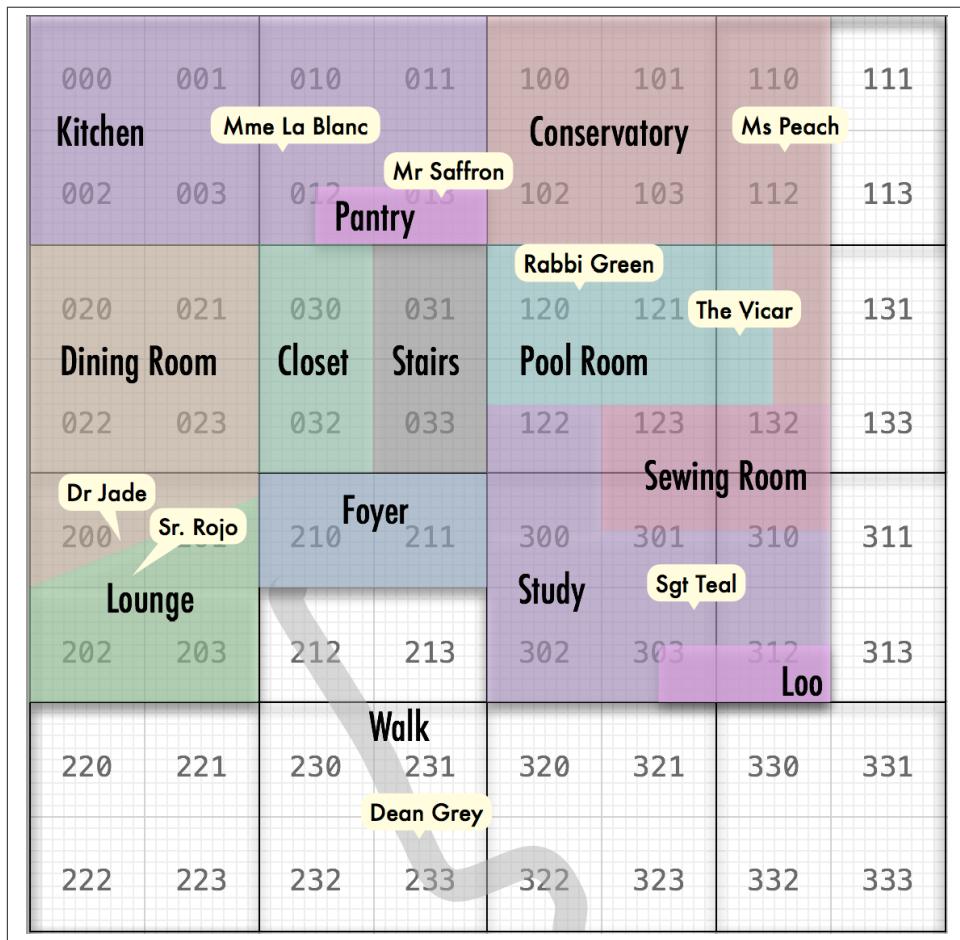


Figure 9-10. Spatial Manor - Rooms (and underlying ZL-3 grid)

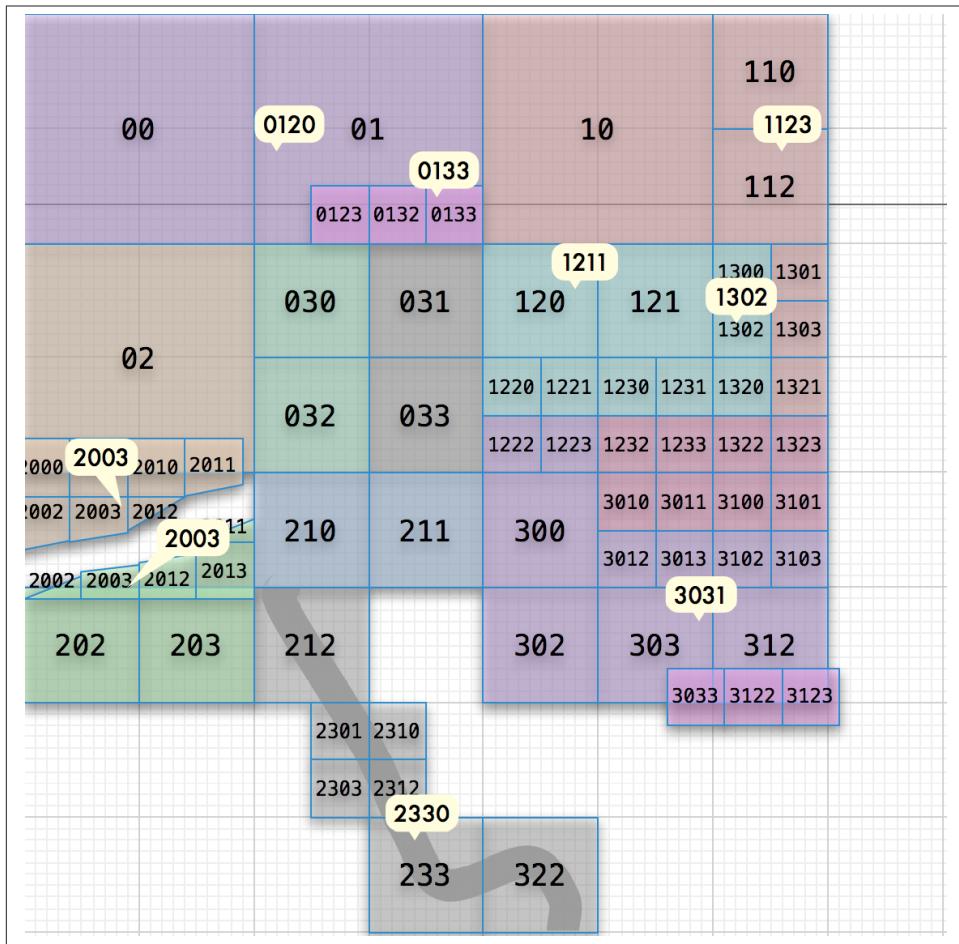


Figure 9-11. Spatial Manor - Decomposed

Exporting data for Presentation by a Tileserver

Features following a constant bearing in any direction — Manhattan's Broadway, or the borderlines of Algeria or Nevada — remain straight lines on the map. This is important for navigational purposes

The locality properties of quadtiles indexing

Typically you will store the shape clipped to the given quad-tile, metadata about that region — population, metric tons of bananas exported annually, an image of its flag, lyrics to its national anthem — is stored under the same key but in independent columns

footnote:[typically the regions are heavyweight, heavily requested and read-only, so they deserve their own table or at least their own column family.

You don't have to break What we do is When tile 0231_1 is requested

```
0230_103 us (California
0230_11X us
0230_2XX pacific
0230_30*
0320_31*
0320_32X
0320_33*
0231_0XX US
0231_1xx US
0231_2** Mexico, SW US, pacific
0231_3** Mexico, SW US, Gulf of Mexico
0231_322 Monterey
0231_323 Tampico
0231_33 Caribbean east of Mexico
0232_3333 4 between Hawaii and Baja calif
```

Querying on the ZL-5 tile 0231_0 (using key 0231_033), any of its four children (using keys 0231_003, 0231_013, 0231_023, or 0231_033) or any of the sixteen ZL-7 descendants 0231_000 through 0231_033 will retrieve a single tile. Querying on its parent 0231 will return tiles for 0231_0 and 0231_1, and a collection of zoom level 6 and 7 tiles covering the southwest US, Mexico and the Gulf of Mexico.

TODO: screenshot.

Our data is stored with no duplication, but

Its easy to decompose or clip a super-tile to a requested (finer) zoom level. The longitude just divides normally along the tile: a hypothetical tile from 16 to 32 would have spatial coordinates 16, 18, 20, The latitude cut points do not subdivide directly, but only need to be calculated for one edge: if tile 0230_00 has bottom edge XX latitude, so does tile 0230_01, 0230_10, 0231_11, and others in its horizontal grid row.

Mechanics of Spatial Data

We kicked off the chapter with two examples that didn't require too many new concepts, but it's time to backtrack a bit and properly cover the mechanics of working with spatial data.

The data types and operations are extremely well standardized by the [Open Geospatial Consortium](#). Nearly all of the operations below have identical behavior within Oracle, PostGIS, SQL Server, and all industrial-strength geospatial systems. In fact, the geo-spatial toolkits for Pig (Pigeon) and Hive (Esri-SFFH) are particularly sympathetic as they both use Esri's wonderful [Esri Geometry API](#) under the hood.

Spatial Data Types

- **Point** — a single location in space, given by its horizontal, then vertical coordinates. That's an easy convention to swallow when you think in terms x, y — but also means you should always list coordinates in the order longitude first then latitude. Get in the habit of always using that ordering.
- **LineString** — a single continuous path, described as an ordered sequence of points. To describe a closed path, repeat the line's start point as its end point. A path is *simple* if it does not cross or touch itself; a path is a *ring* if it is both simple and closed.
- **Polygon** — a connected surface in space, described by at least one closed simple path defining its exterior, and zero one or many non-crossing rings defining any interior holes. The exterior ring is always listed first, and no ring is permitted to cross or touch itself or any other ring.
- **MultiPoint** — a collection of points regarded as a single shape.
- **MultiLineString** — a collection of lines regarded as a single shape. Although a **Polygon** also has multiple chains of coordinates, a **Polygon** is not a **MultiLine String**. Most importantly, a **Polygon** represent a 2-D shape with an interior; a **MultiLineString** represents a collection of 1-D shapes. What's more, the line strings defining a polygon must be non-intersecting rings, while the elements of a **LineString** or **MultiLineString** are permitted to be either open or closed, and may cross or touch.
- **MultiPolygon** — you guessed it, a collection of polygons regarded as a single shape. These polygons are allowed to overlap, lie within each other, or anything else they want to do.
- **Envelope** — an axis-aligned rectangle depicting the minimum and maximum extent of a shape in each coordinate. Since its sides are aligned with the axes, we only have to give the coordinates of two of its corners. From the perspective of the geometry libraries this does not live in the same type hierarchy as the geometry objects above, but it's easy enough to generate the polygon corresponding to an envelope or the envelope of any shape. Any time you're specifying a bounding box, follow the convention of numerically-lowest-coordinates then numerically-highest-coordinates, i.e. $(\min_x, \min_y), (\max_x, \max_y)$. Like the longitude-then-latitude convention, it's violated just often enough to drive you crazy.

Those are the essential data types used by geospatial libraries everywhere. However, when adapting geospatial methods to Pig there are really only two families of shapes to consider:

- Points, which lack spatial extent

- Regions (i.e. all geometries that are not of type `Point`), which span more than one location in space

The first example in this chapter, we were careful to clarify, only covered spatial aggregations of *points*. As soon as we've walked through the core mechanics, we'll demonstrate the same pattern but for spatial aggregations of *regions*, and you'll see the important difference that causes.



Some terminology notes: We'll use the term *geometry* to mean any of our internal data structures: `Point`, `LineString`, `Polygon`, `MultiPoint`, `MultiLineString`, `MultiPolygon`. The term *shape* refers to the geometric shape it describes: one or many points, lines or polygons. Don't read too much into the distinction; mostly, it gives us another word to use because otherwise our sentences are just "blah blah geo geometry geo blah geogeometry blahgeo the geoblah". We'll refer to `Point`, `LineString`, and `Polygon` collectively as singular-types, and `MultiPoint`, `MultiLineString`, `MultiPolygon` as multi-types. And to say "either a point or a multi-point", we'll write *(multi)point* (or similarly *(multi)line*, *(multi)polygon*, *(multi)geometry*) — "the dimension of a (multi)line is one, and the dimension of a (multi)polygon is two". (REVIEWER: which is better, *(multi)point*, or *point/multi-point*?)

Spatial Operations in Pig

Fully explaining and exploring each of the core spatial operations would be a waste of your time and shelf space. For a deep understanding of how they work, you're better off consulting traditional GIS resources — what we're after here is to show you how to use the map/reduce framework to coordinate those core operations. On the other hand, once you know that the operation for "take an object and make a new shape that covers all the points within 100 meters of it" is spelled `GeoBuffer(geom)`, you're 75% of the way there. And reading other resources, we found that the more explanation an author supplied, the larger the amount of topology in the explanation and the less we understood it.

So we're including here what worked for us: a semi-formal description of the operation, pretty pictures demonstrating it, and the list of important specific cases spelled out. If it feels like some bullet points repeat the same description in a different way, that's our intention, as some people latch onto a mathematical statement and others to a physical description. We won't use more than a handful of the operations below in the rest of the book, so you're welcome to just skim this section now (skipping ahead to "Spatial Aggregations on Regions" (REF)) and refer back to it as a reference.

Shape Transforming Operations

We'll start with the operations that transform a shape on its own to produce a new geometry. Some operations return a related but dissimilar object: a shape's bounding box, the point at its "center", and so forth. Others feel more like modifications of their input: reducing the level of detail, finding the

- `GeomEnvelope` — Bounding box for the given shape: an axis-aligned rectangle spanning the shape's minimum and maximum extent in each dimension. Working with rectangles is dramatically simpler and faster than working with arbitrary shape. This is our frontline tool for performing the rough-carpentry work of figuring what might be relevant.
- `GeomCentroid(geom)` — The point at the geometric center of a geometry. The centroid lies at the arithmetic mean of all the input coordinates, weighted by area for (multi)polygon, by line segment length for (multi)lines, or equally for (multi)points. The centroid of a point is itself; the centroid of a single line segment is its midpoint; and the centroid of an empty geometry is empty. This is used in some geometric algorithms, and on occasions where you'd like to informally represent a region as a point at its center — for example, visualizing data about a region as clickable "pushpins" on a map. Be careful, however: the centroid of a polygon is not guaranteed to lie within its interior.
- `GeomPointOnSurface(poly), PointOnSurface(multi_poly)` — An arbitrary point guaranteed to lie on the surface of a given Polygon or MultiPolygon.
- `StartPoint(line), EndPoint(line), PointN(line, idx)` — the first, last or n'th point on the given line. `StartPoint` and `EndPoint` are sugar for `PointN(line, 0)` and `PointN(line, NumPoints(line))` respectively.
- `ExteriorRing(poly), InteriorRingN(poly, idx)` — A LineString giving the polygon's outermost (ExteriorRing) or n-th innermost ring (InteriorRingN), counting inward.
- `GeomBoundary(geom)` — The shape separating the object's interior from its exterior. The boundary will always have one fewer dimensions than its input: a point's boundary is empty; a line's boundary is its start and end point; and a polygon's boundary is a `MultiLineString` of its rings. The boundary of any multi-type geometry is the combined boundaries of its parts.
- `GeomConvexHull(geom)` — The convex polygon (i.e. no PacMan-like indentations) **that minimally covers all points in its input**. Think of this as the shape a rubber band would make if you stretched it around all parts of the input geometry. The ConvexHull is always a single point, line or (usually) polygon — it is never a multi-point/line/polygon.

- `GeomSimplify(geom, tolerance)` — intelligently reduces the number of points in the input by washing out deviations smaller than the given tolerance. For example, an intensely simplified circle will become a triangle. You may call `Simplify` with any geometry type, but it only affects (multi)lines and (multi)polygons — it will not eliminate points from a `MultiPoint` input.
- `GeomBuffer(geom, distance)` — Shape covering the area within a given distance from the input. A `Polygon` for singular inputs, a `MultiPolygon` for multi-inputs. This is useful for doing a “within x distance” spatial join, as you saw above. However, you must be careful to use geodetic (“great circle”) distances if your points are on a sphere. // IMPROVEME: explain a bit better

Constructing and Converting Geometry Objects

Somewhat related are operations that bring shapes in and out of Pig’s control.

- `FromWKTText(chararray), FromGeoJson(chararray)` — converts the serialized description of a shape into the corresponding geometry object. We’ll cover these data formats a bit later in the chapter. Similarly, `ToWKTText(geom)` and `ToGeoJson(geom)` serialize a geometry into a string
- `GeoPoint(x_coord, y_coord)` — constructs a `Point` from the given coordinates
- `GeoEnvelope((x_min, y_min), (x_max, y_max))` — constructs an `Envelope` object from the numerically lowest and numerically highest coordinates. Note that it takes two tuples as inputs, not naked coordinates.
- `GeoMultiToBag(geom)` — splits a (multi)geometry into a bag of simple geometries. A `MultiPoint` becomes a bag of `Points`; a `Point` becomes a bag with a single `Point`, and so forth.
- `GeoBagToMulti(geom)` — combines a bag of geometries into a single multi geometry. For instance, a bag with any mixture of `Point` and `MultiPoint` geometries becomes a single `MultiPoint` object, and similarly for (multi)lines and (multi)polygons. All the elements must have the same dimension — no mixing (multi)points with (multi)lines, etc.

Properties of Shapes

- `GeoArea(geom)`
- `MinX(geom), MinY(geom), MaxX(geom),MaxY(geom)` — the numerically greatest and least extent of a shape in the specified dimension.
- `GeoX(point), GeoY(point)` — X or Y coordinates of a point

- `GeoLength(geom)`
- `GeoLength2dSpheroid(geom)` — Calculates the 2D length of a linestring/multilinestring on an ellipsoid. This is useful if the coordinates of the geometry are in longitude/latitude and a length is desired without reprojection.
- `GeoPerimeter(geom)` — length measurement of a geometry's boundary
- `GeoDistanceSphere(geom)` — Returns minimum distance in meters between two lon/lat geometries. Uses a spherical earth and radius of 6370986 meters. Faster than `GeoDistanceSpheroid`, but less accurate
- `GeoDistance(geom)` — the 2-dimensional cartesian minimum distance (based on spatial ref) between two geometries in projected units.
- `GeoMinDistance(geom)`
- `GeoMaxDistance(geom)` — the 2-dimensional largest distance between two geometries in projected units

There are also a set of meta-operations that report on the geometry objects representing a shape:

- `Dimension(geom)` — This operation returns zero for Point and MultiPoint; 1 for LineString and MultiLineString; and 2 for Polygon and MultiPolygon, regardless of whether those shapes exist in a 2-D or 3-D space
- `CoordDim(geom)` — the number of axes in the coordinate system being used: 2 for X-Y geometries, 3 for X-Y-Z geometries, and so on. Points, lines and polygons within a common coordinate system will all have the same value for `CoordDim`
- `GeometryType(geom)` — string representing the geometry type: 'Point', 'Line String', ..., 'MultiPolygon'.
- `IsEmpty(geom)` — 1 if the geometry contains no actual points.
- `IsLineClosed(line)` — 1 if the given LineString's end point meets its start point.
- `IsSimple` — 1 if the geometry has no anomalous geometric aspects, such as intersecting or being tangent to itself. A multipoint is *simple* if none of its points coincide.
- `IsLineRing` — 1 if the given LineString is a ring — that is, closed and simple.
- `NumGeometries(geom_collection)`
- `NumInteriorRings(poly)`

Operations that Combine Shapes

The power players of our toolkit are operations that combine shapes to produce new ones, most prominently set operations such as intersection or union. These behave

similarly to the set operations on elements in a bag that we explored in chapter (REF), because the underlying mathematics are the same. But whereas the sets in those operations were the elements in two given bags, these operations apply to the *topological point sets* that our geometry objects represent.

- `GeoUnion(geom_a, geom_b)` — geometry representing the merger of the two shapes. A region is within the result if and only if it is within either input.
- `GeoIntersection(geom_a, geom_b)` — geometry representing the intersection of the two shapes. A region is within the result if and only if it is within both inputs.
- `GeoDifference(geom_a, geom_b)` — geometry representing the portion of the first shape excluding the extent of the second shape. A region is within the result if and only if it is within the first input but not the second.
- `GeoSymmDifference(geom_a, geom_b)` — geometry representing the portion of the either shape that is not within the other shape. A region is within the result if and only if it is within one but not both inputs.

Testing the Relationship of Two Shapes

The geospatial toolbox has a set of precisely specified spatial relationships. They each represent a set of constraints on how the boundary, interior, and exterior of one geometry relates to the boundary, interior, and exterior of the other geometry. Our caveat at the top of the chapter about the difficulty of describing these operations correctly without explaining them into incoherence is especially true here. For best results, grab the scripts from the sample code repository (REF) and try various cases.

- `Equals(geom_a, geom_b)` — 1 if the shapes are equal.
- `OrderingEquals(geom_a, geom_b)` — 1 if the shapes are equal *and* their coordinates have the same ordering
- `Intersects(geom_a, geom_b)` — 1 if the shapes intersect: at least one point from the boundary or interior of one shape lies on the boundary or interior of the other.
- `Disjoint(geom_a, geom_b)` — 1 if the shapes do *not* intersect. This operation is sugar for `(GeoIntersects(sa, sb) == 0 ? 1 : 0)`.
- `EnvIntersects(geom_a, geom_b)` — 1 if the bounding envelope of the two shapes intersect.
- ‘`Contains(geom_a, geom_b)` — 1 if `geom_a` completely contains `geom_b`: that is, the shapes’ interiors intersect, and no part of ‘`geom_b`’ lies in the exterior of `geom_a`. If two shapes are equal, then it is true that each contains the other. `Contains(A, B)` is exactly equivalent to `Within(B, A)`.

- ‘Within(geom_a, geom_b) – 1 if `geom_a is completely contained by geom_b: that is, the shapes’ interiors intersect, and no part of geom_a lies in the exterior of geom_b. If two shapes are equal, then it is true that each is within the other.
- Covers(geom_a, geom_b) — 1 if no point in geom_b is outside geom_a. Covered By(geom_a, geom_b) is sugar for Covers(geom_b, geom_a).
- Crosses(geom_a, geom_b) — 1 if the shapes cross: their geometries have some, but not all, interior points in common; and the dimension of the intersection is one less than the higher-dimension of the two shapes. That’s a mouthful, so let’s just look at the cases in turn:
 - A MultiPoint crosses a (multi)line or (multi)polygon as long as at least one of its points lies in the other shape’s interior, and at least one of its points lies in the other shape’s exterior. Points along the border of the polygon(s) or the endpoints of the line(s) don’t matter.
 - A Line/MultiLine crosses a Polygon/MultiPolygon only when part of some line lies within the polygon(s)’ interior and part of some line lies within the polygon(s)’ exterior. Points along the border of a polygon or the endpoints of a line don’t matter.
 - A Line/MultiLine crosses another Line/MultiLine only when the intersection of their interiors consists of one or more points, but no line segments. The endpoints of the lines don’t matter.
 - A Point is never considered to cross any another shape, since you need part of one shape to lie outside the other.
 - A Polygon/MultiPolygon is never considered to cross a Polygon/MultiPolygon, since if their interiors intersect anywhere it is necessarily in a Polygon (and thus not of lower dimension).
- Overlaps(geom_a, geom_b) — 1 if the shapes overlap: their intersection has the same dimension as, but is not equal to, either of the given objects.
- Touches(geom_a, geom_b) — 1 if the shapes touch: their interiors do not intersect but the boundary of one object intersects the interior or boundary of the other.



The Pig and Hive libraries are fairly new — in fact, a large part of the Pig methods described here were contributed by your authors — so don’t be surprised to find functionality that hasn’t been implemented yet. In particular, 3-D and higher geometries are poorly supported; CRS (coordinate reference system) awareness is weak and the catalogue of map projections is small; and many opportunities for optimization remain.

Spatial Join with Regions

How the Merge/Sweep Join Works

The quadord key's properties give us the following promises:

- Its descendants always come after it in sort order
- Once the order comes to a tile that is not one of its descendants, that tile will never again be relevant.

Put another way: all relevant records for a tile come in an unbroken stretch from the first time its key is seen, up to (but not including) the first time a non-descendant key is seen.

While it may seem like we need to keep around a lot of candidate tiles for matching, in fact the only tiles we need to retain

A good way to think of this is to pretend we were doing a depth-first enumeration of each tile and its descendants. (Which we are — we just skip all the keys that don't show up). At every point up down and across the tree, the only relevant rows are for the current tile and its ancestors, and we only need to keep a tile around while we investigate its descendants

(TODO: make a similar diagram showing this point more specifically.)

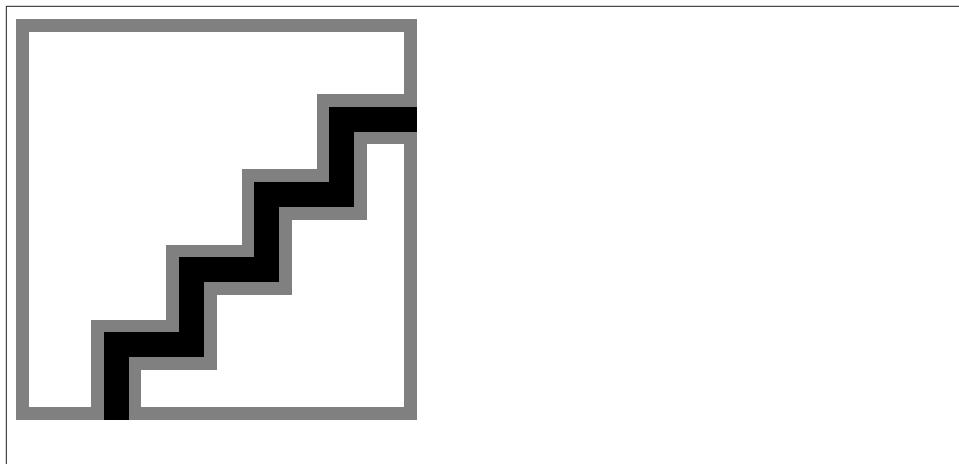


Figure 9-12. Recursive Nesting of Quad Tiles

Sweep Stack

If you paid good attention back in CS312, hearing “depth first search” makes you think “I wonder if I’m about to use a Stack data structure”. If so, give yourself a gold star. We’re not going to actually do a depth-first iteration

- Visit each stack and pop elements until you hit one that is either the same as or an ancestor of our key (or until the stack is empty)
- Push the current element onto its stack
- Iterate over all elements in the other stack, emitting a tuple for each match.

Let’s walk through it. As with the other walkthroughs in this chapter, we’re going to only use four zoom levels and write keys in our “string handle” notation. The string-handle keys have exactly the same sorting properties as our quadord keys do, so the explanation carries over.

quad	slot	place	-- stack 0: tiles for first slot (rooms) --			! -- stack 1: people slot
0	0	West Wing	0	WestWing		!
00	0	Kitchen	0	WestWing	00 Kitchen	!
01	0	Kitchen	0	WestWing	01 Kitchen	!
0120	1	Mme La Rose	0	WestWing	01 Kitchen	! 0120 Mme La Rose
0123	0	Pantry	0	WestWing	01 Kitchen 0123 Pantry	!
0132	0	Pantry	0	WestWing	01 Kitchen 0132 Pantry	!
0133	0	Pantry	0	WestWing	01 Kitchen 0133 Pantry	!
0133	1	Mr Saffron	0	WestWing	01 Kitchen 0133 Pantry	! 0133 Mr Saffron
02	0	Dining	0	WestWing	02 Dining	!
030	0	Closet	0	WestWing	030 Closet	!
0300	1	Red Kelly	0	WestWing	030 Closet	! 0300 Red Kelly
031	0	Stairs	0	WestWing	031 Stairs	!
032	0	Closet	0	WestWing	032 Closet	!
033	0	Stairs	0	WestWing	033 Stairs	!
...						!
10	0	Conservatory	10	Conservatory		!
10	0	East Wing	10	Conservatory	10 EastWing	!
110	0	Conservatory	110	Conservatory		!
110	0	East Wing	110	Conservatory	110 EastWing	!
112	0	Conservatory	112	Conservatory		!
112	0	East Wing	112	Conservatory	112 EastWing	!
1123	1	Ms Peach	112	Conservatory	112 EastWing	! 1123 Ms Peach

The `0 West Wing` covers all of the `0***` blocks, so it comes first in line. There’s nothing to sweep off the stacks, and nothing to pair with, so we just push it onto stack 0. Next is the `00 Kitchen` block, covering `00**`. We keep the `0 WestWing` block (as `0` is a prefix of `00`), push `00 Kitchen` onto stack 0, and since there’s nothing to pair with, continue.

The `01 Kitchen` block is next. It evicts the neighboring `00 Kitchen` block, but not its ancestor `0 WestWing` block. Since there’s still nothing in the people stack, we push `01 Kitchen` and move on. Mme La Rose’s record, the first we’ve seen from the people slot,

now finally gets the party started. Both keys on the stack (0 and 01) are ancestors of 0120, and so nothing is swept and we push Mme La Rose onto the people table. The matching phase generates pairs indicating that Mme La Rose is 01 Kitchen and 0 WestWing at the time of the incident.

The 0120 Pantry tile sweeps its predecessor but produces no matches, as do the next two Pantry tiles. 0133 Mr Saffron finds himself paired with three containing shapes: the WestWing, Kitchen and Pantry. The next step continues with the 02 Dining Room in the lower left of the 0 block. This sweeps out both the Pantry and the Kitchen from the stack, but retains the parent 0 WestWing.

We've supplied a few more of the blocks — trace through them until you're getting the hang of it. Let's skip ahead though. The diagonal south wall separating the Dining room from the Lounge means that the 2002, 2003, 2011, and 2012 blocks contain parts of each room, and it's useful to see where the ambiguity is resolved.

i. | ! 20 0 West Wing | 20 WestWing ! 2000 0 Dining | 20 WestWing 2000 Dining ! 2001 0 Dining | 20 WestWing 2001 Dining ! 2002 0 Dining | 20 WestWing 2002 Dining ! 2002 0 Lounge | 20 WestWing 2002 Dining 2002 Lounge ! 2003 0 Dining | 20 WestWing 2003 Dining ! 2003 0 Lounge | 20 WestWing 2003 Dining 2003 Lounge ! 2003 1 Dr Jade | 20 WestWing 2003 Dining 2003 Lounge ! 2003 Dr Jade 2003 1 Sr Azul | 20 WestWing 2003 Dining 2003 Lounge ! 2003 Dr Jade 2003 Sr Azul 2010 0 Dining | 20 WestWing 2010 Dining !

The 2003 Lounge record comes up with the 20 WestWing and 2003 Dining records already on the stack. The 2003 Dining record has the same tile id, and so according to our rules it is *not* evicted. The 2003 Dr Jade record sweeps nothing from the stack and generates pairs for the WestWing, Dining Room and Lounge. The 2003 Sr Azul record sweeps nothing from either stack (even the Dr Jade record). Its only pairings are with WestWing, Dining Room and Lounge, though — the continued presence of 2003 Dr Jade in the second stack has nothing to do with the matchmaking.

Here is a sample of the pairs that come out of all of this:

0	WestWing	0120	55	30	Mme La Rose
01	Kitchen	0120	55	30	Mme La Rose
0	WestWing	0133	90	40	Mr Saffron
01	Kitchen	0133	90	40	Mr Saffron
0133	Pantry	0133	90	40	Mr Saffron
...					
20	WestWing	2003	20	115	Dr Jade
2003	Dining	2003	20	115	Dr Jade
2003	Lounge	2003	20	115	Dr Jade
20	WestWing	2003	23	122	Sr Azul
2003	Dining	2003	23	122	Sr Azul
2003	Lounge	2003	23	122	Sr Azul

The output of this step only considered tile membership, and so Dr Jade and Sr Azul are each listed in candidate pairings with the Dining Room and the Lounge. That's by design; our next step is to use each room's geometry object to filter out the non-matches.

```
peep_rooms_f = FILTER peep_rooms_g BY GeoIntersects(room.geom, peep.pt);
```

Using only Hadoop's built-in sort and little memory overhead, we were able to assemble records into groups even when they weren't contiguous in the sort order.

An important difference from the conventional COGROUP comes in how we designed the sorting keys and data structures. In a conventional COGROUP, we order the data by the partition key, then the table slot (all records from the left-mentioned input precede those from the last-mentioned input), then any secondary sort keys. That means we don't need a data structure for the last-mentioned input and don't even hold its records in memory — all possible matches for a record from the last input are already sitting hot in RAM ready to make beautiful output tuples. In the spatial COGROUP, we partition on the coarse zoom-level prefix, then sort on the full quadord key before the table slot index. Since the keys must be sorted to support the depth-first-like traversal, it's likely that matching rows from each slot will intermingle. So while the regular COGROUP doesn't have to allocate a data structure for the records in its last-mentioned input, a spatial join of two tables needs to maintain two stacks.

Turning Points of Measurements Into Regions of Influence

Frequently, geospatial data is, for practical reasons, sampled at discrete points but should be understood to represent measurements at all points in space. For example, the measurements in the NCDC datasets are gathered at locations chosen for convenience or value — in some cases, neighboring stations are separated by blocks, in other cases by hundreds of miles. It is useful to be able to reconstruct the underlying spatial distribution from point-sample measurements.

Given a set of locations — broadcast towers, 7-11 stores, hospitals — it is also useful to be able to determine, for any point in space, which of those objects is nearest. When the distribution of objects is even, this is straightforward: choose a bounding box or quad tile you are sure will encompass the point in question and all candidate locations, then choose the nearest candidate. When the distribution is highly uneven, though, the bounding box that works well in rural Montana may return overwhelmingly many results in midtown Manhattan.

We can solve both those problems with a single elegant approach known as Voronoi partitioning. Given a set of seed locations, the Voronoi partitioning returns a set of polygons with the following properties:

- The polygon's 'partition' is the space divided such that every piece of the plane belongs to exactly one polygon.

- There is exactly one polygon for each seed location and all points within it are closer to that seed location than to any other seed location.
- All points on the boundary of two polygons are equidistant from the two neighboring seed locations; and all vertices where Voronoi polygons meet are equidistant from the respective seed locations.

This effectively precomputes the “nearest x” problem: For any point in question, find the unique polygon within which it resides (or rarely, the polygon boundaries upon which it lies). Breaking those polygons up by quad tile at a suitable zoom level makes it easy to either store them in HBase (or equivalent) for fast querying or as data files optimized for a spatial JOIN.

It also presents a solution to the spatial sampling problem by assigning the measurements taken at each sample location to its Voronoi region. You can use these piece-wise regions directly or follow up with some sort of spatial smoothing as your application requires. Let’s dive in and see how to do this in practice.

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

There’s an elegant solution known as a Voronoi diagram⁵.

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

⁵. see [Wikipedia entry](#) or (with a Java-enabled browser) this [Voronoi Diagram applet](#)

from the centerish on either side; polygon corners are equidistant from centerishes of all touching polygons⁶.

The Quadtile Grid System

The quadtile trick is to the world into four and make a Z pattern across them:

Within each of those, make a **Z again**:

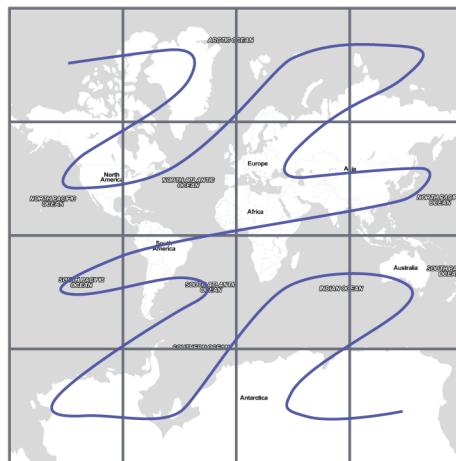


Figure 9-13. Z-path of quadtiles

As you go along, index each tile, as shown in [Figure 9-14](#):

6. 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

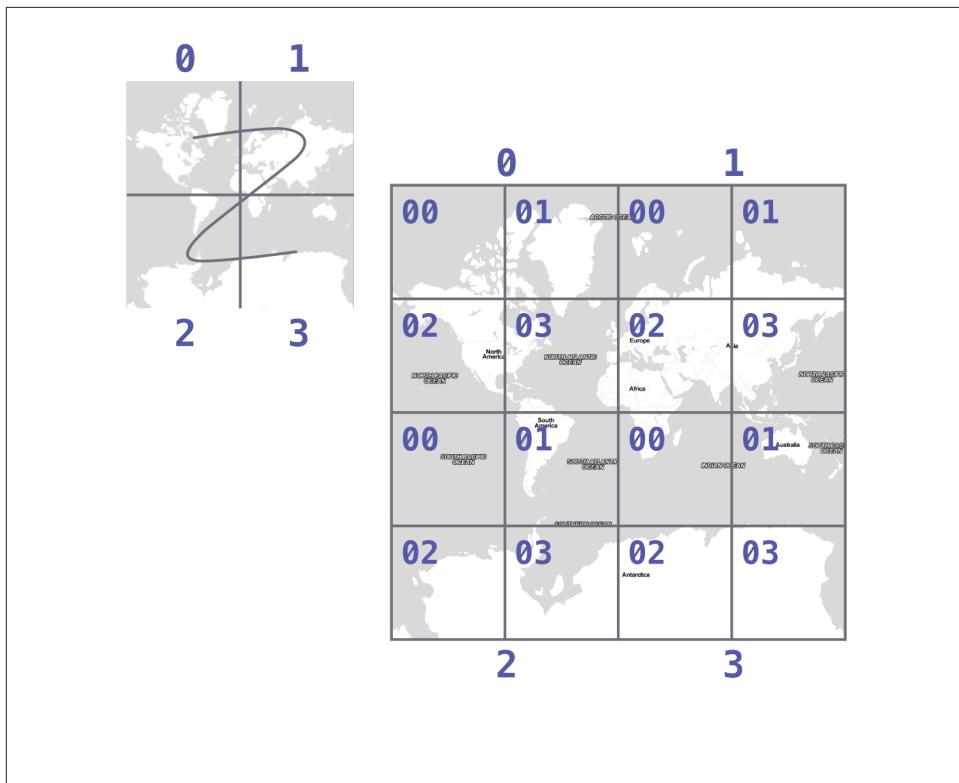


Figure 9-14. Quadtile Numbering

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.

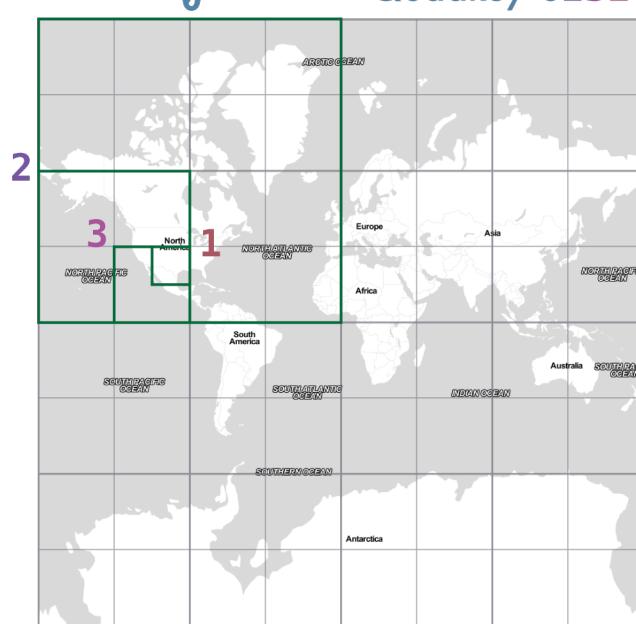
Note: you'll sometimes see people refer to quadtile coordinates as X/Y/Z or Z/X/Y; the Z here refers to zoom level, not a traditional third coordinate.

Quadtile Practicalities

Converting points to quadkeys (quadtile indexes)

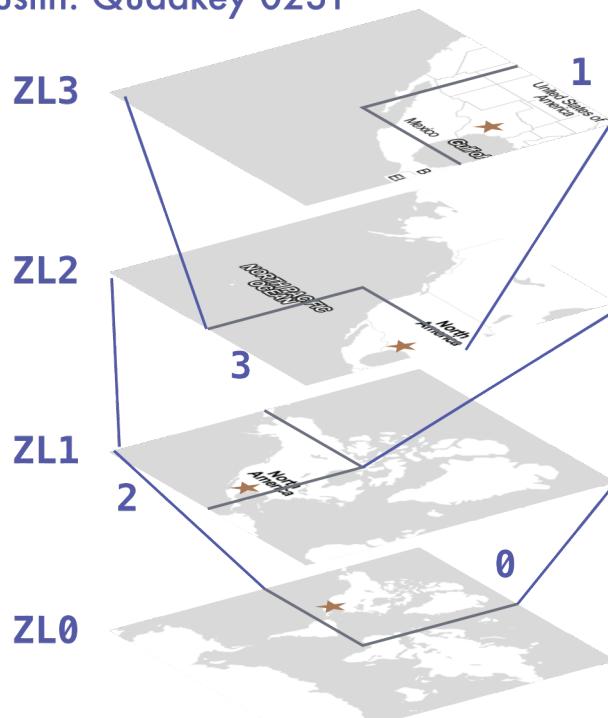
Each grid cell is contained in its parent

Quadkey 0231



You can also think of it as a tree:

Austin: Quadkey 0231



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 level
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. We like `lng`.

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 accuracy 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.

Quadtile Ready Reference

Zoom Level Latitude Lateral Radius	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
ZL 0	1 Rec	66 kB	The World	0,0	30,0	40,0	50,0	85,1	0,0
ZL 1	4 Rec	0 MB		6,378	5,524	4,886	4,100	550	3,963 mi
ZL 2	16 Rec	1 MB		20,038 km	17,353 km	15,350 km	12,880 km	1,729 km	24,902 mi
ZL 3	64 Rec	4 MB	The US (SF-NYC)	10,019 km	8,676 km	7,675 km	6,440 km	864 km	4,769 mi
ZL 4	256 Rec	17 MB	Western Europe (Lisbon-Rome-Berlin-Cork)	5,009 km	4,338 km	3,837 km	3,220 km	432 km	2,384 mi
ZL 5	1 K Rec	67 MB	Honshu (Japan), British Isles (GB+Irel)	2,505 km	2,169 km	1,919 km	1,610 km	216 km	1,192 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 super-periph), 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 km	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, - ¼ 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 21				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	Lateral	66 K Rec	1 M Rec	17 M Rec	1 B Rec	69 B Rec
		% of Equator	Radius	ZL 8	ZL 10	ZL 12	ZL 15	ZL 18
0	Quito, Nairobi, Singapore	100%	6,378	157 km	39 km	10 km	1,223 m	153 m
5	Côte d'Ivoire, Bogotá; S: Kinshasa, Jakarta	100%	6,354	156 km	39 km	10 km	1,218 m	152 m
10	Caracas, Saigon, Addis Ababa	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: Joburg, 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	1,003 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	70 m	88 m
60	S Peterburg, 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	Title X	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
San Francisco	-73.8	37.6	0320103310021220	Title X	0	0	1	3	6	12	24	48	96	192	384	770	1541	3082	6164	12328	24657	49315	98631	197262	394524	789048
Austin	-97.7	30.2	0231031212221213	Title X	0	0	0	1	3	6	12	24	48	99	198	396	792	1585	3170	6341	12483	25364	50733	101467	202035	405870
London	-0.5	51.5	0313131310303102	Title X	0	0	1	3	6	13	26	52	105	210	421	843	1687	3374	6749	13498	26997	52995	102990	215980	431961	
Mumbai (Bombay)	72.9	19.1	123003111212021	Title X	0	1	2	5	10	21	42	84	170	340	601	1362	2723	5450	10900	24081	49403	87204	174409	348846	697691	
Tokyo	139.8	35.6	1330021123302132	Title X	0	1	3	7	14	28	57	114	228	456	913	1826	3653	7306	14613	29226	58453	116907	233815	467430	931425	
Shanghai	121.8	31.1	1321211030213301	Title X	0	1	3	6	13	26	53	107	214	429	858	1716	3432	6858	130735	27470	54941	109883	206684	413368	879071	
Auckland	174.8	-37	3113330003023213	Title X	0	1	3	7	15	31	63	126	252	504	1009	2018	4036	8073	16146	32293	64587	129175	258351	516702	102405	248746
			Title Y	0	1	2	4	9	19	39	78	156	312	625	1250	2501	5003	10007	20014	40029	80058	160117	320234	640468		

Data Formats

Let's take a moment to look at the different file formats used for geographic data. Each has particular tradeoffs that may lead you to make different choices than we have.

GeoJSON

GeoJSON is a new but well-thought-out geodata format, able to represent arbitrary-dimensional geometries in a way that translates nicely to other leading geospatial formats. Its principal advantage is that it is in all respects a JSON file, compatible by any system capable of reading and writing JSON (which is by now most systems). A GeoJSON-aware system will load the `coordinates` field as a shape, but to anything else it's still recognizable as a regular old array. Ironically, the place you're most likely to have a compatibility fail is with traditional GIS systems; due to its young age many GIS systems will lack GeoJSON drivers. However, it's quite easy to convert to and from GeoJSON (see “Converting Among Geospatial Data Formats” below (REF)). The other drawbacks of GeoJSON are those common to any JSON format: it's not particularly space-efficient, and you must parse the whole object before using any part of it. As we'll mention several times, data compression makes the space-efficiency matter less than you think, and compared to disk throughput and the cost of geospatial operations parsing JSON is faster than you think.

If you won't always want to use the geometry data, however, you may also choose to use a TSV format with embedded JSON as we did with WKT/TSV above. Serialize the feature's raw geometry object into a field as GeoJSON, and its properties into individual fields as usual. Since no legal JSON document can contain a raw tab or newline, it is perfectly safe to serialize the geometry into a TSV field. The `FromGeoJson` UDF (note capitalization) in Pig will convert a GeoJSON geometry into a geometry object.

In all, GeoJSON makes an excellent interchange format among different data analysis systems and is a sound choice for development and exploratory analytics.

A GeoJSON `geometry` defines only the shape's type, and coordinates. A GeoJSON `feature` simply contains both a `geometry` object and a `properties` object holding string-key / arbitrary-value pairs according to any scheme you design. Additionally, any GeoJSON object can optionally specify a string identifier (`id`), its bounding box (`bbox`) and the Coordinate Reference System (`crs`) that should be used to interpret its coordinates. Wrap an array of features in a `FeatureCollection` object and you're ready to map! Here is an example GeoJSON feature collection:

```
{  
  "type": "FeatureCollection",  
  "features": [  
    { "type": "Feature",  
      "properties": {"prop0": "value0"},  
      "geometry": {"type": "Point", "coordinates": [102.0, 0.5]}  
    },  
    { "type": "Feature",  
      "properties": {"prop0": "value0"},  
      "geometry": {"type": "LineString", "coordinates": [[10.0, 2.0],[102.0, 0.5]]}  
    },  
    { "type": "Feature",  
      "properties": {"prop0": "value0"},  
      "geometry": {"type": "Polygon", "coordinates": [[[10.0, 2.0], [102.0, 0.5], [102.0, 1.0], [10.0, 2.0]]]}  
    }  
  ]  
}
```

```

    "properties": {
      "prop0": "value0",
      "prop1": {"this": "that"}
    },
    "bbox": [0.0,0.0,8.0,20.0]
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [ [0.0,0.0],[0.0,20.0],[8.0,20.0],[8.0,0.0],[0.0,0.0] ],
      [ [3.0,9.0],[3.0,11.0],[5.0,11.0],[5.0,9.0],[3.0,9.0] ]
    ]
  }
}
]
}
}

```

We pretty-printed that example along multiple lines, but in practice you will want to treat each GeoJSON feature as an independent JSON object, each on its own line. You can restore such a file to the status of GeoJSON feature collection by replacing all new-lines that precede a record with a comma, then stapling `{"type": "FeatureCollection", "features": [` and `]} to the file's front and back.`

GeoJSON geometries encode the full set of primitives we like to use:

- For **Point** geometries, just supply an array in x,y order: `[longitude, latitude]`
- For **LineString** paths, supply an array of points in order. A **LineString** having the same initial and final coordinates will be interpreted as a closed path.
- **Polygon** objects are specified using an array of rings, each of which is an array of points. You must repeat the first point in each ring to make it a closed path. The **Polygon** in the example above describes a rectangle from (0,0) to (8,20), with a 2x2 hole in its center. The first array is the outer ring; other paths in the array are interior rings or holes. For example, South Africa's outer border would be the first ring in the array, followed by the coordinates of the inner ring delimiting Lesotho (an independent country lying completely within South Africa). Regions with multiple parts such as Hawaii or Denmark require a **MultiPolygon** instead.
- The **MultiPoint/MultiLineString/MultiPolygon** types expect an array of the coordinates as appropriate for the singular type. It's fine to supply a **Multi** type with only one element, but you must have it enclosed in an array: `{"type": "Point", "coordinates": [102.0, 0.5]}` and `{"type": "MultiPoint", "coordinates": [[102.0, 0.5]]}` should behave equivalently.
- For ease of processing, you can attach a Bounding Box (**bbox**) annotation to any GeoJSON object. Supply the coordinates in `[left, bottom, right, top]` order — that is, `[xmin, ymin, xmax, ymax]`. A **bbox** is *not* an independent geometry: it is an annotation on another geometry.

The **GeoJSON** standard is as readable a specification as you'll see, so refer to it for anything deeper than we cover here.

Well-Known Text + Tab-Separated Values

At this point in the book you've long since either quit reading in disgust, or you've gotten used to the idea that until performance concerns demonstrate otherwise, in most cases the best data format is the silly-seeming TSV (tab-separated values) scheme. To restate its tradeoffs, a TSV file is easily inspectable, travels anywhere, and can be manipulated from the commandline as plain-text. It's restartable (the damage from a corrupt record lasts only until the next newline), doesn't require special quoting or escaping, and is trivial to parse. Representing numbers in decimal gives mediocre space-efficiency, but keep in mind that a well-configured hadoop cluster (REF) compresses most data as it hits the disk, and so the overhead is nowhere near as large as it might appear. By now you're quite comfortable working around the lack of complex types and need for an explicit schema.

Given that, we'll continue to tax your credulity and advise that until performance concerns demonstrate otherwise, the primitive but oh-so-simple Well-Known Text format is the right choice for geospatial data.

WKT encodes our familiar primitives — Point, LineString, Polygon and MultiPoint, MultiLineString, MultiPolygon. (There are additional geometries for specifying circles, triangle surfaces, meshes and parameterized curves, but we won't get into those.) A WKT object is given by simply stating the geometry type followed by its comma-separated coordinates within parentheses. Whitespace is ignored and any other content is disallowed. Here's an example (the newlines and spacing around the braces is purely ornamental):

```
POINT (102.0 0.5)
LINESTRING (10.0 2.0, 102.0 0.5)
POLYGON (
    (0.0 0.0, 0.0 20.0, 8.0 20.0, 8.0 0.0, 0.0 0.0),
    (3.0 9.0, 3.0 11.0, 5.0 11.0, 5.0 9.0, 3.0 9.0) )
MULTIPOLYGON ((
    (0.0 0.0, 0.0 20.0, 8.0 20.0, 8.0 0.0, 0.0 0.0),
    (3.0 9.0, 3.0 11.0, 5.0 11.0, 5.0 9.0, 3.0 9.0) )
    MULTIPOLYGON (
        ((0.0 0.0, 0.0 20.0, 8.0 20.0, 8.0 0.0, 0.0 0.0)),
        ((3.0 9.0, 3.0 11.0, 5.0 11.0, 5.0 9.0, 3.0 9.0)) )
```

Coordinate pairs are given in longitude/latitude (x/y) order — hopefully you've begun to internalize that convention — with spaces in between. Each string of points — a line-string path or a ring in a polygon — is comma-delimited and wrapped in its own set of parenthesis

All the objections to TSV weigh in against WKT as well — it is unsophisticated, not terribly space efficient, and seems clunky at first use. But all the advantages carry over too — it's commandline friendly, travels anywhere, can be manipulated even in the absence of a parser. For development use, we generally like to work with TSV files holding shapes as embedded WKT fields.

Well-Known Binary (WKB)

WKT is easily translated into *Well-Known Binary (WKB)* format, a straightforward [binary encoding of Well-Known Text \(WKT\)](#). You'll give up the direct access and commandline friendliness, but WKB is nearly as widely understood as WKT and has exactly the same capabilities. Since WKB is more space-efficient and somewhat faster to decode, you may wish to move to it for production work or high-scale applications.

Other Important Formats

The preceding sections describe all the file formats we've found worthwhile for use within Hadoop, but Wikipedia lists several dozen other [geospatial file formats](#). It's worth calling out a few others you'll encounter.

The [*Shapefile*](#) (aka *Esri Shapefile* or *Arcview Shapefile*) format is a complex and powerful geospatial vector data format, ubiquitous in the traditional GIS world. Like GeoJSON, it can represent both shapes and metadata, shares the same (Multi)Point/(Multi)Line/(Multi)Polygon primitives, and can handle two- or three-dimensional data⁷. Unlike GeoJSON, it's not a useful interchange format — although every GIS system will have Shapefile facilities, few applications outside of that realm will. Don't go near the specification — it's incredibly complex, only mostly-specified, and there are excellent open-source libraries for working with shapefiles that will give far better results than anything you should attempt. A shapefile is actually a collection of multiple files: a .shp, a .shx and a .dbf file and potentially others as well. Each collection is intended to represent a single layer of data and so can only contain a single geometry type: you cannot combine airports (point), flight paths (lines) and air traffic control zones (polygons) in the same file. Those limitations — multiple files and homogenous layers — make it a poor choice for representing data on your cluster.

TopoJSON is a [companion format to GeoJSON](#), specifically optimized for data visualization in the browser. (The “Map Viewer for Chimps” tool that we've distributed uses it for reference data). While the other data formats described here represent regions independently — closed squarish polygons for Colorado, Utah, Arizona, etc.. — TopoJSON instead maintains the mesh of edges that define those polygons, along with metadata to recover the original regions. Those duplicate paths cause excess storage size

7. but not higher, as opposed to the arbitrary dimensions available to GeoJSON

and redundant data processing; in the worst case, numerical error can cause borders that should be coincident to stray, leading to visual artifacts and incorrect results. TopoJSON's pre-constructed mesh avoids those problems and makes many tasks possible or simpler, such as cartograms (independent rescaling of each shape based on an attribute) or geometry simplification (eliminating fine-grained detail for rendering). At present, its principal adoption is limited to the d3 Javascript library. Everything we see indicates that d3 is emerging as the best toolkit for lightweight data visualization primitives, and we expect increasing adoption of its byproducts. Having just described how the mesh representation is great for rendering purposes, it's exactly wrong for our use. We want to be able to peel shapes apart and send them to the correct context group. We want to store a shape and its associated data in the same record on disk (as GeoJSON and TSV+WKT do), both to accommodate file splitting and to enable processing as plain data. The github.com/mbostock/topojson TopoJSON project has tools for converting TopoJSON to and from GeoJSON, ESRI Shapefiles, and a few other formats.

Keyhole Markup Language is the [XML-based format used by Google Earth](#). Keyhole, a company acquired by Google, built both the core of Google's online geographic offering and an internet community of enthusiasts who curate geolocation and 3-D models of earth features. The signal-to-noise ratio is often low (and Google occasionally gaslights file locations), but with patience you can find some fairly remarkable data sets under open licenses through Google Earth or the surviving Keyhole community. KML files are distributed with either a .kml extension (plaintext XML) or with a .kmz extension (a ZIP 2.0-compressed bundle containing that .kml file). You should not build your data flows around KML. It's first of all a bad choice for high-scale data analysis in all the ways that any XML-based format is a bad choice — see our "Airing of Grievances" in chapter (REF). KML is even less compact than GeoJSON and lacks the ability to specify a coordinate reference system. Though some traditional GIS applications will import KML, they're just as likely to accept GeoJSON; outside of the GIS world, generic JSON is far superior to generic XML. Seek out and import, yes, but otherwise avoid working with KML.

Open Street Maps (OSM) is one of the crown jewels of the open data movement: a [massive database of places, roads and spatial data](#), community-generated and available under an open license. Together with [GeoCommons](#) and [Natural Earth](#), anyone with an internet connection can freely access geospatial data sets that used to cost millions of dollars if available at all. OSM distributes their data in [a variety of formats](#) documented on their wiki, none of them useful for data analysis at any scale. See the instructions given by [Michal Migurski](#) or [Jacob Perkins](#) for how to convert the data directly in Hadoop.

Converting Among Geospatial Data Formats

As always, our advice is to pick one data format to work with and tolerate no others. Convert all foreign data formats immediately upon receipt, and produce exports (where

necessary) in a separate and final step. As we advised above with WKB (REF), you *may* judiciously choose two formats: one for efficiency and one for interchange with other applications. But do yourself a favor and prove that the interchange format actually costs you enough money to deal with the hassle.

The open-source (MIT License) **GDAL Library** is a superb tool for converting among all the data formats you'll encounter in practice. It handles not only the vector formats we've focused on in this chapter but also raster data, such as satellite imagery from the National Geodetic Survey, .PNG files from Google Maps and other tileservers, and so on. Following closely Mike Bostock's [bost.ocks.org/mike/map/\[Let's make a Map!\]](http://bost.ocks.org/mike/map/[Let's make a Map!]) presentation, here's a brief demonstration of using GDAL to translate an ESRI shapefile to both GeoJSON (for use in Hadoop) and TopoJSON (for efficient rendering in D3).

Install GDAL using your system's package manager (for Mac OSX users running homebrew it's `brew install gdal`) or download binaries from the **GDAL website**. If running `ogr2ogr --help` from the commandline dumps a bewildering soup of options to the screen you've probably installed it correctly.

```
# Go to where your data will live
datadir=~/data
mkdir -p $datadir/{ripd,rawd,out}
mkdir -p $datadir/rawd/natural_earth/shp
#
file=ne_10m_admin_1_states_provinces

# yeah, the link has the http part repeated...
wget -nc http://www.naturalearthdata.com/http://www.naturalearthdata.com/download/10m/cultural/$file.zip
# extract the zip file
(cd $datadir/rawd/natural_earth/shp ; unzip $datadir/ripd/$file.zip)
#
ogr2ogr -f GeoJSON \
$datadir/rawd/natural_earth/great_britain_subunits.json \
$datadir/rawd/natural_earth/shp/$file.shp
```

It's not immediately apparent how to export a TSV file containing WKT (Well-Known Text); you need to use the CSV driver with options as shown:

```
ogr2ogr -f CSV \
-lco GEOMETRY=AS_WKT -lco SEPARATOR=TAB -lco CREATE_CSVT=YES \
/tmp/great_britain_subunits \
$datadir/rawd/natural_earth/shp/$file.shp
mv /tmp/great_britain_subunits/$file.csv \
$datadir/rawd/natural_earth/great_britain_subunits.wkt.tsv
```

Although the output file will be in a subdirectory with a `.csv` extension, it is nonetheless a tab-separated file. The above code block exports it to a temporary location and then renames it.

Incidentally, `ogr2ogr` also offers a simple set of predicates for extracting only selected layers. The following command chooses only the states within Ireland and Great Britain:

```
ogr2ogr -f CSV \  
    -lco GEOMETRY=AS_WKT -lco SEPARATOR=TAB -lco CREATE_CSVT=YES \  
    -where "ADM0_A3 IN ('GBR', 'IRL')" \  
    $datadir/rawd/natural_earth/great_britain_subunits.wkt.tsv \  
    $datadir/rawd/natural_earth/shp/$file.shp
```

Whenever you meet a new data set listing data for the United Kingdom, check whether “admin-1” covers state-level units (Northumberland, Liverpool, etc) and not the intermediate subdivisions of Great Britain, Wales, Scotland, etc. (The same caution applies to Greece, Canada and a few other countries.) We’re safe here, because the Natural Earth dataset maintains separate fields for that information.