PART V
Case Studies

CHAPTER 22

Composable Data at Cerner

Ryan Brush
Micah Whitacre

Healthcare information technology is often a story of automating existing processes. This is changing. Demands to improve care quality and control its costs are growing, creating a need for better systems to support those goals. Here we look at how Cerner is using the Hadoop ecosystem to make sense of healthcare and—building on that knowledge—to help solve such problems.

From CPUs to Semantic Integration

Cerner has long been focused on applying technology to healthcare, with much of our history emphasizing electronic medical records. However, new problems required a broader approach, which led us to look into Hadoop.

In 2009, we needed to create better search indexes of medical records. This led to processing needs not easily solved with other architectures. The search indexes required expensive processing of clinical documentation: extracting terms from the documentation and resolving their relationships with other terms. For instance, if a user typed "heart disease," we wanted documents discussing a myocardial infarction to be returned. This processing was quite expensive—it can take several seconds of CPU time for larger documents—and we wanted to apply it to many millions of documents. In short, we needed to throw a lot of CPUs at the problem, and be cost effective in the process.

Among other options, we considered a staged event-driven architecture (SEDA) approach to ingest documents at scale. But Hadoop stood out for one important need: we wanted to reprocess the many millions of documents frequently, in a small number of hours or faster. The logic for knowledge extraction from the clinical documents was rapidly improving, and we needed to roll improvements out to the world quickly. In Hadoop, this simply meant running a new version of a MapReduce job over data already in place. The process documents were then loaded into a cluster of Apache Solr servers to support application queries.

These early successes set the stage for more involved projects. This type of system and its data can be used as an empirical basis to help control costs and improve care across entire populations. And since healthcare data is often fragmented across systems and institutions, we needed to first bring in all of that data and make sense of it.

With dozens of data sources and formats, and even standardized data models subject to interpretation, we were facing an enormous semantic integration problem. Our biggest challenge was not the *size* of the data—we knew Hadoop could scale to our needs—but the sheer complexity of cleaning, managing, and transforming it for our needs. We needed higher-level tools to manage that complexity.

Enter Apache Crunch

Bringing together and analyzing such disparate datasets creates a lot of demands, but a few stood out:

- We needed to split many processing steps into modules that could easily be assembled into a sophisticated pipeline.
- We needed to offer a higher-level programming model than raw MapReduce.
- We needed to work with the complex structure of medical records, which have several hundred unique fields and several levels of nested substructures.

We explored a variety of options in this case, including Pig, Hive, and Cascading. Each of these worked well, and we continue to use Hive for ad hoc analysis, but they were unwieldy when applying arbitrary logic to our complex data structures. Then we heard of Crunch (see Chapter 18), a project led by Josh Wills that is similar to the FlumeJava system from Google. Crunch offers a simple Java-based programming model and static type checking of records—a perfect fit for our community of Java developers and the type of data we were working with.

Building a Complete Picture

Understanding and managing healthcare at scale requires significant amounts of clean, normalized, and relatable data. Unfortunately, such data is typically spread across a number of sources, making it difficult and error prone to consolidate. Hospitals, doctors' offices, clinics, and pharmacies each hold portions of a person's records in industry-standard formats such as CCDs (Continuity of Care Documents), HL7 (Health Level 7, a healthcare data interchange format), CSV files, or proprietary formats.

Our challenge is to take this data; transform it into a clean, integrated representation; and use it to create registries that help patients manage specific conditions, measure operational aspects of healthcare, and support a variety of analytics, as shown in Figure 22-1.

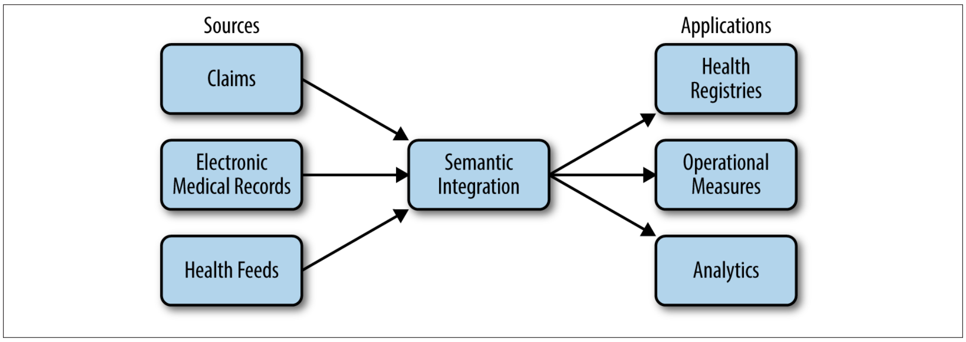

Figure 22-1. Operational data flow

An essential step is to create a clean, semantically integrated basis we can build on, which is the focus of this case study. We start by normalizing data to a common structure. Earlier versions of this system used different models, but have since migrated to Avro for storing and sharing data between processing steps. Example 22-1 shows a simplified Avro IDL to illustrate how our common structures look.

Example 22-1. Avro IDL for common data types

```
@namespace("com.cerner.example")
protocol PersonProtocol {

  record Demographics {
    string firstName;
    string lastName;
    string dob;
    ...
  }

  record LabResult {
    string personId;
    string labDate;
    int labId;
    int labTypeId;
    int value;
  }

  record Medication {
    string personId;
    string medicationId;
    string dose;
    string doseUnits;
    string frequency;
    ...
  }

  record Diagnosis {
    string personId;
    string diagnosisId;
    string date;
    ...
  }

  record Allergy {
    string personId;
    int allergyId;
    int substanceId;
    ...
  }

  /**
   * Represents a person's record from a single source.
   */
  record PersonRecord {
    string personId;
    Demographics demographics;
    array<LabResult> labResults;
    array<Allergy> allergies;
    array<Medication> medications;
    array<Diagnosis> diagnoses;
    . . .
  }
}
```

Note that a variety of data types are all nested in a common person record rather than in separate datasets. This supports the most common usage pattern for this data—looking at a complete record—without requiring downstream operations to do a number of expensive joins between datasets.

A series of Crunch pipelines are used to manipulate the data into a `PCollection<PersonRecord>` hiding the complexity of each source and providing a simple interface to interact with the raw, normalized record data. Behind the scenes, each `PersonRecord` can be stored in HDFS or as a row in HBase with the individual data elements spread throughout column families and qualifiers. The result of the aggregation looks like the data in Table 22-1.

Table 22-1. Aggregated data

Source	Person ID	Person demographics	Data
Doctor's office	12345	Abraham Lincoln ...	Diabetes diagnosis, lab results
Hospital	98765	Abe Lincoln ...	Flu diagnosis
Pharmacy	98765	Abe Lincoln ...	Allergies, medications
Clinic	76543	A. Lincoln ...	Lab results

Consumers wishing to retrieve data from a collection of authorized sources call a "retriever" API that simply produces a Crunch `PCollection` of requested data:

```
Set<String> sources = ...;
PCollection<PersonRecord> personRecords =
    RecordRetriever.getData(pipeline, sources);
```

This retriever pattern allows consumers to load datasets while being insulated from how and where they are physically stored. At the time of this writing, some use of this pattern is being replaced by the emerging Kite SDK for managing data in Hadoop. Each entry in the retrieved `PCollection<PersonRecord>` represents a person's complete medical record within the context of a single source.

Integrating Healthcare Data

There are dozens of processing steps between raw data and answers to healthcare-related questions. Here we look at one: bringing together data for a single person from multiple sources.

Unfortunately, the lack of a common patient identifier in the United States, combined with noisy data such as variations in a person's name and demographics between systems, makes it difficult to accurately unify a person's data across sources. Information spread across multiple sources might look like Table 22-2.

Table 22-2. Data from multiple sources

Source	Person ID	First name	Last name	Address	Gender
Doctor's office	12345	Abraham	Lincoln	1600 Pennsylvania Ave.	M
Hospital	98765	Abe	Lincoln	Washington, DC	M
Hospital	45678	Mary Todd	Lincoln	1600 Pennsylvania Ave.	F
Clinic	76543	A.	Lincoln	Springfield, IL	M

This is typically resolved in healthcare by a system called an *Enterprise Master Patient Index* (EMPI). An EMPI can be fed data from multiple systems and determine which records are indeed for the same person. This is achieved in a variety of ways, ranging from humans explicitly stating relationships to sophisticated algorithms that identify commonality.

In some cases, we can load EMPI information from external systems, and in others we compute it within Hadoop. The key is that we can expose this information for use in our Crunch-based pipelines. The result is a `PCollection<EMPIRecord>` with the data structured as follows:

```
@namespace("com.cerner.example")
protocol EMPIProtocol {

  record PersonRecordId {
    string sourceId;
    string personId
  }

  /**
   * Represents an EMPI match.
   */
  record EMPIRecord {
    string empiId;
    array<PersonRecordId> personIds;
  }
}
```

Given EMPI information for the data in this structure, `PCollection<EMPIRecord>` would contain data like that shown in Table 22-3.

Table 22-3. EMPI data

EMPI identifier	PersonRecordIds (<SourceId, PersonId>)
EMPI-1	<offc-135, 12345> <hspt-246, 98765> <clnc-791, 76543>
EMPI-2	<hspt-802, 45678>

In order to group a person's medical records in a single location based upon the provided `PCollection<EMPIRecord>` and `PCollection<PersonRecord>`, the collections must be converted into a `PTable`, keyed by a common key. In this situation, a `Pair<String, String>`, where the first value is the `sourceId` and the second is the `personId`, will guarantee a unique key to use for joining.

The first step is to extract the common key from each `EMPIRecord` in the collection:

```
PCollection<EMPIRecord> empiRecords = ...;
PTable<Pair<String, String>, EMPIRecord> keyedEmpiRecords =
    empiRecords.parallelDo(
  new DoFn<EMPIRecord, Pair<Pair<String, String>, EMPIRecord>>() {
    @Override
    public void process(EMPIRecord input,
        Emitter<Pair<Pair<String, String>, EMPIRecord>> emitter) {
      for (PersonRecordId recordId: input.getPersonIds()) {
        emitter.emit(Pair.of(
            Pair.of(recordId.getSourceId(), recordId.getPersonId()), input));
      }
    }
  }, tableOf(pairs(strings(), strings()), records(EMPIRecord.class)
);
```

Next, the same key needs to be extracted from each `PersonRecord`:

```
PCollection<PersonRecord> personRecords = ...;
PTable<Pair<String, String>, PersonRecord> keyedPersonRecords = personRecords.by(
    new MapFn<PersonRecord, Pair<String, String>>() {
  @Override
  public Pair<String, String> map(PersonRecord input) {
    return Pair.of(input.getSourceId(), input.getPersonId());
  }
}, pairs(strings(), strings()));
```

Joining the two `PTable` objects will return a `PTable<Pair<String, String>, Pair<EMPIRecord, PersonRecord>>`. In this situation, the keys are no longer useful, so we change the table to be keyed by the EMPI identifier:

```
PTable<String, PersonRecord> personRecordKeyedByEMPI = keyedPersonRecords
    .join(keyedEmpiRecords)
    .values()
    .by(new MapFn<Pair<PersonRecord, EMPIRecord>>() {
  @Override
  public String map(Pair<PersonRecord, EMPIRecord> input) {
    return input.second().getEmpiId();
  }
}, strings()));
```

The final step is to group the table by its key to ensure all of the data is aggregated together for processing as a complete collection:

```
PGroupedTable<String, PersonRecord> groupedPersonRecords =
    personRecordKeyedByEMPI.groupByKey();
```

The `PGroupedTable` would contain data like that in Table 22-4.

This logic to unify data sources is the first step of a larger execution flow. Other Crunch functions downstream build on these steps to meet many client needs. In a common use case, a number of problems are solved by loading the contents of the unified `PersonRecords` into a rules-based processing model to emit new clinical knowledge. For instance, we may run rules over those records to determine if a diabetic is receiving recommended care, and to indicate areas that can be improved. Similar rule sets exist for a variety of needs, ranging from general wellness to managing complicated conditions. The logic can be complicated and with a lot of variance between use cases, but it is all hosted in functions composed in a Crunch pipeline.

Table 22-4. Grouped EMPI data

EMPI identifier	Iterable<PersonRecord>
`EMPI-1`	`{ "personId": "12345", "demographics": { "firstName": "Abraham", "lastName": "Lincoln", ... }, "labResults": [...] }, { "personId": "98765", "demographics": { "firstName": "Abe", "lastName": "Lincoln", ... }, "diagnoses": [...] }, { "personId": "98765", "demographics": { "firstName": "Abe", "lastName": "Lincoln", ... }, "medications": [...]}, { "personId": "76543", "demographics": { "firstName": "A.", "lastName": "Lincoln", ... } ... }`
`EMPI-2`	`{ "personId": "45678", "demographics": { "firstName": "Mary Todd", "lastName": "Lincoln", ... } ... }`

Composability over Frameworks

The patterns described here take on a particular class of problem in healthcare centered around the person. However, this data can serve as the basis for understanding operational and systemic properties of healthcare as well, creating new demands on our ability to transform and analyze it.

Libraries like Crunch help us meet emerging demands because they help make our data and processing logic composable. Rather than a single, static framework for data processing, we can modularize functions and datasets and reuse them as new needs emerge. Figure 22-2 shows how components can be wired into one another in novel ways, with each box implemented as one or more Crunch `DoFns`. Here we leverage person records to identify diabetics and recommend health management programs, while using those composable pieces to integrate operational data and drive analytics of the health system.

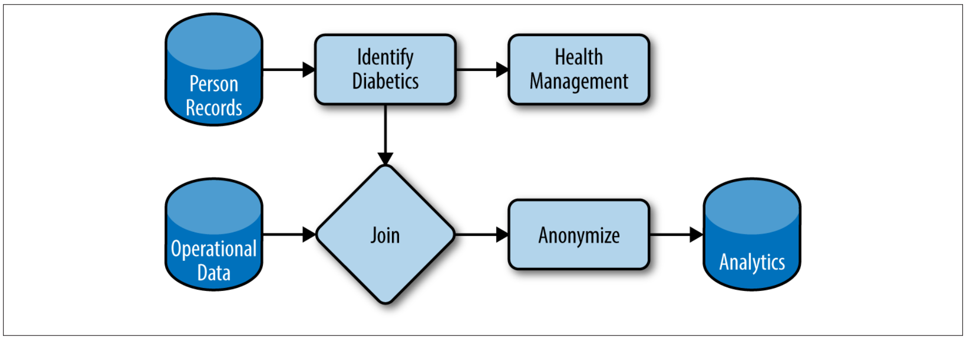

Figure 22-2. Composable datasets and functions

Composability also makes iterating through new problem spaces easier. When creating a new view of data to answer a new class of question, we can tap into existing datasets and transformations and emit our new version. As the problem becomes better understood, that view can be replaced or updated iteratively. Ultimately, these new functions and datasets can be contributed back and leveraged for new needs. The result is a growing catalog of datasets to support growing demands to understand the data.

Processing is orchestrated with Oozie. Every time new data arrives, a new dataset is created with a unique identifier in a well-defined location in HDFS. Oozie coordinators watch that location and simply launch Crunch jobs to create downstream datasets, which may subsequently be picked up by other coordinators. At the time of this writing, datasets and updates are identified by UUIDs to keep them unique. However, we are in the process of placing new data in timestamp-based partitions in order to better work with Oozie's nominal time model.

Moving Forward

We are looking to two major steps to maximize the value from this system more efficiently.

First, we want to create prescriptive practices around the Hadoop ecosystem and its supporting libraries. A number of good practices are defined in this book and elsewhere, but they often require significant expertise to implement effectively. We are using and building libraries that make such patterns explicit and accessible to a larger audience.

Crunch offers some good examples of this, with a variety of join and processing patterns built into the library.

Second, our growing catalog of datasets has created a demand for simple and prescriptive data management to complement the processing features offered by Crunch. We have been adopting the Kite SDK to meet this need in some use cases, and expect to expand its use over time.

The end goal is a secure, scalable catalog of data to support many needs in healthcare, including problems that have not yet emerged. Hadoop has shown it can scale to our data and processing needs, and higher-level libraries are now making it usable by a larger audience for many problems.

CHAPTER 23

Biological Data Science: Saving Lives with Software

Matt Massie

It's hard to believe a decade has passed since the MapReduce paper appeared at OSDI'04. It's also hard to overstate the impact that paper had on the tech industry; the MapReduce paradigm opened distributed programming to nonexperts and enabled large-scale data processing on clusters built using commodity hardware. The open source community responded by creating open source MapReduce-based systems, like Apache Hadoop and Spark, that enabled data scientists and engineers to formulate and solve problems at a scale unimagined before.

While the tech industry was being transformed by MapReduce-based systems, biology was experiencing its own metamorphosis driven by second-generation (or "next-generation") sequencing technology; see Figure 23-1. Sequencing machines are scientific instruments that read the chemical "letters" (A, C, T, and G) that make up your genome: your complete set of genetic material. To have your genome sequenced when the MapReduce paper was published cost about $20 million and took many months to complete; today, it costs just a few thousand dollars and takes only a few days. While the first human genome took decades to create, in 2014 alone an estimated 228,000 genomes were sequenced worldwide.[1] This estimate implies around 20 petabytes (PB) of sequencing data were generated in 2014 worldwide.

1. See Antonio Regalado, "EmTech: Illumina Says 228,000 Human Genomes Will Be Sequenced This Year," September 24, 2014.

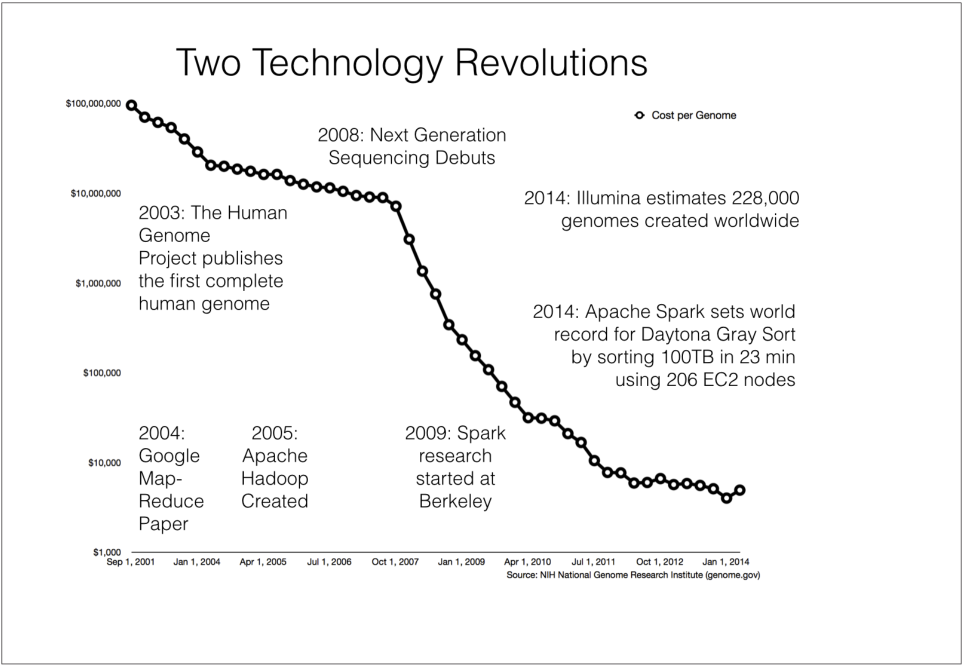

Figure 23-1. Timeline of big data technology and cost of sequencing a genome

The plummeting cost of sequencing points to superlinear growth of genomics data over the coming years. This DNA data deluge has left biological data scientists struggling to process data in a timely and scalable way using current genomics software. The AMPLab is a research lab in the Computer Science Division at UC Berkeley focused on creating novel big data systems and applications. For example, Apache Spark (see Chapter 19) is one system that grew out of the AMPLab. Spark recently broke the world record for the Daytona Gray Sort, sorting 100 TB in just 23 minutes. The team at Databricks that broke the record also demonstrated they could sort 1 PB in less than 4 hours!

Consider this amazing possibility: we have technology today that could analyze every genome collected in 2014 on the order of days using a few hundred machines.

While the AMPLab identified genomics as the ideal big data application for technical reasons, there are also more important compassionate reasons: the timely processing of biological data saves lives. This short use case will focus on systems we use and have developed, with our partners and the open source community, to quickly analyze large biological datasets.

The Structure of DNA

The discovery in 1953 by Francis Crick and James D. Watson, using experimental data collected by Rosalind Franklin and Maurice Wilkins, that DNA has a double helix structure was one of the greatest scientific discoveries of the 20th century. Their *Nature* article entitled "Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid" contains one of the most profound and understated sentences in science:

> It has not escaped our notice that the specific pairing we have postulated immediately suggests a possible copying mechanism for the genetic material.

This "specific pairing" referred to the observation that the bases adenine (A) and thymine (T) always pair together and guanine (G) and cytosine (C) always pair together; see Figure 23-2. This deterministic pairing enables a "copying mechanism": the DNA double helix unwinds and complementary base pairs snap into place, creating two exact copies of the original DNA strand.

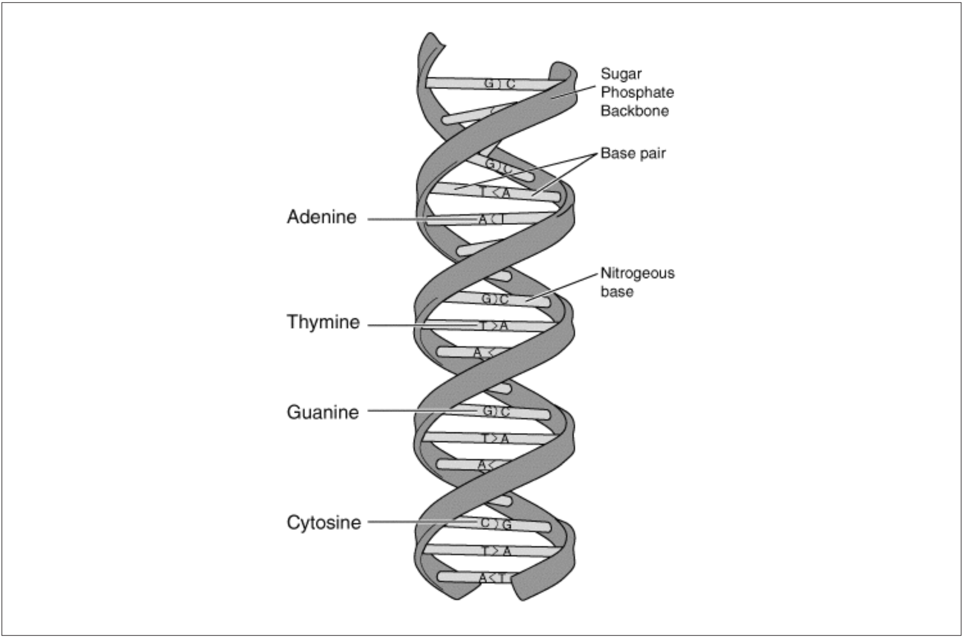

Figure 23-2. DNA double helix structure

The Genetic Code: Turning DNA Letters into Proteins

Without proteins, there is no life. DNA serves as a recipe for creating proteins. A protein is a chain of amino acids that folds into a specific 3D shape[2] to serve a particular structure or function. As there are a total of 20 amino acids[3] and only four letters in the DNA alphabet (A, C, T, G), nature groups these letters in words, called *codons*. Each codon is three bases long (since two bases would only support $4^2=16$ amino acids).

In 1968, Har Gobind Khorana, Robert W. Holley, and Marshall Nirenberg received the Nobel Prize in Physiology or Medicine for successfully mapping amino acids associated with each of the 64 codons. Each codon encodes a single amino acid, or designates the start and stop positions (see Table 23-1). Since there are 64 possible codons and only 20 amino acids, multiple codons correspond to some of the amino acids.

Table 23-1. Codon table

Amino acid	Codon(s)	Amino acid	Codon(s)
Alanine	GC{U,C,A,G}	Leucine	UU{A,G} or CU{U,C,A,G}
Arginine	CG{U,C,A,G} or AG{A,G}	Lysine	AA{A,G}
Asparagine	AA{U,C}	Methionine	AUG
Aspartic acid	GA{U,C}	Phenylalanine	UU{U,C}
Cysteine	UG{U,C}	Proline	CC{U,C,A,G}
Glutamic acid	GA{A,G}	Threonine	AC{U,C,A,G}
Glutamine	CA{A,G}	Serine	UC{U,C,A,G} or AG{U,C}
Glycine	GG{U,C,A,G}	Tryptophan	UGG
Histidine	CA{U,C}	Tyrosine	UA{U,C}
Isoleucine	AU{U,C,A}	Valine	GU{U,C,A,G}
START!	AUG	STOP!	UAA or UGA or UAG

Because every organism on Earth evolved from the same common ancestor, *every organism on Earth uses the same genetic code, with few variations*. Whether the organism is a tree, worm, fungus, or cheetah, the codon UGG encodes tryptophan. Mother Nature has been the ultimate practitioner of code reuse over the last few billion years.

DNA is not directly used to synthesize amino acids. Instead, a process called *transcription* copies the DNA sequence that codes for a protein into *messenger RNA* (mRNA). These mRNA carry information from the nuclei of your cells to the surrounding *cytoplasm* to create proteins in a process called *translation*.

2. This process is called *protein folding*. The Folding@home allows volunteers to donate CPU cycles to help researchers determine the mechanisms of protein folding.
3. There are also a few nonstandard amino acids not shown in the table that are encoded differently.

You probably noticed that this lookup table doesn't have the DNA letter T (for thymine) and has a new letter U (for uracil). During *transcription*, U is substituted for T:

```
$ echo "ATGGTGACTCCTACATGA" | sed 's/T/U/g' | fold -w 3
AUG
GUG
ACU
CCU
ACA
UGA
```

Looking up these codons in the codon table, we can determine that this particular DNA strand will translate into a protein with the following amino acids in a chain: methionine, valine, threonine, proline, and threonine. This is a contrived example, but it logically demonstrates how DNA instructs the creation of proteins that make you uniquely *you*. It's a marvel that science has allowed us to understand the language of DNA, including the start and stop punctuations.

Thinking of DNA as Source Code

At the cellular level, your body is a completely distributed system. Nothing is centralized. It's like a cluster of 37.2 trillion[4] cells executing the same code: your DNA.

If you think of your DNA as source code, here are some things to consider:

- The source is comprised of only four characters: A, C, T, and G.
- The source has two contributors, your mother and father, who contributed 3.2 billion letters each. In fact, the reference genome provided by the Genome Reference Consortium (GRC) is nothing more than an ASCII file with 3.2 billion characters inside.[5]
- The source is broken up into 25 separate files called *chromosomes* that each hold varying fractions of the source. The files are numbered, and tend to get smaller in size, with chromosome 1 holding ~250 million characters and chromosome 22 holding only ~50 million. There are also the X, Y, and mitochondrial chromosomes. The term *chromosome* basically means "colored thing," from a time when biologists could stain them but didn't know what they were.
- The source is executed on your biological machinery three letters (i.e., a codon) at a time, using the genetic code explained previously—not unlike a Turing machine that reads chemical letters instead of paper ribbon.

4. See Eva Bianconi et al., "An estimation of the number of cells in the human body," *Annals of Human Biology*, November/December 2013.

5. You might expect this to be 6.4 billion letters, but the reference genome is, for better or worse, a *haploid* representation of the average of dozens of individuals.

- The source has about 20,000 functions, called *genes*, which each create a protein when executed. The location of each gene in the source is called the *locus*. You can think of a gene as a specific range of contiguous base positions on a chromosome. For example, the BRCA1 gene implicated in breast cancer can be found on chromosome 17 from positions 41,196,312 to 41,277,500. A gene is like a "pointer" or "address," whereas alleles (described momentarily) are the actual content. Everyone has the BRCA1 gene, but not everyone has alleles that put them at risk.
- A *haplotype* is similar to an object in object-oriented programming languages that holds specific functions (genes) that are typically inherited together.
- The source has two definitions for each gene, called *alleles*—one from your mother and one from your father—which are found at the same position of paired chromosomes (while the cells in your body are *diploid*—that is, they have two alleles per gene—there are organisms that are *triploid*, *tetraploid*, etc.). Both alleles are executed and the resultant proteins interact to create a specific *phenotype*. For example, proteins that make or degrade eye color pigment lead to a particular phenotype, or an observable characteristic (e.g., blue eyes). If the alleles you inherit from your parents are identical, you're *homozygous* for that allele; otherwise, you're *heterozygous*.
- A *single-nucleic polymorphism* (SNP), pronounced "snip," is a single-character change in the source code (e.g., from `ACTGACTG` to `ACTTACTG`).
- An *indel* is short for *insert-delete* and represents an insertion or deletion from the reference genome. For example, if the reference has `CCTGACTG` and your sample has four characters inserted—say, `CCTGCCTAACTG`—then it is an indel.
- Only 0.5% of the source gets translated into the proteins that sustain your life. That portion of the source is called your *exome*. A human exome requires a few gigabytes to store in compressed binary files.
- The other 99.5% of the source is commented out and serves as word padding (*introns*); it is used to regulate when genes are turned on, repeat, and so on.[6] A *whole genome* requires a few hundred gigabytes to store in compressed binary files.
- Every cell of your body has the same source,[7] but it can be selectively commented out by *epigenetic* factors like *DNA methylation* and *histone modification*, not unlike an `#ifdef` statement for each cell type (e.g., `#ifdef RETINA` or `#ifdef LIVER`).

6. Only about 28% of your DNA is transcribed into nascent RNA, and after RNA splicing, only about 1.5% of the RNA is left to code for proteins. Evolutionary selection occurs at the DNA level, with most of your DNA providing support to the other 0.5% or being deselected altogether (as more fitting DNA evolves). There are some cancers that appear to be caused by dormant regions of DNA being resurrected, so to speak.
7. There is actually, on average, about 1 error for each billion DNA "letters" copied. So, each cell isn't *exactly* the same.

These factors are responsible for making cells in your retina operate differently than cells in your liver.

- The process of *variant calling* is similar to running *diff* between two different DNA sources.

These analogies aren't meant to be taken too literally, but hopefully they helped familiarize you with some genomics terminology.

The Human Genome Project and Reference Genomes

In 1953, Watson and Crick discovered the structure of DNA, and in 1965 Nirenberg, with help from his NIH colleagues, cracked the genetic code, which expressed the rules for translating DNA or mRNA into proteins. Scientists knew that there were millions of human proteins but didn't have a complete survey of the human genome, which made it impossible to fully understand the genes responsible for protein synthesis. For example, if each protein was created by a single gene, that would imply millions of protein-coding genes in the human genome.

In 1990, the Human Genome Project set out to determine all the chemical base pairs that make up human DNA. This collaborative, international research program published the first human genome in April of 2003,[8] at an estimated cost of $3.8 billion. The Human Genome Project generated an estimated $796 billion in economic impact, equating to a return on investment (ROI) of 141:1.[9] The Human Genome Project found about 20,500 genes—significantly fewer than the millions you would expect with a simple 1:1 model of gene to protein, since proteins can be assembled from a combination of genes, post-translational processes during folding, and other mechanisms.

While this first human genome took over a decade to build, once created, it made "bootstrapping" the subsequent sequencing of other genomes much easier. For the first genome, scientists were operating in the dark. They had no reference to search as a roadmap for constructing the full genome. There is no technology to date that can read a whole genome from start to finish; instead, there are many techniques that vary in the speed, accuracy, and length of DNA fragments they can read. Scientists in the Human Genome Project had to sequence the genome in pieces, with different pieces being more easily sequenced by different technologies. Once you have a complete human genome, subsequent human genomes become much easier to construct; you can use the first genome as a reference for the second. The fragments from the second genome can be pattern matched to the first, similar to having the picture on a jigsaw puzzle's box to

8. Intentionally 50 years after Watson and Crick's discovery of the 3D structure of DNA.
9. Jonathan Max Gitlin, "Calculating the economic impact of the Human Genome Project," June 2013.

help inform the placement of the puzzle pieces. It helps that most coding sequences are highly conserved, and *variants* only occur at 1 in 1,000 loci.

Shortly after the Human Genome Project was completed, the Genome Reference Consortium (GRC), an international collection of academic and research institutes, was formed to improve the representation of reference genomes. The GRC publishes a new human reference that serves as something like a common coordinate system or map to help analyze new genomes. The latest human reference genome, released in February 2014, was named `GRCh38`; it replaced `GRCh37`, which was released five years prior.

Sequencing and Aligning DNA

Second-generation sequencing is rapidly evolving, with numerous hardware vendors and new sequencing methods being developed about every six months; however, a common feature of all these technologies is the use of massively parallel methods, where thousands or even millions of reactions occur simultaneously. The double-stranded DNA is split down the middle, the single strands are copied many times, and the copies are randomly shredded into small fragments of different lengths called *reads*, which are placed into the sequencer. The sequencer reads the "letters" in each of these reads, in parallel for high throughput, and outputs a raw ASCII file containing each read (e.g., `AGTTTCGGGATC...`), as well as a quality estimate for each letter read, to be used for downstream analysis.

A piece of software called an *aligner* takes each read and works to find its position in the reference genome (see Figure 23-3).[10] A complete human genome is about 3 billion base (A, C, T, G) pairs long.[11] The reference genome (e.g., `GRCh38`) acts like the picture on a puzzle box, presenting the overall contours and colors of the human genome. Each short read is like a puzzle piece that needs to be fit into position as closely as possible. A common metric is "edit distance," which quantifies the number of operations necessary to transform one string to another. Identical strings have an edit distance of zero, and an indel of one letter has an edit distance of one. Since humans are 99.9% identical to one another, most of the reads will fit to the reference quite well and have a low edit distance. The challenge with building a good aligner is handling idiosyncratic reads.

10. There is also a second approach, *de novo* assembly, where reads are put into a graph data structure to create long sequences without mapping to a reference genome.

11. Each base is about 3.4 angstroms, so the DNA from a single human cell stretches over 2 meters end to end!

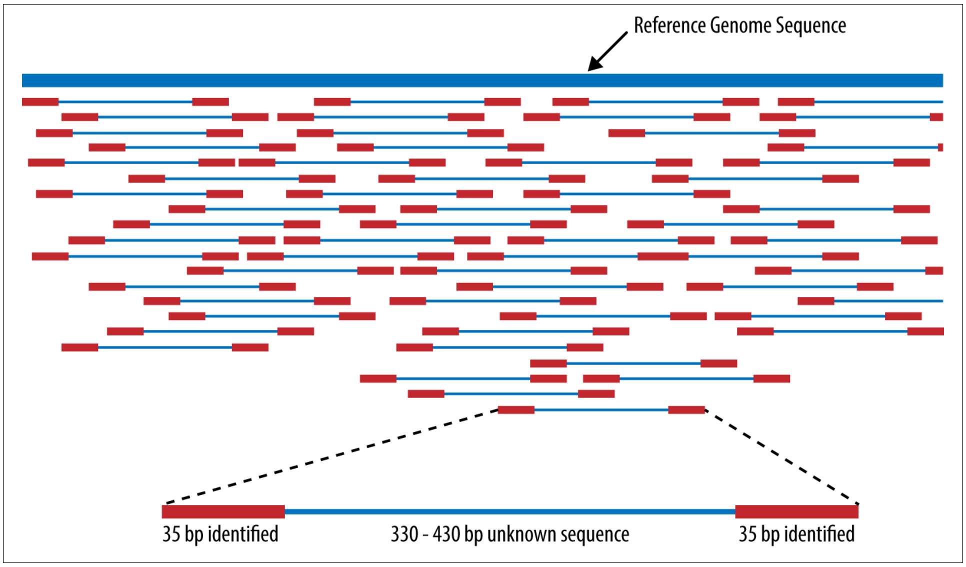

Figure 23-3. Aligning reads to a reference genome, from Wikipedia

ADAM, A Scalable Genome Analysis Platform

Aligning the reads to a reference genome is only the first of a series of steps necessary to generate reports that are useful in a clinical or research setting. The early stages of this processing pipeline look similar to any other extract-transform-load (ETL) pipelines that need data deduplication and normalization before analysis.

The sequencing process duplicates genomic DNA, so it's possible that the same DNA reads are generated multiple times; these duplicates need to be marked. The sequencer also provides a quality estimate for each DNA "letter" that it reads, which has sequencer-specific biases that need to be adjusted. Aligners often misplace reads that have indels (inserted or deleted sequences) that need to be repositioned on the reference genome. Currently, this preprocessing is done using single-purpose tools launched by shell scripts on a single machine. These tools take multiple days to finish the processing of whole genomes. The process is disk bound, with each stage writing a new file to be read into subsequent stages, and is an ideal use case for applying general-purpose big data technology. ADAM is able to handle the same preprocessing in under two hours.

ADAM is a genome analysis platform that focuses on rapidly processing petabytes of high-coverage, whole genome data. ADAM relies on Apache Avro, Parquet, and Spark. These systems provide many benefits when used together, since they:

- Allow developers to focus on algorithms without needing to worry about distributed system failures
- Enable jobs to be run locally on a single machine, on an in-house cluster, or in the cloud *without changing code*
- Compress legacy genomic formats and provide predicate pushdown and projection for performance
- Provide an agile way of customizing and evolving data formats
- Are designed to easily scale out using only commodity hardware
- Are shared with a standard Apache 2.0 license[12]

Literate programming with the Avro interface description language (IDL)

The Sequence Alignment/Map (SAM) specification defines the mandatory fields listed in Table 23-2.

Table 23-2. Mandatory fields in the SAM format

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	[!-?A-~]{1,255}	Query template NAME
2	FLAG	Int	[0, 2^{16}-1]	bitwise FLAG
3	RNAME	String	*|[!-()+-<>-~][!-~]*	Reference sequence NAME
4	POS	Int	[0,2^{31}-1]	1-based leftmost mapping POSition
5	MAPQ	Int	[0,2^{8}-1]	MAPping Quality
6	CIGAR	String	*|([0-9]+[MIDNSHPX=])+	CIGAR string
7	RNEXT	String	*|=|[!-()+-><-~][!-~]*	Ref. name of the mate/NEXT read
8	PNEXT	Int	[0,2^{31}-1]	Position of the mate/NEXT read
9	TLEN	Int	[-2^{31}+1,2^{31}-1]	observed Template LENgth
10	SEQ	String	*|[A-Za-z=.]+	segment SEQuence
11	QUAL	String	[!-~]	ASCII of Phred-scaled base QUALity+33

Any developers who want to implement this specification need to translate this English spec into their computer language of choice. In ADAM, we have chosen instead to use

12. Unfortunately, some of the more popular software in genomics has an ill-defined or custom, restrictive license. Clean open source licensing and source code are necessary for science to make it easier to reproduce and understand results.

literate programming with a spec defined in Avro IDL. For example, the mandatory fields for SAM can be easily expressed in a simple Avro record:

```
record AlignmentRecord {
  string qname;
  int flag;
  string rname;
  int pos;
  int mapq;
  string cigar;
  string rnext;
  int pnext;
  int tlen;
  string seq;
  string qual;
}
```

Avro is able to autogenerate native Java (or C++, Python, etc.) classes for reading and writing data and provides standard interfaces (e.g., Hadoop's `InputFormat`) to make integration with numerous systems easy. Avro is also designed to make schema evolution easier. In fact, the ADAM schemas we use today have evolved to be more sophisticated, expressive, and customized to express a variety of genomic models such as structural variants, genotypes, variant calling annotations, variant effects, and more.

UC Berkeley is a member of the Global Alliance for Genomics & Health, a non-governmental, public-private partnership consisting of more than 220 organizations across 30 nations, with the goal of maximizing the potential of genomics medicine through effective and responsible data sharing. The Global Alliance has embraced this literate programming approach and publishes its schemas in Avro IDL as well. Using Avro has allowed researchers around the world to talk about data at the logical level, without concern for computer languages or on-disk formats.

Column-oriented access with Parquet

The SAM and BAM[13] file formats are *row-oriented*: the data for each record is stored together as a single line of text or a binary record. (See "Other File Formats and Column-Oriented Formats" on page 136 for further discussion of row- versus column-oriented formats.) A single paired-end read in a SAM file might look like this:

```
read1   99 chrom1  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGATACTG *
read1  147 chrom1 37 30 9M         =  7 -39 CAGCGGCAT         * NM:i:1
```

A typical SAM/BAM file contains many millions of rows, one for each DNA read that came off the sequencer. The preceding text fragment translates loosely into the view shown in Table 23-3.

13. BAM is the compressed binary version of the SAM format.

Table 23-3. Logical view of SAM fragment

Name	Reference	Position	MapQ	CIGAR	Sequence
`read1`	`chromo some1`	7	30	`8M2I4M1D3M`	`TTAGA TAAAGGA TACTG`
`read1`	`chromo some1`	37	30	`9M`	`CAGCGG CAT`

In this example, the read, identified as `read1`, was mapped to the reference genome at `chromosome1`, positions 7 and 37. This is called a "paired-end" read as it represents a single strand of DNA that was read from each end by the sequencer. By analogy, it's like reading an array of length 150 from `0..50` and `150..100`.

The `MapQ` score represents the probability that the sequence is mapped to the reference correctly. `MapQ` scores of 20, 30, and 40 have a probability of being correct of 99%, 99.9%, and 99.99%, respectively. To calculate the probability of error from a `MapQ` score, use the expression $10^{(-\mathrm{MapQ}/10)}$ (e.g., $10^{(-30/10)}$ is a probability of 0.001).

The `CIGAR` explains how the individual nucleotides in the DNA sequence map to the reference.[14] The `Sequence` is, of course, the DNA sequence that was mapped to the reference.

There is a stark mismatch between the SAM/BAM *row-oriented* on-disk format and the *column-oriented* access patterns common to genome analysis. Consider the following:

- A range query to find data for a particular gene linked to breast cancer, named BRCA1: "Find all reads that cover chromosome 17 from position 41,196,312 to 41,277,500"
- A simple filter to find poorly mapped reads: "Find all reads with a `MapQ` less than 10"
- A search of all reads with insertions or deletions, called *indels*: "Find all reads that contain `I` or `D` in the `CIGAR` string"
- Count the number of unique *k*-mers: "Read every `Sequence` and generate all possible substrings of length *k* in the string"

Parquet's predicate pushdown feature allows us to rapidly filter reads for analysis (e.g., finding a gene, ignoring poorly mapped reads). Projection allows for precise materialization of only the columns of interest (e.g., reading only the sequences for *k*-mer counting).

14. The first record's Compact Idiosyncratic Gap Alignment Report (CIGAR) string is translated as "8 matches (8M), 2 inserts (2I), 4 matches (4M), 1 delete (1D), 3 matches (3M)."

Additionally, a number of the fields have low cardinality, making them ideal for data compression techniques like run-length encoding (RLE). For example, given that humans have only 23 pairs of chromosomes, the `Reference` field will have only a few dozen unique values (e.g., `chromosome1`, `chromosome17`, etc.). We have found that storing BAM records inside Parquet files results in ~20% compression. Using the `PrintFooter` command in Parquet, we have found that quality scores can be run-length encoded and bit-packed to compress ~48%, but they still take up ~70% of the total space. We're looking forward to Parquet 2.0, so we can use delta encoding on the quality scores to compress the file size even more.

A simple example: *k*-mer counting using Spark and ADAM

Let's do "word count" for genomics: counting *k*-mers. The term *k-mers* refers to all the possible subsequences of length *k* for a read. For example, if you have a read with the sequence `AGATCTGAAG`, the 3-mers for that sequence would be `['AGA', 'GAT', 'ATC', 'TCT', 'CTG', 'TGA', 'GAA', 'AAG']`. While this is a trivial example, *k*-mers are useful when building structures like De Bruijn graphs for sequence assembly. In this example, we are going to generate all the possible 21-mers from our reads, count them, and then write the totals to a text file.

This example assumes that you've already created a `SparkContext` named `sc`. First, we create a Spark RDD of `AlignmentRecords` using a pushdown predicate to remove low-quality reads and a projection to only materialize the `sequence` field in each read:

```
// Load reads from 'inputPath' into an RDD for analysis
val adamRecords: RDD[AlignmentRecord] = sc.adamLoad(args.inputPath,

  // Filter out all low-quality reads that failed vendor quality checks
  predicate = Some(classOf[HighQualityReadsPredicate]),

  // Only materialize the 'sequence' from each record
  projection = Some(Projection(AlignmentRecordField.sequence)))
```

Since Parquet is a column-oriented storage format, it can rapidly materialize only the `sequence` column and quickly skip over the unwanted fields. Next, we walk over each `sequence` using a sliding window of length *k*=21, emit a count of `1L`, and then `reduceByKey` using the *k*-mer subsequence as the key to get the total counts for the input file:

```
// The length of k-mers we want to count
val kmerLength = 21

// Process the reads into an RDD of tuples with k-mers and counts
val kmers: RDD[(String, Long)] = adamRecords.flatMap(read => {
  read.getSequence
    .toString
    .sliding(kmerLength)
    .map(k => (k, 1L))
}).reduceByKey { case (a, b) => a + b}
```

```
// Print the k-mers as a text file to the 'outputPath'
kmers.map { case (kmer, count) => s"$count,$kmer"}
  .saveAsTextFile(args.outputPath)
```

When run on sample `NA21144`, chromosome 11 in the 1000 Genomes project,[15] this job outputs the following:

```
AAAAAAAAAAAAAAAAAAAAAA, 124069
TTTTTTTTTTTTTTTTTTTTTT, 120590
ACACACACACACACACACACAC, 41528
GTGTGTGTGTGTGTGTGTGTGT, 40905
CACACACACACACACACACACA, 40795
TGTGTGTGTGTGTGTGTGTGTG, 40329
TAATCCCAGCACTTTGGGAGGC, 32122
TGTAATCCCAGCACTTTGGGAG, 31206
CTGTAATCCCAGCACTTTGGGA, 30809
GCCTCCCAAAGTGCTGGGATTA, 30716
...
```

ADAM can do much more than just count *k*-mers. Aside from the preprocessing stages already mentioned—duplicate marking, base quality score recalibration, and indel realignment—it also:

- Calculates coverage read depth at each variant in a Variant Call Format (VCF) file
- Counts the *k*-mers/*q*-mers from a read dataset
- Loads gene annotations from a Gene Transfer Format (GTF) file and outputs the corresponding gene models
- Prints statistics on all the reads in a read dataset (e.g., % mapped to reference, number of duplicates, reads mapped cross-chromosome, etc.)
- Launches legacy variant callers, pipes reads into *stdin*, and saves output from *stdout*
- Comes with a basic genome browser to view reads in a web browser

However, the most important thing ADAM provides is an open, scalable platform. All artifacts are published to Maven Central (search for group ID `org.bdgenomics`) to make it easy for developers to benefit from the foundation ADAM provides. ADAM data is stored in Avro and Parquet, so you can also use systems like SparkSQL, Impala, Apache Pig, Apache Hive, or others to analyze the data. ADAM also supports job written in Scala, Java, and Python, with more language support on the way.

At Scala.IO in Paris in 2014, Andy Petrella and Xavier Tordoir used Spark's MLlib *k*-means with ADAM for population stratification across the 1000 Genomes dataset (pop-

15. Arguably the most popular publicly available dataset, found at *http://www.1000genomes.org*.

ulation stratification is the process of assigning an individual genome to an ancestral group). They found that ADAM/Spark improved performance by a factor of 150.

From Personalized Ads to Personalized Medicine

While ADAM is designed to rapidly and scalably analyze aligned reads, it does not align the reads itself; instead, ADAM relies on standard short-reads aligners. The Scalable Nucleotide Alignment Program (SNAP) is a collaborative effort including participants from Microsoft Research, UC San Francisco, and the AMPLab as well as open source developers, shared with an Apache 2.0 license. The SNAP aligner is as accurate as the current best-of-class aligners, like BWA-mem, Bowtie2, and Novalign, but runs between 3 and 20 times faster. This speed advantage is important when doctors are racing to identify a pathogen.

In 2013, a boy went to the University of Wisconsin Hospital and Clinics' Emergency Department three times in four months with symptoms of encephalitis: fevers and headaches. He was eventually hospitalized without a successful diagnosis after numerous blood tests, brain scans, and biopsies. Five weeks later, he began having seizures that required he be placed into a medically induced coma. In desperation, doctors sampled his spinal fluid and sent it to an experimental program led by Charles Chiu at UC San Francisco, where it was sequenced for analysis. The speed and accuracy of SNAP allowed UCSF to quickly filter out all human DNA and, from the remaining 0.02% of the reads, identify a rare infectious bacterium, *Leptospira santarosai*. They reported the discovery to the Wisconsin doctors *just two days after they sent the sample*. The boy was treated with antibiotics for 10 days, awoke from his coma, and was discharged from the hospital two weeks later.[16]

If you're interested in learning more about the system the Chiu lab used—called Sequence-based Ultra-Rapid Pathogen Identification (SURPI)—they have generously shared their software with a permissive BSD license and provide an Amazon EC2 Machine Image (AMI) with SURPI preinstalled. SURPI collects 348,922 unique bacterial sequences and 1,193,607 unique virus sequences from numerous sources and saves them in 29 SNAP-indexed databases, each approximately 27 GB in size, for fast search.

Today, more data is analyzed for personalized advertising than personalized medicine, but that will not be the case in the future. With personalized medicine, people receive customized healthcare that takes into consideration their unique DNA profiles. As the price of sequencing drops and more people have their genomes sequenced, the increase in statistical power will allow researchers to understand the genetic mechanisms underlying diseases and fold these discoveries into the personalized medical model, to

16. Michael Wilson et al., "Actionable Diagnosis of Neuroleptospirosis by Next-Generation Sequencing," *New England Journal of Medicine*, June 2014.

improve treatment for subsequent patients. While only 25 PB of genomic data were generated worldwide this year, next year that number will likely be 100 PB.

Join In

While we're off to a great start, the ADAM project is still an experimental platform and needs further development. If you're interested in learning more about programming on ADAM or want to contribute code, take a look at *Advanced Analytics with Spark: Patterns for Learning from Data at Scale* by Sandy Ryza et al. (O'Reilly, 2014), which includes a chapter on analyzing genomics data with ADAM and Spark. You can find us at *http://bdgenomics.org*, on IRC at #adamdev, or on Twitter at @bigdatagenomics.

CHAPTER 24

Cascading

Chris K. Wensel

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

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

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

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

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

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

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

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

Fields, Tuples, and Pipes

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

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

So, in the abstract, keys and values not only bind map to reduce, but reduce to the next map, and then to the next reduce, and so on (Figure 24-1). That is, key-value pairs are sourced from input files and stream through chains of map and reduce operations, and finally rest in an output file. When you implement enough of these chained MapReduce applications, you start to see a well-defined set of key-value manipulations used over and over again to modify the key-value data stream.

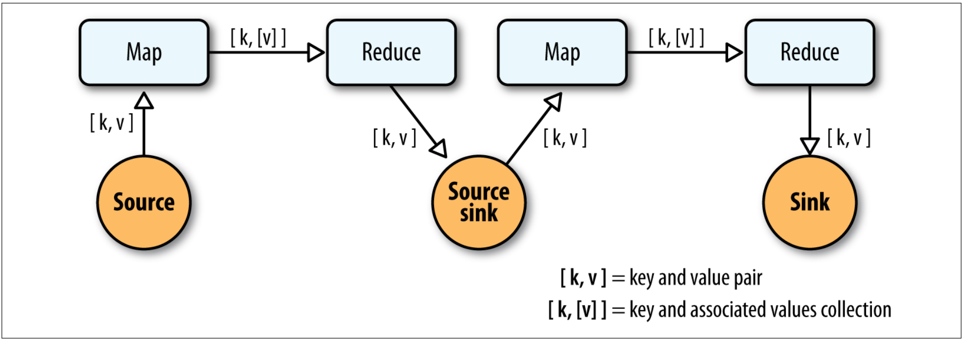

Figure 24-1. Counting and sorting in MapReduce

Cascading simplifies this by abstracting away keys and values and replacing them with tuples that have corresponding field names, similar in concept to tables and column names in a relational database. During processing, streams of these fields and tuples are then manipulated as they pass through user-defined operations linked together by pipes (Figure 24-2).

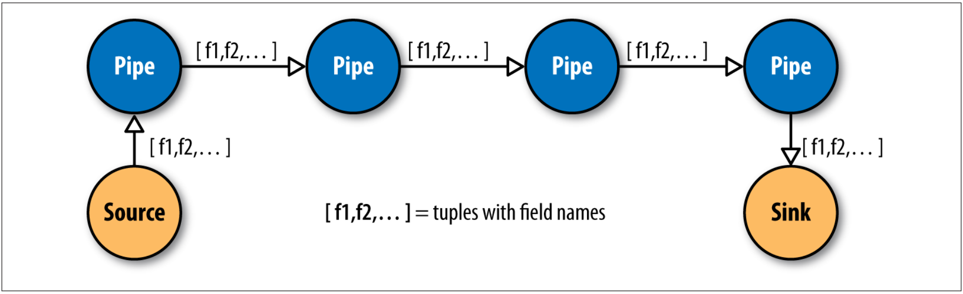

Figure 24-2. Pipes linked by fields and tuples

So, MapReduce keys and values are reduced to:

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

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

And the map and reduce operations are abstracted behind one or more pipe instances (Figure 24-3):

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

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

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

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

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

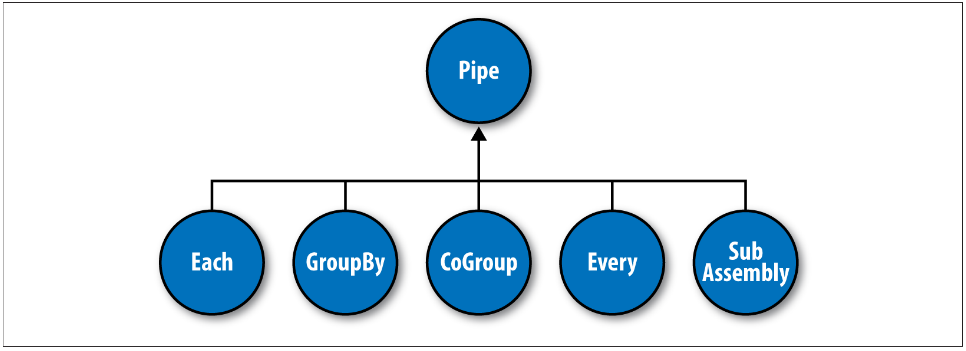

Figure 24-3. Pipe types

All these pipes are chained together by the developer into "pipe assemblies," in which each assembly can have many input tuple streams (sources) and many output tuple streams (sinks). See Figure 24-4.

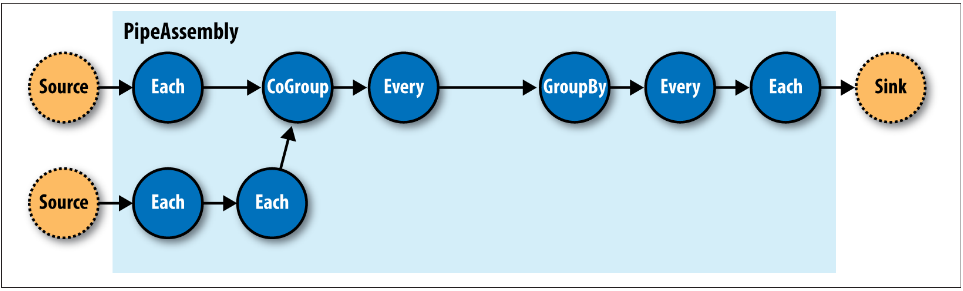

Figure 24-4. A simple PipeAssembly

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

For example, a developer who wanted to provide a "secondary sorting" of reducer values would need to implement a map, a reduce, a "composite" key (two keys nested in a parent key), a value, a partitioner, an "output value grouping" comparator, and an "output key" comparator, all of which would be coupled to one another in varying ways, and very likely would not be reusable in subsequent applications.

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

Operations

As mentioned earlier, Cascading departs from MapReduce by introducing alternative operations that are applied either to individual tuples or groups of tuples (Figure 24-5):

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

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

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

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

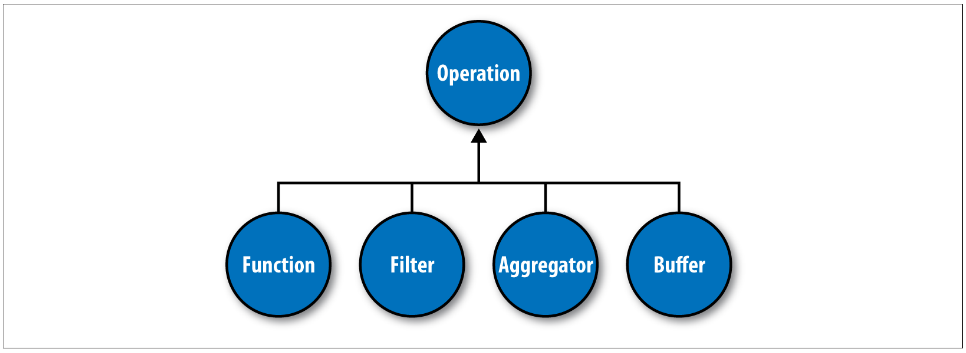

Figure 24-5. Operation types

Operations are bound to pipes when the pipe assembly is created (Figure 24-6).

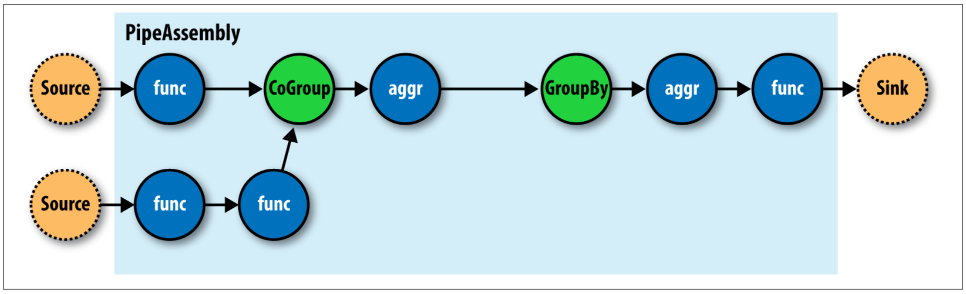

Figure 24-6. An assembly of operations

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

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

Taps, Schemes, and Flows

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

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

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

Note that `Taps` are not part of a pipe assembly, and so they are not a type of `Pipe`. But they are connected with pipe assemblies when they are made cluster executable. When a pipe assembly is connected with the necessary number of source and sink `Tap` instances, we get a `Flow`. The `Taps` either emit or capture the field names the pipe assembly expects. That is, if a `Tap` emits a tuple with the field name "line" (by reading data from a file on HDFS), the head of the pipe assembly must be expecting a "line" value as well. Otherwise, the process that connects the pipe assembly with the `Taps` will immediately fail with an error.

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

If you think of a pipe assembly like a Java class, then a `Flow` is like a Java object instance (Figure 24-7). That is, the same pipe assembly can be "instantiated" many times into new `Flows`, in the same application, without fear of any interference between them. This allows pipe assemblies to be created and shared like standard Java libraries.

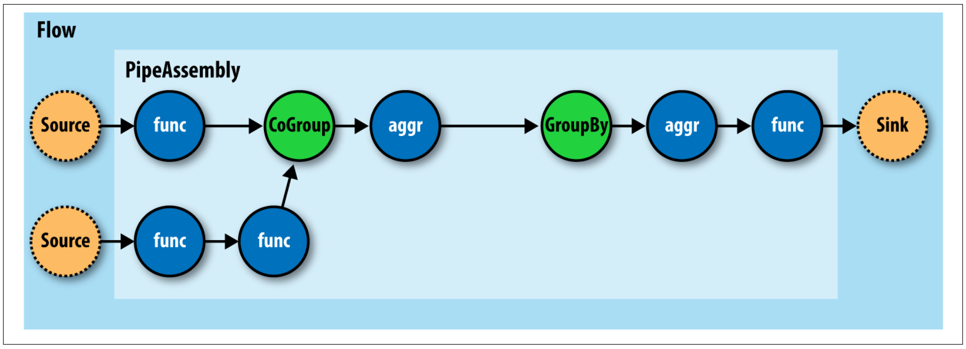

Figure 24-7. A Flow

Cascading in Practice

Now that we know what Cascading is and have a good idea of how it works, what does an application written in Cascading look like? See Example 24-1.

Example 24-1. Word count and sort

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

Scheme sinkScheme = new TextLine(); ➌
Tap sink =
    new Hfs(sinkScheme, outputPath, SinkMode.REPLACE); ➍

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

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

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

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

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

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

```
flow.complete();⓫
```

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

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

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

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

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

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

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

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

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

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

One obvious problem with this example is that some words might have uppercase letters in some instances—for example, "the" and "The" when the word comes at the beginning of a sentence. We might consider inserting a new operation to force all the words to lowercase, but we realize that all future applications that need to parse words from documents should have the same behavior, so we'll instead create a reusable pipe called SubAssembly, just like we would by creating a subroutine in a traditional application (see Example 24-2).

Example 24-2. Creating a SubAssembly

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

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

  setTails(previous); ❸
  }
 }
```

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

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

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

First, we create a `SubAssembly` pipe to hold our "parse words" pipe assembly. Because this is a Java class, it can be reused in any other application, as long as there is an incoming field named "word" (Example 24-3). Note that there are ways to make this function even more generic, but they are covered in the Cascading User Guide.

Example 24-3. Extending word count and sort with a SubAssembly

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

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

Pipe assembly = new Pipe("wordcount");

assembly =
    new ParseWordsAssembly(assembly); ❶

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

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

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

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

```
flow.complete();
```

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

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

Flexibility

Let's take a step back and see what this new model has given us—or better yet, what it has taken away.

You see, we no longer think in terms of MapReduce jobs, or `Mapper` and `Reducer` interface implementations and how to bind or link subsequent MapReduce jobs to the ones that precede them. During runtime, the Cascading "planner" figures out the optimal way to partition the pipe assembly into MapReduce jobs and manages the linkages between them (Figure 24-8).

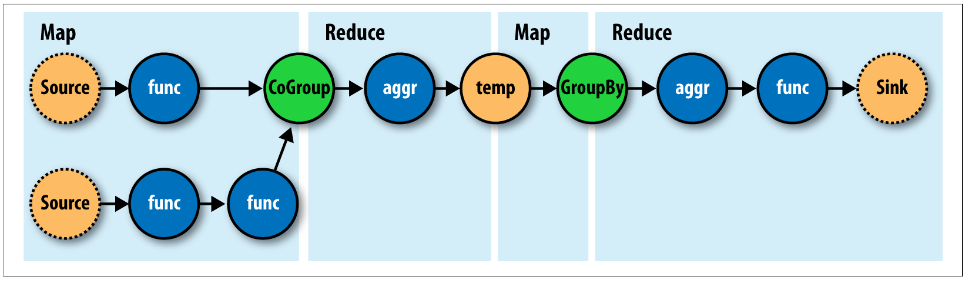

Figure 24-8. How a Flow translates to chained MapReduce jobs

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

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

Hadoop and Cascading at ShareThis

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

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

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

In this overview, we will focus on the "log processing pipeline" (Figure 24-9). This pipeline simply takes data stored in an S3 bucket, processes it (as described shortly), and stores the results back into another bucket. The Simple Queue Service (SQS) is used to coordinate the events that mark the start and completion of data processing runs. Downstream, other processes pull data to load into AsterData, pull URL lists from Hypertable to source a web crawl, or pull crawled page data to create Lucene indexes for use by Katta. Note that Hadoop is central to the ShareThis architecture. It is used to coordinate the processing and movement of data between architectural components.

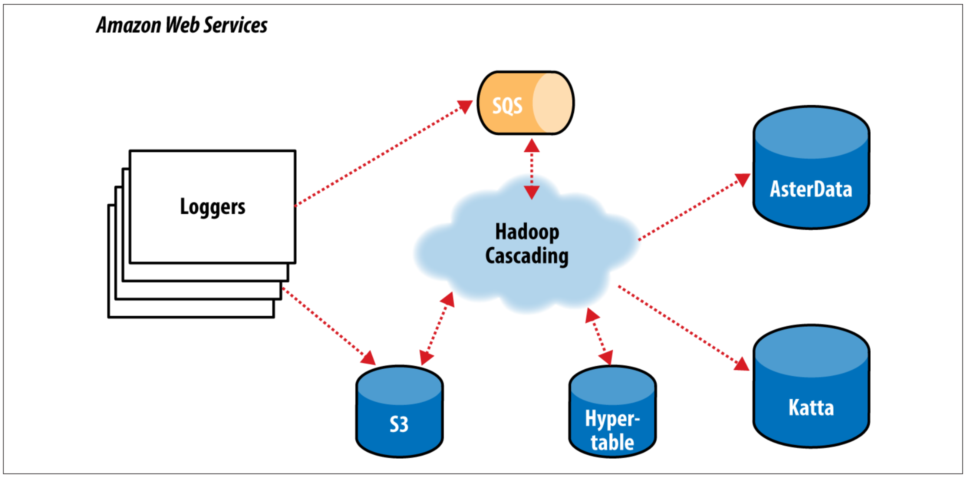

Figure 24-9. The ShareThis log processing pipeline

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

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

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

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

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

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

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

In the unit test in Example 24-4, we see the splitter isn't being tested, but it is added in another integration test not shown.

Example 24-4. Unit testing a Flow

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

  Pipe pipe = new Pipe("parser");

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

  pipe = new LogParser(pipe);

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

  Flow flow = new FlowConnector().connect(source, sink, pipe);
  flow.complete(); // run the test flow

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

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

In production, all the subassemblies are joined and planned into a `Flow`, but instead of just source and sink `Taps`, trap `Taps` were planned in (Figure 24-10). Normally, when an operation throws an exception from a remote mapper or reducer task, the `Flow` will fail and kill all its managed MapReduce jobs. When a `Flow` has traps, any exceptions are caught and the data causing the exception is saved to the `Tap` associated with the current trap. Then the next `Tuple` is processed without stopping the `Flow`. Sometimes you want your `Flows` to fail on errors, but in this case, the ShareThis developers knew they could go back and look at the "failed" data and update their unit tests while the production system kept running. Losing a few hours of processing time was worse than losing a couple of bad records.

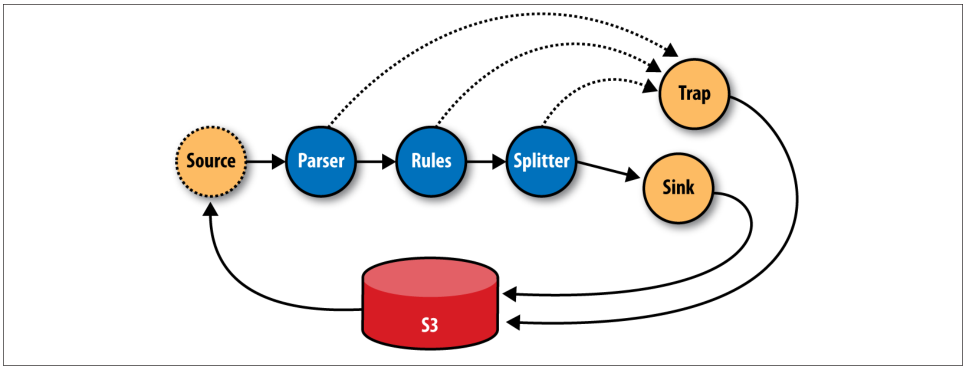

Figure 24-10. The ShareThis log processing flow

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

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

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

Summary

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

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

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