6 Join the NoSQL movement

This chapter covers

- Understanding NoSQL databases and why they're used today
- Identifying the differences between NoSQL and relational databases
- Defining the ACID principle and how it relates to the NoSQL BASE principle
- Learning why the CAP theorem is important for multi-node database setup
- Applying the data science process to a project with the NoSQL database Elasticsearch

This chapter is divided into two parts: a theoretical start and a practical finish.

- In the first part of this chapter we'll look into NoSQL databases in general and answer these questions: Why do they exist? Why not until recently? What types are there and why should you care?
- In part two we'll tackle a real-life problem—disease diagnostics and profiling—using freely available data, Python, and a NoSQL database.

No doubt you've heard about NoSQL databases and how they're used religiously by many high-tech companies. But what are NoSQL databases and what makes them so different from the relational or SQL databases you're used to? *NoSQL* is short for *Not Only Structured Query Language*, but although it's true that NoSQL databases can allow you to query them with SQL, you don't have to focus on the actual name. Much debate has already raged over the name and whether this group of new databases should even have a collective name at all. Rather, let's look at what they represent as opposed to *relational database management systems (RDBMS)*. Traditional databases reside on a single computer or server. This used to be fine as a long as your data didn't outgrow your server, but it hasn't been the case for many companies for a long time now. With the growth of the internet, companies such as Google and Amazon felt they were held back by these single-node databases and looked for alternatives.

Numerous companies use single-node NoSQL databases such as MongoDB because they want the flexible schema or the ability to hierarchically aggregate data. Here are several early examples:

- Google's first NoSQL solution was Google BigTable, which marked the start of the *columnar databases*.[1]
- Amazon came up with Dynamo, *a key-value store* .[2]
- Two more database types emerged in the quest for partitioning: the *document store* and the *graph database*.

We'll go into detail on each of the four types later in the chapter.

Please note that, although size was an important factor, these databases didn't originate solely from the need to handle larger volumes of data. Every *V* of big data has influence (volume, variety, velocity, and sometimes veracity). Graph databases, for instance, can handle network data. Graph database enthusiasts even claim that everything can be seen as a network. For example, how do you prepare dinner? With ingredients. These ingredients are brought together to form the dish and can be used along with other ingredients to form other dishes. Seen from this point of a view, ingredients and recipes are part of a network. But recipes and ingredients could also be stored in your relational database or a document store; it's all how you look at the problem. Herein lies the strength of NoSQL: the ability to look at a problem from a different angle, shaping the data structure to the use case. As a data scientist, your job is to find the best answer to any problem. Although sometimes this is still easier to attain using RDBMS, often a particular NoSQL database offers a better approach.

Are relational databases doomed to disappear in companies with big data because of the need for partitioning? No, NewSQL platforms (not to be confused with NoSQL) are the RDBMS answer to the need for cluster setup. NewSQL databases follow the relational model but are capable of being divided into a distributed cluster like NoSQL

[1] See http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf.

[2] See http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

databases. It's not the end of relational databases and certainly not the end of SQL, as platforms like Hive translate SQL into MapReduce jobs for Hadoop. Besides, not every company needs big data; many do fine with small databases and the traditional relational databases are perfect for that.

If you look at the big data mind map shown in figure 6.1, you'll see four types of NoSQL databases.

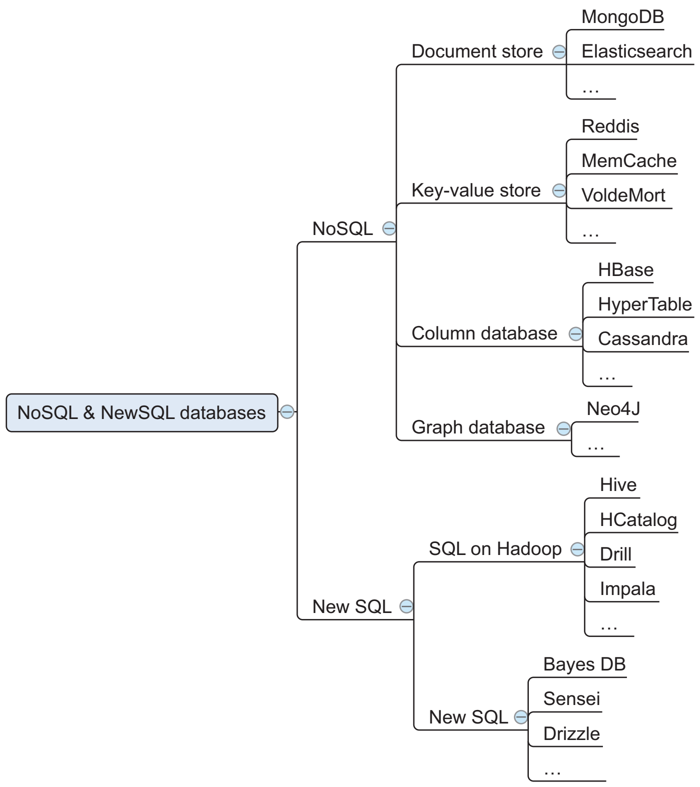


Figure 6.1 NoSQL and NewSQL databases

These four types are document store, key-value store, graph database, and column database. The mind map also includes the NewSQL partitioned relational databases. In the future this big split between NoSQL and NewSQL will become obsolete because every database type will have its own focus, while combining elements from both NoSQL and NewSQL databases. The lines are slowly blurring as RDBMS types get NoSQL features such as the column-oriented indexing seen in columnar databases. But for now it's a good way to show that the old relational databases have moved past their single-node setup, while other database types are emerging under the NoSQL denominator.

Let's look at what NoSQL brings to the table.

6.1 Introduction to NoSQL

As you've read, the goal of NoSQL databases isn't only to offer a way to partition databases successfully over multiple nodes, but also to present fundamentally different ways to model the data at hand to fit its structure to its use case and not to how a relational database requires it to be modeled.

To help you understand NoSQL, we're going to start by looking at the core ACID principles of single-server relational databases and show how NoSQL databases rewrite them into BASE principles so they'll work far better in a distributed fashion. We'll also look at the CAP theorem, which describes the main problem with distributing databases across multiple nodes and how ACID and BASE databases approach it.

6.1.1 ACID: the core principle of relational databases

The main aspects of a traditional relational database can be summarized by the concept ACID:

- *Atomicity*—The "all or nothing" principle. If a record is put into a database, it's put in completely or not at all. If, for instance, a power failure occurs in the middle of a database write action, you wouldn't end up with half a record; it wouldn't be there at all.
- *Consistency*—This important principle maintains the integrity of the data. No entry that makes it into the database will ever be in conflict with predefined rules, such as lacking a required field or a field being numeric instead of text.
- *Isolation*—When something is changed in the database, nothing can happen on this exact same data at exactly the same moment. Instead, the actions happen in serial with other changes. Isolation is a scale going from low isolation to high isolation. On this scale, traditional databases are on the "high isolation" end. An example of low isolation would be Google Docs: Multiple people can write to a document at the exact same time and see each other's changes happening instantly. A traditional Word document, on the other end of the spectrum, has high isolation; it's locked for editing by the first user to open it. The second person opening the document can view its last saved version but is unable to see unsaved changes or edit the document without first saving it as a copy. So once someone has it opened, the most up-to-date version is completely isolated from anyone but the editor who locked the document.
- *Durability*—If data has entered the database, it should survive permanently. Physical damage to the hard discs will destroy records, but power outages and software crashes should not.

ACID applies to all relational databases and certain NoSQL databases, such as the graph database Neo4j. We'll further discuss graph databases later in this chapter and in chapter 7. For most other NoSQL databases another principle applies: BASE. To understand BASE and why it applies to most NoSQL databases, we need to look at the CAP Theorem.

6.1.2 CAP Theorem: the problem with DBs on many nodes

Once a database gets spread out over different servers, it's difficult to follow the ACID principle because of the consistency ACID promises; the CAP Theorem points out why this becomes problematic. The CAP Theorem states that a database can be any two of the following things but never all three:

- *Partition tolerant*—The database can handle a network partition or network failure.
- *Available*—As long as the node you're connecting to is up and running and you can connect to it, the node will respond, even if the connection between the different database nodes is lost.
- *Consistent*—No matter which node you connect to, you'll always see the exact same data.

For a single-node database it's easy to see how it's always available and consistent:

- *Available*—As long as the node is up, it's available. That's all the CAP availability promises.
- *Consistent*—There's no second node, so nothing can be inconsistent.

Things get interesting once the database gets partitioned. Then you need to make a choice between availability and consistency, as shown in figure 6.2.

Let's take the example of an online shop with a server in Europe and a server in the United States, with a single distribution center. A German named Fritz and an American named Freddy are shopping at the same time on that same online shop. They see an item and only one is still in stock: a bronze, octopus-shaped coffee table. Disaster strikes, and communication between the two local servers is temporarily down. If you were the owner of the shop, you'd have two options:

- *Availability*—You allow the servers to keep on serving customers, and you sort out everything afterward.
- *Consistency*—You put all sales on hold until communication is reestablished.

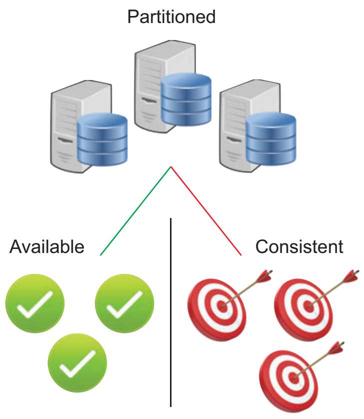


Figure 6.2 CAP Theorem: when partitioning your database, you need to choose between availability and consistency.

In the first case, Fritz and Freddy will both buy the octopus coffee table, because the last-known stock number for both nodes is "one" and both nodes are allowed to sell it, as shown in figure 6.3.

If the coffee table is hard to come by, you'll have to inform either Fritz or Freddy that he won't receive his table on the promised delivery date or, even worse, he will

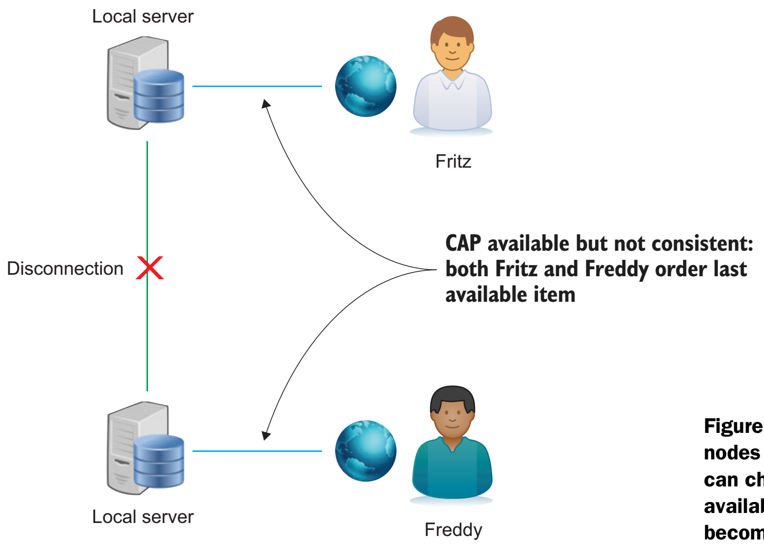

Local server
Fritz
Disconnection
CAP available but not consistent: both Fritz and Freddy order last available item
Local server
Freddy

Figure 6.3 CAP Theorem: if nodes get disconnected, you can choose to remain available, but the data could become inconsistent.

never receive it. As a good businessperson, you might compensate one of them with a discount coupon for a later purchase, and everything might be okay after all.

The second option (figure 6.4) involves putting the incoming requests on hold temporarily.

This might be fair to both Fritz and Freddy if after five minutes the web shop is open for business again, but then you might lose both sales and probably many more. Web shops tend to choose availability over consistency, but it's not the optimal choice

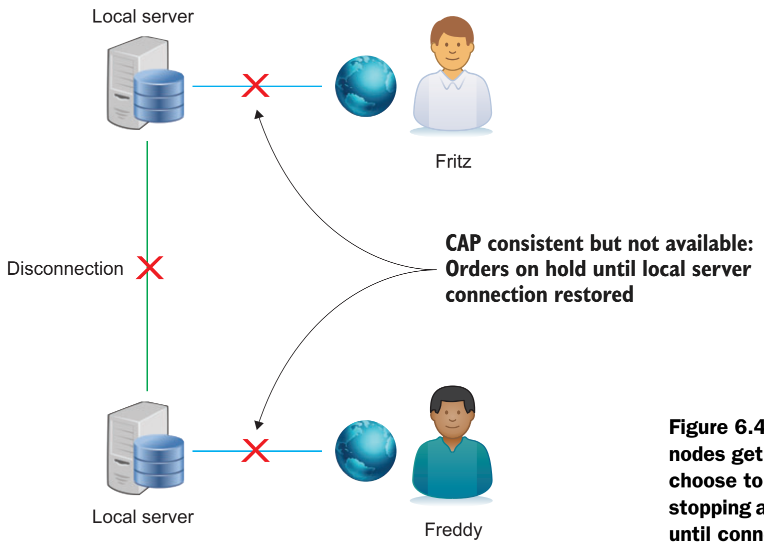

Local server
Fritz
Disconnection
CAP consistent but not available: Orders on hold until local server connection restored
Local server
Freddy

Figure 6.4 CAP Theorem: if nodes get disconnected, you can choose to remain consistent by stopping access to the databases until connections are restored

in all cases. Take a popular festival such as Tomorrowland. Festivals tend to have a maximum allowed capacity for safety reasons. If you sell more tickets than you're allowed because your servers kept on selling during a node communication failure, you could sell double the number allowed by the time communications are reestablished. In such a case it might be wiser to go for consistency and turn off the nodes temporarily. A festival such as Tomorrowland is sold out in the first couple of hours anyway, so a little downtime won't hurt as much as having to withdraw thousands of entry tickets.

6.1.3 The BASE principles of NoSQL databases

RDBMS follows the ACID principles; NoSQL databases that don't follow ACID, such as the document stores and key-value stores, follow BASE. BASE is a set of much softer database promises:

- *Basically available*—Availability is guaranteed in the CAP sense. Taking the web shop example, if a node is up and running, you can keep on shopping. Depending on how things are set up, nodes can take over from other nodes. Elasticsearch, for example, is a NoSQL document–type search engine that divides and replicates its data in such a way that node failure doesn't necessarily mean service failure, via the process of *sharding*. Each *shard* can be seen as an individual database server instance, but is also capable of communicating with the other shards to divide the workload as efficiently as possible (figure 6.5). Several shards can be present on a single node. If each shard has a replica on another node, node failure is easily remedied by re-dividing the work among the remaining nodes.
- *Soft state*—The state of a system might change over time. This corresponds to the *eventual consistency principle*: the system might have to change to make the data

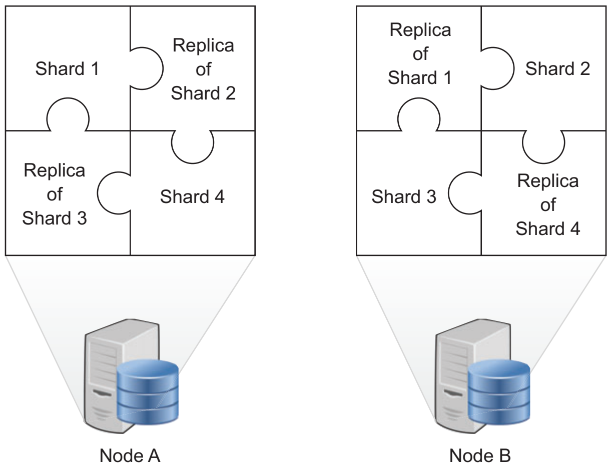


Figure 6.5 Sharding: each shard can function as a self-sufficient database, but they also work together as a whole. The example represents two nodes, each containing four shards: two main shards and two replicas. Failure of one node is backed up by the other.

consistent again. In one node the data might say "A" and in the other it might say "B" because it was adapted. Later, at conflict resolution when the network is back online, it's possible the "A" in the first node is replaced by "B." Even though no one did anything to explicitly change "A" into "B," it will take on this value as it becomes consistent with the other node.

- *Eventual consistency*—The database will become consistent over time. In the web shop example, the table is sold twice, which results in data inconsistency. Once the connection between the individual nodes is reestablished, they'll communicate and decide how to resolve it. This conflict can be resolved, for example, on a first-come, first-served basis or by preferring the customer who would incur the lowest transport cost. Databases come with default behavior, but given that there's an actual business decision to make here, this behavior can be overwritten. Even if the connection is up and running, latencies might cause nodes to become inconsistent. Often, products are kept in an online shopping basket, but putting an item in a basket doesn't lock it for other users. If Fritz beats Freddy to the checkout button, there'll be a problem once Freddy goes to check out. This can easily be explained to the customer: he was too late. But what if both press the checkout button at the exact same millisecond and both sales happen?

ACID versus BASE

The BASE principles are somewhat contrived to fit *acid* and *base* from chemistry: an acid is a fluid with a low pH value. A base is the opposite and has a high pH value. We won't go into the chemistry details here, but figure 6.6 shows a mnemonic to those familiar with the chemistry equivalents of acid and base.

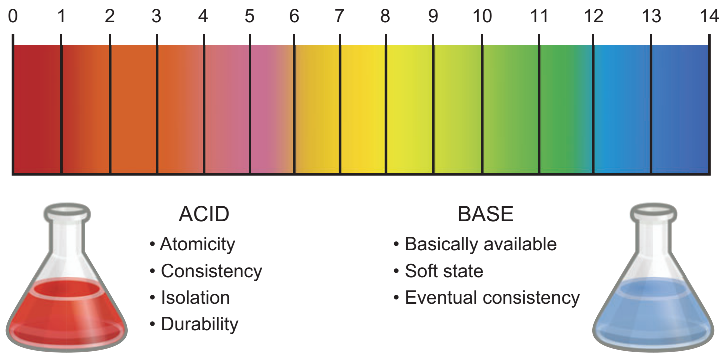


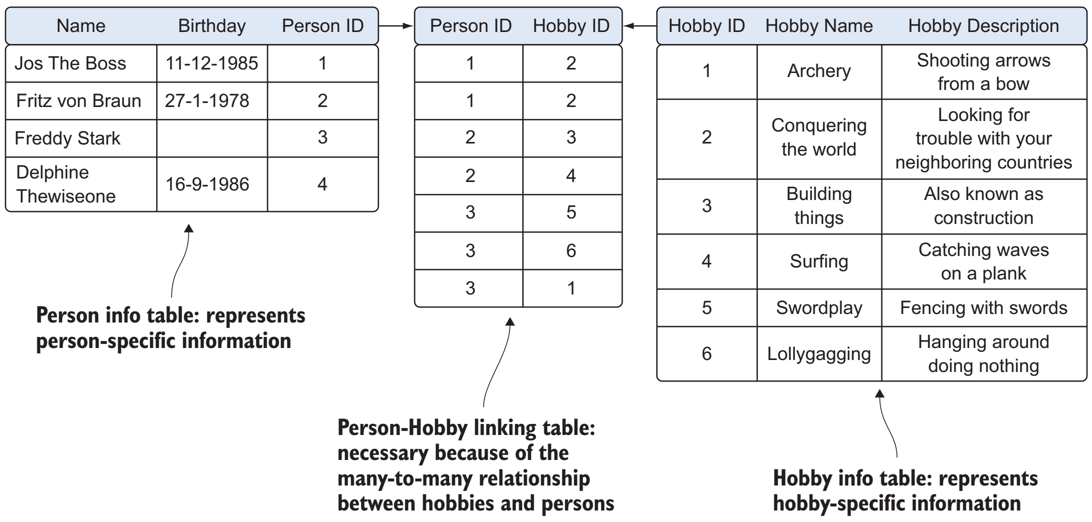
Figure 6.6 ACID versus BASE: traditional relational databases versus most NoSQL databases. The names are derived from the chemistry concept of the pH scale. A pH value below 7 is acidic; higher than 7 is a base. On this scale, your average surface water fluctuates between 6.5 and 8.5.

6.1.4 NoSQL database types

As you saw earlier, there are four big NoSQL types: key-value store, document store, column-oriented database, and graph database. Each type solves a problem that can't be solved with relational databases. Actual implementations are often combinations of these. OrientDB, for example, is a *multi-model database*, combining NoSQL types. OrientDB is a graph database where each node is a document.

Before going into the different NoSQL databases, let's look at relational databases so you have something to compare them to. In data modeling, many approaches are possible. Relational databases generally strive toward *normalization*: making sure every piece of data is stored only once. Normalization marks their structural setup. If, for instance, you want to store data about a person and their hobbies, you can do so with two tables: one about the person and one about their hobbies. As you can see in figure 6.7, an additional table is necessary to link hobbies to persons because of their *many-to-many relationship*: a person can have multiple hobbies and a hobby can have many persons practicing it.

A full-scale relational database can be made up of many entities and linking tables. Now that you have something to compare NoSQL to, let's look at the different types.



Name	Birthday	Person ID
Jos The Boss	11-12-1985	1
Fritz von Braun	27-1-1978	2
Freddy Stark		3
Delphine Thewiseone	16-9-1986	4

Person ID	Hobby ID
1	2
1	2
2	3
2	4
3	5
3	6
3	1

Hobby ID	Hobby Name	Hobby Description
1	Archery	Shooting arrows from a bow
2	Conquering the world	Looking for trouble with your neighboring countries
3	Building things	Also known as construction
4	Surfing	Catching waves on a plank
5	Swordplay	Fencing with swords
6	Lollygagging	Hanging around doing nothing

Person info table: represents person-specific information
Person-Hobby linking table: necessary because of the many-to-many relationship between hobbies and persons
Hobby info table: represents hobby-specific information

Figure 6.7 Relational databases strive toward normalization (making sure every piece of data is stored only once). Each table has unique identifiers (primary keys) that are used to model the relationship between the entities (tables), hence the term relational.

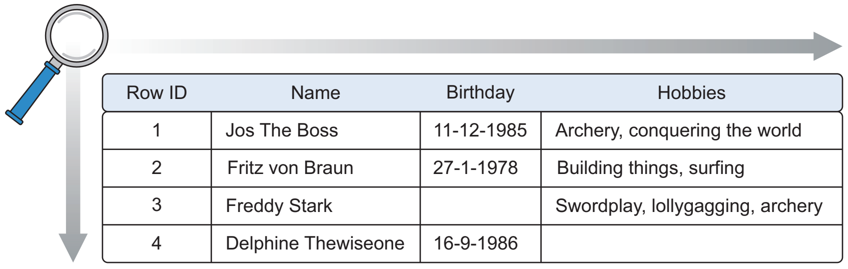
Column-oriented database

Traditional relational databases are row-oriented, with each row having a row id and each field within the row stored together in a table. Let's say, for example's sake, that no extra data about hobbies is stored and you have only a single table to describe people, as shown in figure 6.8. Notice how in this scenario you have slight denormalization because hobbies could be repeated. If the hobby information is a nice extra but not essential to your use case, adding it as a list within the Hobbies column is an acceptable approach. But if the information isn't important enough for a separate table, should it be stored at all?

Row ID	Name	Birthday	Hobbies
1	Jos The Boss	11-12-1985	Archery, conquering the world
2	Fritz von Braun	27-1-1978	Building things, surfing
3	Freddy Stark		Swordplay, lollygagging, archery
4	Delphine Thewiseone	16-9-1986	

Figure 6.8 Row-oriented database layout. Every entity (person) is represented by a single row, spread over multiple columns.

Every time you look up something in a row-oriented database, every row is scanned, regardless of which columns you require. Let's say you only want a list of birthdays in September. The database will scan the table from top to bottom and left to right, as shown in figure 6.9, eventually returning the list of birthdays.

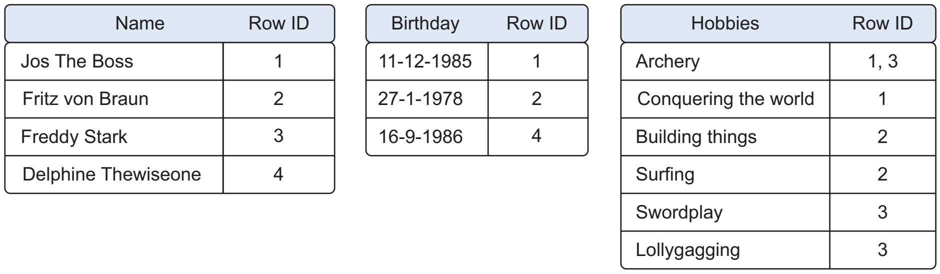


Row ID	Name	Birthday	Hobbies
1	Jos The Boss	11-12-1985	Archery, conquering the world
2	Fritz von Braun	27-1-1978	Building things, surfing
3	Freddy Stark		Swordplay, lollygagging, archery
4	Delphine Thewiseone	16-9-1986	

Figure 6.9 Row-oriented lookup: from top to bottom and for every entry, all columns are taken into memory

Indexing the data on certain columns can significantly improve lookup speed, but indexing every column brings extra overhead and the database is still scanning all the columns.

Column databases store each column separately, allowing for quicker scans when only a small number of columns is involved; see figure 6.10.



Name	Row ID
Jos The Boss	1
Fritz von Braun	2
Freddy Stark	3
Delphine Thewiseone	4

Birthday	Row ID
11-12-1985	1
27-1-1978	2
16-9-1986	4

Hobbies	Row ID
Archery	1, 3
Conquering the world	1
Building things	2
Surfing	2
Swordplay	3
Lollygagging	3

Figure 6.10 Column-oriented databases store each column separately with the related row numbers. Every entity (person) is divided over multiple tables.

This layout looks very similar to a row-oriented database with an index on every column. A database *index* is a data structure that allows for quick lookups on data at the cost of storage space and additional writes (index update). An index maps the row number to the data, whereas a column database maps the data to the row numbers; in that way counting becomes quicker, so it's easy to see how many people like archery, for instance. Storing the columns separately also allows for optimized compression because there's only one data type per table.

When should you use a row-oriented database and when should you use a column-oriented database? In a column-oriented database it's easy to add another column because none of the existing columns are affected by it. But adding an entire record requires adapting all tables. This makes the row-oriented database preferable over the column-oriented database for online transaction processing (OLTP), because this implies adding or changing records constantly. The column-oriented database shines when performing analytics and reporting: summing values and counting entries. A row-oriented database is often the operational database of choice for actual transactions (such as sales). Overnight batch jobs bring the column-oriented database up to date, supporting lightning-speed lookups and aggregations using MapReduce algorithms for reports. Examples of column-family stores are Apache HBase, Facebook's Cassandra, Hypertable, and the grandfather of wide-column stores, Google BigTable.

KEY-VALUE STORES

Key-value stores are the least complex of the NoSQL databases. They are, as the name suggests, a collection of key-value pairs, as shown in figure 6.11, and this simplicity makes them the most scalable of the NoSQL database types, capable of storing huge amounts of data.

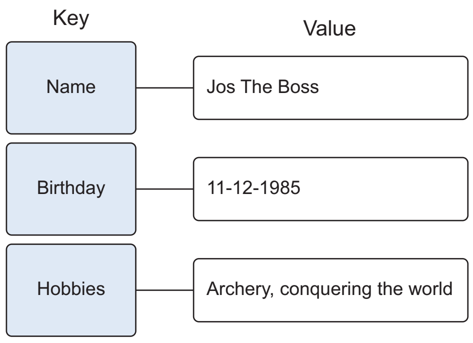


Figure 6.11 Key-value stores store everything as a key and a value.

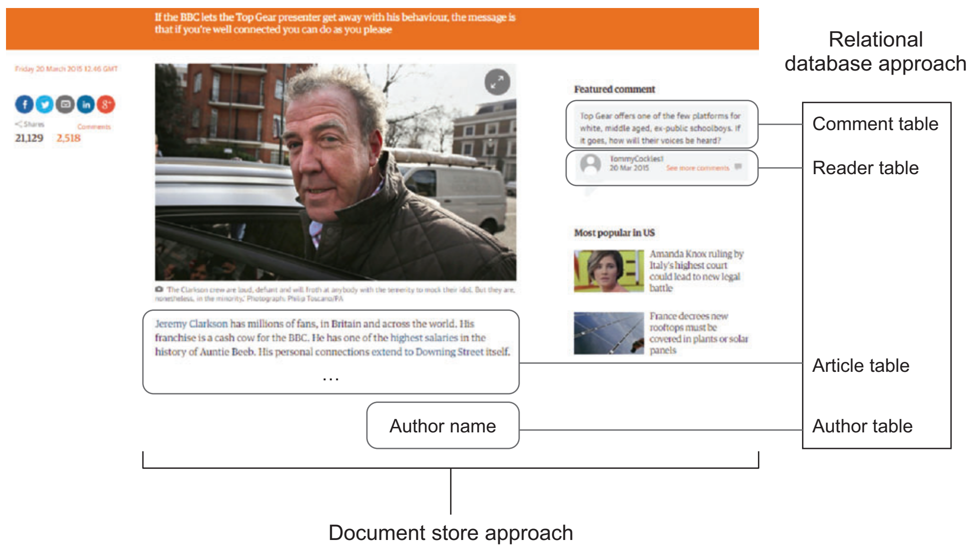
The value in a key-value store can be anything: a string, a number, but also an entire new set of key-value pairs encapsulated in an object. Figure 6.12 shows a slightly more complex key-value structure. Examples of key-value stores are Redis, Voldemort, Riak, and Amazon's Dynamo.

```
{"internal data":[{"entities":[
    {"customer":[
        {"id":1,"name":"Freddy"},
        {"id":2,"name":"Fritz"}
    ]},
    {"legal entities":[
        {"id":1,"company":"Maiton"}
    ]}]
},{"Products":[
    {"furniture":[
        {"id":1,"name":"Octopus Table","stock":1}
    ]
}]}]}
```

Figure 6.12 Key-value nested structure

Document stores

Document stores are one step up in complexity from key-value stores: a document store does assume a certain document structure that can be specified with a schema. Document stores appear the most natural among the NoSQL database types because they're designed to store everyday documents as is, and they allow for complex querying and calculations on this often already aggregated form of data. The way things are stored in a relational database makes sense from a normalization point of view: everything should be stored only once and connected via foreign keys. Document stores care little about normalization as long as the data is in a structure that makes sense. A relational data model doesn't always fit well with certain business cases. Newspapers or magazines, for example, contain articles. To store these in a relational database, you need to chop them up first: the article text goes in one table, the author and all the information about the author in another, and comments on the article when published on a website go in yet another. As shown in figure 6.13, a newspaper article



If the BBC lets the Top Gear presenter get away with his behaviour, the message is that if you're well connected you can do as you please

Friday 20 March 2015 12.46 GMT

Shares 21,129 Comments 2,518

'The Clarkson crew are loud, defiant and will froth at anybody with the temerity to mock their idol. But they are, nonetheless, in the minority.' Photograph: Philip Toscano/PA

Jeremy Clarkson has millions of fans, in Britain and across the world. His franchise is a cash cow for the BBC. He has one of the highest salaries in the history of Auntie Beeb. His personal connections extend to Downing Street itself.

...

Featured comment

Top Gear offers one of the few platforms for white, middle aged, ex-public schoolboys. If it goes, how will their voices be heard?

TommyCockles1 20 Mar 2015 See more comments

Most popular in US

Amanda Knox ruling by Italy's highest court could lead to new legal battle

France decrees new rooftops must be covered in plants or solar panels

```
{
    "articles": [
        {
            "title": "title of the article",
            "articleID": 1,
            "body": "body of the article",
            "author": "Isaac Asimov",
            "comments": [
                {
                    "username": "Fritz"
                    "join date": "1/4/2014"
                    "commentid": 1,
                    "body": "this is a great article",
                    "replies": [
                        {
                            "username": "Freddy",
                            "join date": "11/12/2013",
                            "commentid": 2,
                            "body": "seriously? it's rubbish"
                        }
                    ]
                },
                {
                    "username": "Stark",
                    "join date": "19/06/2011",
                    "commentid": 3,
                    "body": "I don't agree with the conclusion"
                }
            ]
        }
    ]
}
```

Figure 6.13 Document stores save documents as a whole, whereas an RDMS cuts up the article and saves it in several tables. The example was taken from the *Guardian* website.

can also be stored as a single entity; this lowers the cognitive burden of working with the data for those used to seeing articles all the time. Examples of document stores are MongoDB and CouchDB.

GRAPH DATABASES

The last big NoSQL database type is the most complex one, geared toward storing relations between entities in an efficient manner. When the data is highly interconnected, such as for social networks, scientific paper citations, or capital asset clusters, graph databases are the answer. Graph or network data has two main components:

- *Node*—The entities themselves. In a social network this could be people.
- *Edge*—The relationship between two entities. This relationship is represented by a line and has its own properties. An edge can have a direction, for example, if the arrow indicates who is whose boss.

Graphs can become incredibly complex given enough relation and entity types. Figure 6.14 already shows that complexity with only a limited number of entities. Graph databases like Neo4j also claim to uphold ACID, whereas document stores and key-value stores adhere to BASE.

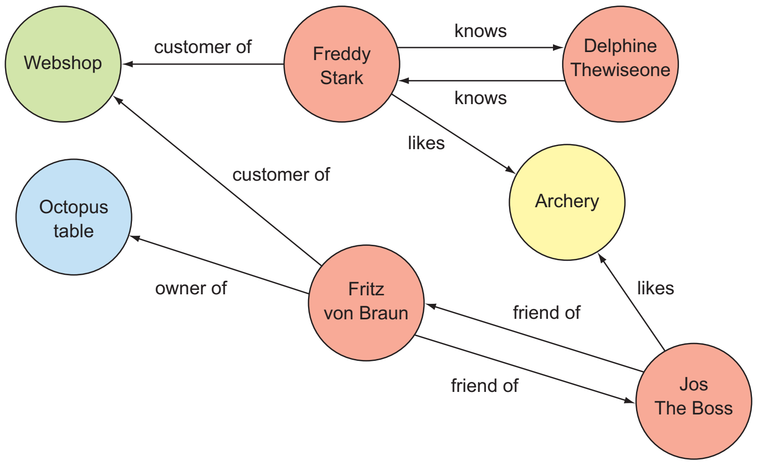
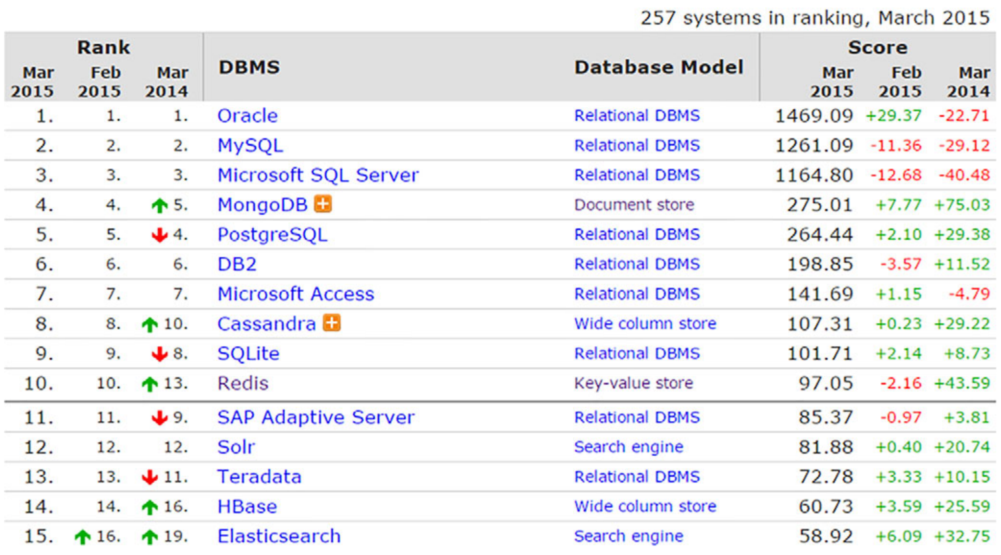


Figure 6.14 Graph data example with four entity types (person, hobby, company, and furniture) and their relations without extra edge or node information

The possibilities are endless, and because the world is becoming increasingly interconnected, graph databases are likely to win terrain over the other types, including the still-dominant relational database. A ranking of the most popular databases and how they're progressing can be found at http://db-engines.com/en/ranking.



257 systems in ranking, March 2015

Rank			DBMS	Database Model	Score		
Mar 2015	Feb 2015	Mar 2014			Mar 2015	Feb 2015	Mar 2014
1.	1.	1.	Oracle	Relational DBMS	1469.09	+29.37	-22.71
2.	2.	2.	MySQL	Relational DBMS	1261.09	-11.36	-29.12
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1164.80	-12.68	-40.48
4.	4.	↑ 5.	MongoDB	Document store	275.01	+7.77	+75.03
5.	5.	↓ 4.	PostgreSQL	Relational DBMS	264.44	+2.10	+29.38
6.	6.	6.	DB2	Relational DBMS	198.85	-3.57	+11.52
7.	7.	7.	Microsoft Access	Relational DBMS	141.69	+1.15	-4.79
8.	8.	↑ 10.	Cassandra	Wide column store	107.31	+0.23	+29.22
9.	9.	↓ 8.	SQLite	Relational DBMS	101.71	+2.14	+8.73
10.	10.	↑ 13.	Redis	Key-value store	97.05	-2.16	+43.59
11.	11.	↓ 9.	SAP Adaptive Server	Relational DBMS	85.37	-0.97	+3.81
12.	12.	12.	Solr	Search engine	81.88	+0.40	+20.74
13.	13.	↓ 11.	Teradata	Relational DBMS	72.78	+3.33	+10.15
14.	14.	↑ 16.	HBase	Wide column store	60.73	+3.59	+25.59
15.	↑ 16.	↑ 19.	Elasticsearch	Search engine	58.92	+6.09	+32.75

Figure 6.15 Top 15 databases ranked by popularity according to DB-Engines.com in March 2015

Figure 6.15 shows that with 9 entries, relational databases still dominate the top 15 at the time this book was written, and with the coming of NewSQL we can't count them out yet. Neo4j, the most popular graph database, can be found at position 23 at the time of writing, with Titan at position 53.

Now that you've seen each of the NoSQL database types, it's time to get your hands dirty with one of them.

6.2 Case study: What disease is that?

It has happened to many of us: you have sudden medical symptoms and the first thing you do is Google what disease the symptoms might indicate; then you decide whether it's worth seeing a doctor. A web search engine is okay for this, but a more dedicated database would be better. Databases like this exist and are fairly advanced; they can be almost a virtual version of Dr. House, a brilliant diagnostician in the TV series *House M.D.* But they're built upon well-protected data and not all of it is accessible by the public. Also, although big pharmaceutical companies and advanced hospitals have access to these virtual doctors, many general practitioners are still stuck with only their books. This information and resource asymmetry is not only sad and dangerous, it needn't be there at all. If a simple, disease-specific search engine were used by all general practitioners in the world, many medical mistakes could be avoided.

In this case study, you'll learn how to build such a search engine here, albeit using only a fraction of the medical data that is freely accessible. To tackle the problem, you'll use a modern NoSQL database called Elasticsearch to store the data, and the data science process to work with the data and turn it into a resource that's fast and easy to search. Here's how you'll apply the process:

1. *Setting the research goal.*
2. *Data collection*—You'll get your data from Wikipedia. There are more sources out there, but for demonstration purposes a single one will do.
3. *Data preparation*—The Wikipedia data might not be perfect in its current format. You'll apply a few techniques to change this.
4. *Data exploration*—Your use case is special in that step 4 of the data science process is also the desired end result: you want your data to become easy to explore.
5. *Data modeling*—No real data modeling is applied in this chapter. Document-term matrices that are used for search are often the starting point for advanced topic modeling. We won't go into that here.
6. *Presenting results*—To make data searchable, you'd need a user interface such as a website where people can query and retrieve disease information. In this chapter you won't go so far as to build an actual interface. Your secondary goal: profiling a disease category by its keywords; you'll reach this stage of the data science process because you'll present it as a word cloud, such as the one in figure 6.16.

Figure 6.16 A sample word cloud on non-weighted diabetes keywords

To follow along with the code, you'll need these items:

- A Python session with the elasticsearch-py and Wikipedia libraries installed (`pip install elasticsearch` and `pip install wikipedia`)
- A locally set up Elasticsearch instance; see appendix A for installation instructions
- The IPython library

NOTE The code for this chapter is available to download from the Manning website for this book at https://manning.com/books/introducing-data-science and is in IPython format.

Elasticsearch: the open source search engine/NoSQL database

To tackle the problem at hand, diagnosing a disease, the NoSQL database you'll use is Elasticsearch. Like MongoDB, Elasticsearch is a document store. But unlike MongoDB, Elasticsearch is a search engine. Whereas MongoDB is great at performing complex calculations and MapReduce jobs, Elasticsearch's main purpose is full-text search. Elasticsearch will do basic calculations on indexed numerical data such as summing, counts, median, mean, standard deviation, and so on, but in essence it remains a search engine.

Elasticsearch is built on top of Apache Lucene, the Apache search engine created in 1999. Lucene is notoriously hard to handle and is more a building block for more user-friendly applications than an end–to–end solution in itself. But Lucene is an enormously powerful search engine, and Apache Solr followed in 2004, opening for public use in 2006. Solr (an open source, enterprise search platform) is built on top of Apache Lucene and is at this moment still the most versatile and popular open source search engine. Solr is a great platform and worth investigating if you get involved in a project requiring a search engine. In 2010 Elasticsearch emerged, quickly gaining in popularity. Although Solr can still be difficult to set up and configure, even for small projects, Elasticsearch couldn't be easier. Solr still has an advantage in the number of possible plugins expanding its core functionality, but Elasticsearch is quickly catching up and today its capabilities are of comparable quality.

6.2.1 Step 1: Setting the research goal

Can you diagnose a disease by the end of this chapter, using nothing but your own home computer and the free software and data out there? Knowing what you want to do and how to do it is the first step in the data science process, as shown in figure 6.17.

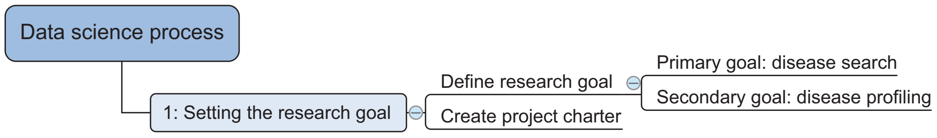

Data science process
1: Setting the research goal
Define research goal
Create project charter
Primary goal: disease search
Secondary goal: disease profiling

Figure 6.17 Step 1 in the data science process: setting the research goal

- Your primary goal is to set up a disease search engine that would help general practitioners in diagnosing diseases.
- Your secondary goal is to profile a disease: What keywords distinguish it from other diseases?

This secondary goal is useful for educational purposes or as input to more advanced uses such as detecting spreading epidemics by tapping into social media. With your research goal and a plan of action defined, let's move on to the data retrieval step.

6.2.2 Steps 2 and 3: Data retrieval and preparation

Data retrieval and data preparation are two distinct steps in the data science process, and even though this remains true for the case study, we'll explore both in the same section. This way you can avoid setting up local intermedia storage and immediately do data preparation while the data is being retrieved. Let's look at where we are in the data science process (see figure 6.18).

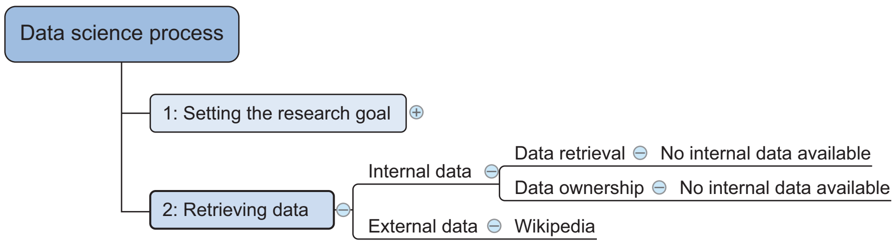


Figure 6.18 Data science process step 2: data retrieval. In this case there's no internal data; all data will be fetched from Wikipedia.

As shown in figure 6.18 you have two possible sources: internal data and external data.

- *Internal data*—You have no disease information lying around. If you currently work for a pharmaceutical company or a hospital, you might be luckier.
- *External data*—All you can use for this case is external data. You have several possibilities, but you'll go with Wikipedia.

When you pull the data from Wikipedia, you'll need to store it in your local Elasticsearch index, but before you do that you'll need to prepare the data. Once data has entered the Elasticsearch index, it can't be altered; all you can do then is query it. Look at the data preparation overview in figure 6.19.

As shown in figure 6.19 there are three distinct categories of data preparation to consider:

- *Data cleansing*—The data you'll pull from Wikipedia can be incomplete or erroneous. Data entry errors and spelling mistakes are possible—even false information isn't excluded. Luckily, you don't need the list of diseases to be exhaustive, and you can handle spelling mistakes at search time; more on that later. Thanks to the Wikipedia Python library, the textual data you'll receive is fairly clean already. If you were to scrape it manually, you'd need to add HTML cleaning, removing all HTML tags. The truth of the matter is full-text search tends to be fairly robust toward common errors such as incorrect values. Even if you dumped in HTML tags on purpose, they'd be unlikely to influence the results; the HTML tags are too different from normal language to interfere.

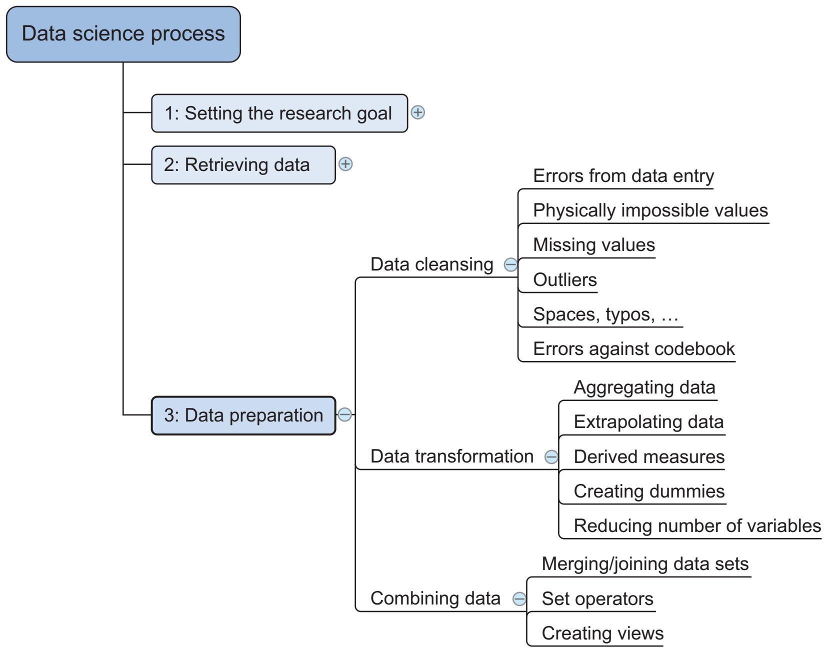


Figure 6.19 Data science process step 3: data preparation

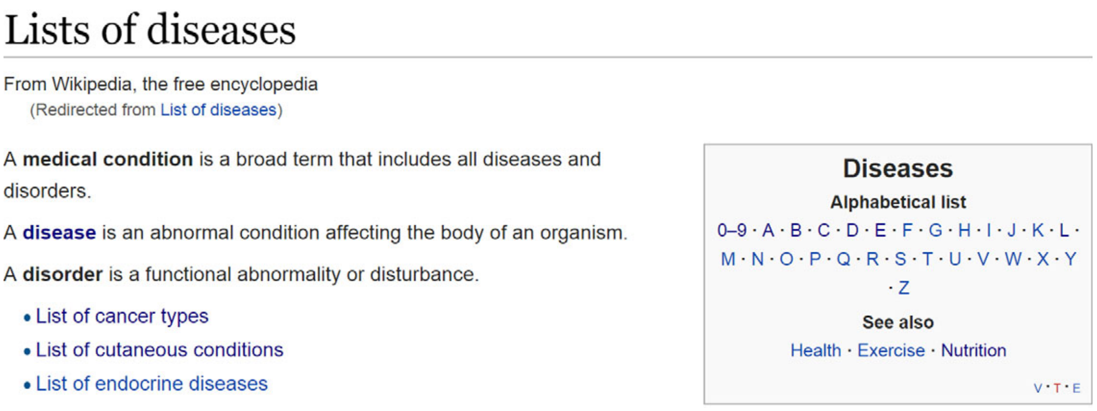
- *Data transformation*—You don't need to transform the data much at this point; you want to search it as is. But you'll make the distinction between page title, disease name, and page body. This distinction is almost mandatory for search result interpretation.
- *Combining data*—All the data is drawn from a single source in this case, so you have no real need to combine data. A possible extension to this exercise would be to get disease data from another source and match the diseases. This is no trivial task because no unique identifier is present and the names are often slightly different.

You can do data cleansing at only two stages: when using the Python program that connects Wikipedia to Elasticsearch and when running the Elasticsearch internal indexing system:

- *Python*—Here you define what data you'll allow to be stored by your document store, but you won't clean the data or transform the data at this stage, because Elasticsearch is better at it for less effort.
- *Elasticsearch*—Elasticsearch will handle the data manipulation (creating the index) under the hood. You can still influence this process, and you'll do so more explicitly later in this chapter.

Now that you have an overview of the steps to come, let's get to work. If you followed the instructions in the appendix, you should now have a local instance of Elasticsearch up and running. First comes data retrieval: you need information on the different diseases. You have several ways to get that kind of data. You could ask companies for their data or get data from Freebase or other open and free data sources. Acquiring your data can be a challenge, but for this example you'll be pulling it from Wikipedia. This is a bit ironic because searches on the Wikipedia website itself are handled by Elasticsearch. Wikipedia used to have its own system build on top of Apache Lucene, but it became unmaintainable, and as of January 2014 Wikipedia began using Elasticsearch instead.

Wikipedia has a Lists of diseases page, as shown in figure 6.20. From here you can borrow the data from the alphabetical lists.



Lists of diseases

From Wikipedia, the free encyclopedia
(Redirected from List of diseases)

A **medical condition** is a broad term that includes all diseases and disorders.

A **disease** is an abnormal condition affecting the body of an organism.

A **disorder** is a functional abnormality or disturbance.

- List of cancer types
- List of cutaneous conditions
- List of endocrine diseases

Diseases

Alphabetical list

0–9 · A · B · C · D · E · F · G · H · I · J · K · L · M · N · O · P · Q · R · S · T · U · V · W · X · Y · Z

See also

Health · Exercise · Nutrition

V · T · E

Figure 6.20 Wikipedia's Lists of diseases page, the starting point for your data retrieval

You know what data you want; now go grab it. You could download the entire Wikipedia data dump. If you want to, you can download it to http://meta.wikimedia.org/wiki/Data_dump_torrents#enwiki.

Of course, if you were to index the entire Wikipedia, the index would end up requiring about 40 GB of storage. Feel free to use this solution, but for the sake of preserving storage and bandwidth, we'll limit ourselves in this book to pulling only the data we intend to use. Another option is scraping the pages you require. Like Google, you can make a program crawl through the pages and retrieve the entire rendered HTML. This would do the trick, but you'd end up with the actual HTML, so you'd need to clean that up before indexing it. Also, unless you're Google, websites aren't too fond of crawlers scraping their web pages. This creates an unnecessarily high amount of traffic, and if enough people send crawlers, it can bring the HTTP server to its knees, spoiling the fun for everyone. Sending billions of requests at the same time is also one of the ways denial of service (DoA) attacks are performed. If you do need to scrape a website, script in a time gap between each page request. This way, your scraper more closely mimics the behavior of a regular website visitor and you won't blow up their servers.

Luckily, the creators of Wikipedia are smart enough to know that this is exactly what would happen with all this information open to everyone. They've put an API in place from which you can safely draw your information. You can read more about it at http://www.mediawiki.org/wiki/API:Main_page.

You'll draw from the API. And Python wouldn't be Python if it didn't already have a library to do the job. There are several actually, but the easiest one will suffice for your needs: Wikipedia.

Activate your Python virtual environment and install all the libraries you'll need for the rest of the book:

```
pip install wikipedia
pip install Elasticsearch
```

You'll use Wikipedia to tap into Wikipedia. Elasticsearch is the main Elasticsearch Python library; with it you can communicate with your database.

Open your favorite Python interpreter and import the necessary libraries:

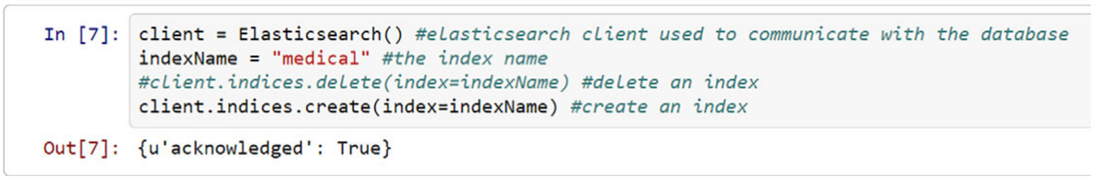
```
from elasticsearch import Elasticsearch
import wikipedia
```

You're going to draw data from the Wikipedia API and at the same time index on your local Elasticsearch instance, so first you need to prepare it for data acceptance.

```
client = Elasticsearch()                         <- Elasticsearch client used to communicate with database
indexName = "medical"                            <- Index name
client.indices.create(index=indexName)           <- Create index
```

The first thing you need is a client. `Elasticsearch()` can be initialized with an address but the default is localhost:9200. `Elasticsearch()` and `Elasticsearch ('localhost:9200')` are thus the same thing: your client is connected to your local Elasticsearch node. Then you create an index named `"medical"`. If all goes well, you should see an `"acknowledged:true"` reply, as shown in figure 6.21.

Elasticsearch claims to be schema-less, meaning you can use Elasticsearch without defining a database schema and without telling Elasticsearch what kind of data it



```
In [7]: client = Elasticsearch() #elasticsearch client used to communicate with the database
        indexName = "medical" #the index name
        #client.indices.delete(index=indexName) #delete an index
        client.indices.create(index=indexName) #create an index

Out[7]: {u'acknowledged': True}
```

Figure 6.21 Creating an Elasticsearch index with Python-Elasticsearch

needs to expect. Although this is true for simple cases, you can't avoid having a schema in the long run, so let's create one, as shown in the following listing.

Listing 6.1 Adding a mapping to the document type

```
diseaseMapping = {
        'properties': {
            'name': {'type': 'string'},
            'title': {'type': 'string'},
            'fulltext': {'type': 'string'}
        }
    }
client.indices.put_mapping(index=indexName,
doc_type='diseases',body=diseaseMapping )
```

Defining a mapping and attributing it to the disease doc type.

The "diseases" doc type is updated with a mapping. Now we define the data it should expect.

This way you tell Elasticsearch that your index will have a document type called `"disease"`, and you supply it with the field type for each of the fields. You have three fields in a disease document: `name`, `title`, and `fulltext`, all of them of type `string`. If you hadn't supplied the mapping, Elasticsearch would have guessed their types by looking at the first entry it received. If it didn't recognize the field to be `boolean`, `double`, `float`, `long`, `integer`, or `date`, it would set it to `string`. In this case, you didn't need to manually specify the mapping.

Now let's move on to Wikipedia. The first thing you want to do is fetch the List of diseases page, because this is your entry point for further exploration:

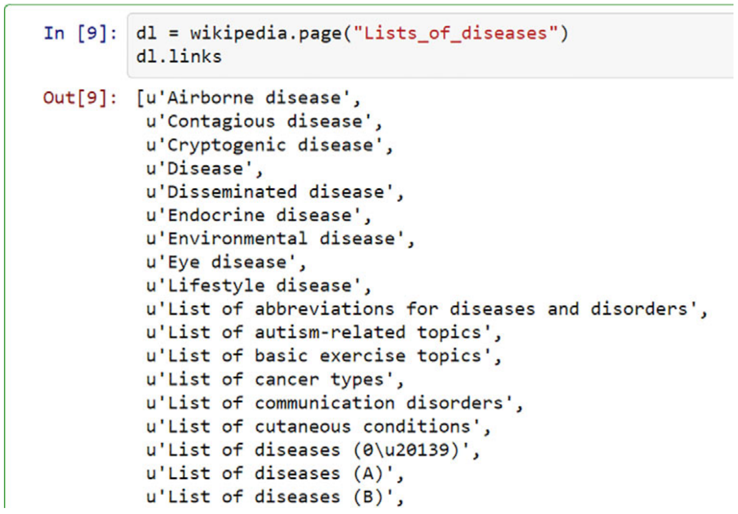
```
dl = wikipedia.page("Lists_of_diseases")
```

You now have your first page, but you're more interested in the listing pages because they contain links to the diseases. Check out the links:

```
dl.links
```

The List of diseases page comes with more links than you'll use. Figure 6.22 shows the alphabetical lists starting at the sixteenth link.

```
dl = wikipedia.page("Lists_of_diseases")
dl.links
```


```
In [9]: dl = wikipedia.page("Lists_of_diseases")
        dl.links

Out[9]: [u'Airborne disease',
         u'Contagious disease',
         u'Cryptogenic disease',
         u'Disease',
         u'Disseminated disease',
         u'Endocrine disease',
         u'Environmental disease',
         u'Eye disease',
         u'Lifestyle disease',
         u'List of abbreviations for diseases and disorders',
         u'List of autism-related topics',
         u'List of basic exercise topics',
         u'List of cancer types',
         u'List of communication disorders',
         u'List of cutaneous conditions',
         u'List of diseases (0\u20139)',
         u'List of diseases (A)',
         u'List of diseases (B)',
```

Figure 6.22 Links on the Wikipedia page Lists of diseases. It has more links than you'll need.

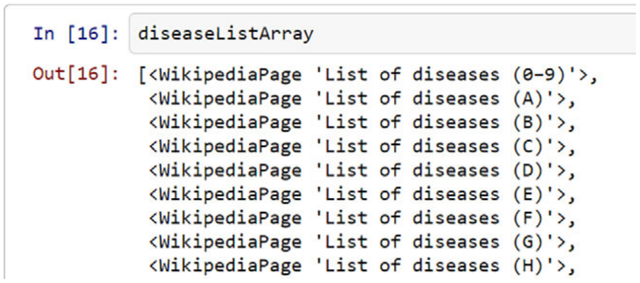
This page has a considerable array of links, but only the alphabetic lists interest you, so keep only those:

```
diseaseListArray = []
for link in dl.links[15:42]:
    try:
        diseaseListArray.append(wikipedia.page(link))
    except Exception,e:
        print str(e)
```

You've probably noticed that the subset is hardcoded, because you know they're the 16th to 43rd entries in the array. If Wikipedia were to add even a single link before the ones you're interested in, it would throw off the results. A better practice would be to use regular expressions for this task. For exploration purposes, hardcoding the entry numbers is fine, but if regular expressions are second nature to you or you intend to turn this code into a batch job, regular expressions are recommended. You can find more information on them at https://docs.python.org/2/howto/regex.html.

One possibility for a regex version would be the following code snippet.

```
diseaseListArray = []
check = re.compile("List of diseases*")
for link in dl.links:
    if check.match(link):
        try:
            diseaseListArray.append(wikipedia.page(link))
        except Exception,e:
            print str(e)
```



```
In [16]: diseaseListArray

Out[16]: [<WikipediaPage 'List of diseases (0-9)'>,
          <WikipediaPage 'List of diseases (A)'>,
          <WikipediaPage 'List of diseases (B)'>,
          <WikipediaPage 'List of diseases (C)'>,
          <WikipediaPage 'List of diseases (D)'>,
          <WikipediaPage 'List of diseases (E)'>,
          <WikipediaPage 'List of diseases (F)'>,
          <WikipediaPage 'List of diseases (G)'>,
          <WikipediaPage 'List of diseases (H)'>,
```

Figure 6.23 First Wikipedia disease list, "list of diseases (0-9)"

Figure 6.23 shows the first entries of what you're after: the diseases themselves.

```
diseaseListArray[0].links
```

It's time to index the diseases. Once they're indexed, both data entry and data preparation are effectively over, as shown in the following listing.

Listing 6.2 Indexing diseases from Wikipedia

```
checkList = [["0","1","2","3","4","5","6","7","8","9"],
["A"],["B"],["C"],["D"],["E"],["F"],["G"],["H"],
["I"],["J"],["K"],["L"],["M"],["N"],["O"],["P"],
["Q"],["R"],["S"],["T"],["U"],["V"],["W"],["X"],["Y"],["Z"]]
docType = 'diseases'
for diseaselistNumber, diseaselist in enumerate(diseaseListArray):
    for disease in diseaselist.links:
        try:
            if disease[0] in checkList[diseaselistNumber]
and disease[0:3] !="List":
                currentPage = wikipedia.page(disease)
                client.index(index=indexName,
doc_type=docType,id = disease, body={"name": disease,
     "title":currentPage.title ,
"fulltext":currentPage.content})
        except Exception,e:
            print str(e)
```

The checklist is an array containing an array of allowed first characters. If a disease doesn't comply, skip it.

Document type you'll index.

Looping through disease lists.

Looping through lists of links for every disease list.

First check if it's a disease, then index it.

Because each of the list pages will have links you don't need, check to see if an entry is a disease. You indicate for each list what character the disease starts with, so you check for this. Additionally you exclude the links starting with "list" because these will pop up once you get to the L list of diseases. The check is rather naïve, but the cost of having a few unwanted entries is rather low because the search algorithms will exclude irrelevant results once you start querying. For each disease you index the disease name and the full text of the page. The name is also used as its index ID; this is useful for several advanced Elasticsearch features but also for quick lookup in the browser. For example, try this URL in your browser: http://localhost:9200/medical/diseases/11%20beta%20hydroxylase%20deficiency. The title is indexed separately; in most cases the link name and the page title will be identical and sometimes the title will contain an alternative name for the disease.

With at least a few diseases indexed it's possible to make use of the Elasticsearch URI for simple lookups. Have a look at a full body search for the word *headache* in figure 6.24. You can already do this while indexing; Elasticsearch can update an index and return queries for it at the same time.

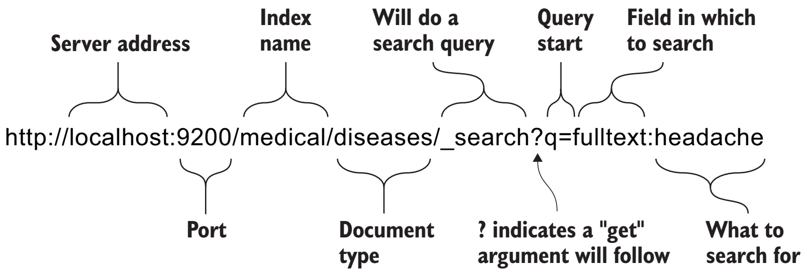
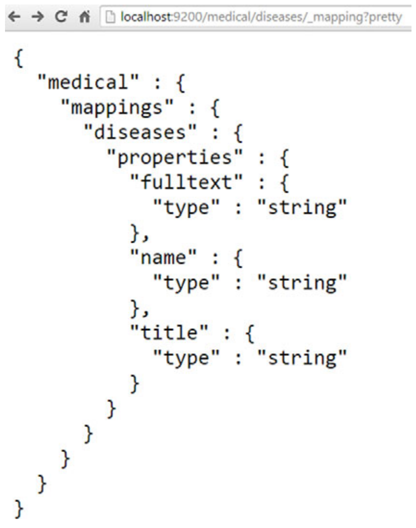


Figure 6.24 The Elasticsearch URL example buildup

If you don't query the index, you can still get a few results without knowing anything about the index. Specifying http://localhost:9200/medical/diseases/_search will return the first five results. For a more structured view on the data you can ask for the mapping of this document type at http://localhost:9200/medical/diseases/_mapping?pretty. The pretty `get` argument shows the returned JSON in a more readable format, as can be seen in figure 6.25. The mapping does appear to be the way you specified it: all fields are type `string`.

The Elasticsearch URL is certainly useful, yet it won't suffice for your needs. You still have diseases to diagnose, and for this you'll send POST requests to Elasticsearch via your Elasticsearch Python library.

With data retrieval and preparation accomplished, you can move on to exploring your data.


localhost:9200/medical/diseases/_mapping?pretty

```
{
  "medical" : {
    "mappings" : {
      "diseases" : {
        "properties" : {
          "fulltext" : {
            "type" : "string"
          },
          "name" : {
            "type" : "string"
          },
          "title" : {
            "type" : "string"
          }
        }
      }
    }
  }
}
```

Figure 6.25 Diseases document type mapping via Elasticsearch URL

6.2.3 Step 4: Data exploration

It's not lupus. It's never lupus!

—Dr. House of *House M.D.*

Data exploration is what marks this case study, because the primary goal of the project (disease diagnostics) is a specific way of exploring the data by querying for disease symptoms. Figure 6.26 shows several data exploration techniques, but in this case it's non-graphical: interpreting text search query results.

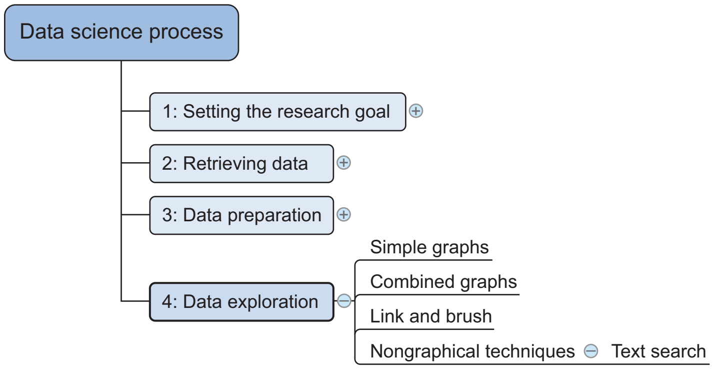


Figure 6.26 Data science process step 4: data exploration

The moment of truth is here: can you find certain diseases by feeding your search engine their symptoms? Let's first make sure you have the basics up and running. Import the Elasticsearch library and define global search settings:

```
from elasticsearch import Elasticsearch
client = Elasticsearch()
indexName = "medical"
docType="diseases"
searchFrom = 0
searchSize= 3
```

You'll return only the first three results; the default is five.

Elasticsearch has an elaborate JSON query language; every search is a POST request to the server and will be answered with a JSON answer. Roughly, the language consists of three big parts: queries, filters, and aggregations. A *query* takes in search keywords and puts them through one or more analyzers before the words are looked up in the index. We'll get deeper into analyzers a bit later in this chapter. A *filter* takes keywords like a query does but doesn't try to analyze what you give it; it filters on the conditions we provide. Filters are thus less complex but many times more efficient because they're also temporarily stored within Elasticsearch in case you use the same filter twice. *Aggregations* can be compared to the SQL group; buckets of words will be created, and for each bucket relevant statistics can be calculated. Each of these three compartments has loads of options and features, making elaborating on the entire language here impossible. Luckily, there's no need to go into the complexity that Elasticsearch queries can represent. We'll use the "Query string query language," a way to query the data that closely resembles the Google search query language. If, for instance, you want a search term to be mandatory, you add a plus (+) sign; if you want to exclude the search term, you use a minus (-) sign. Querying Elasticsearch isn't recommended because it decreases performance; the search engine first needs to translate the query string into its native JSON query language. But for your purposes it will work nicely; also, performance won't be a factor on the several thousand records you have in your index. Now it's time to query your disease data.

PROJECT PRIMARY OBJECTIVE: DIAGNOSING A DISEASE BY ITS SYMPTOMS

If you ever saw the popular television series *House M.D.*, the sentence "It's never lupus" may sound familiar. Lupus is a type of autoimmune disease, where the body's immune system attacks healthy parts of the body. Let's see what symptoms your search engine would need to determine that you're looking for lupus.

Start off with three symptoms: fatigue, fever, and joint pain. Your imaginary patient has all three of them (and more), so make them all mandatory by adding a plus sign before each one:

Listing 6.3 "simple query string" Elasticsearch query with three mandatory keywords

The dictionary named searchBody contains the search request information we'll send.

We want the name field in our results.

The query part. Other things are possible here, like aggregations. More on that later.

A simple query string is a type of query that takes input in much the same way the Google homepage would.

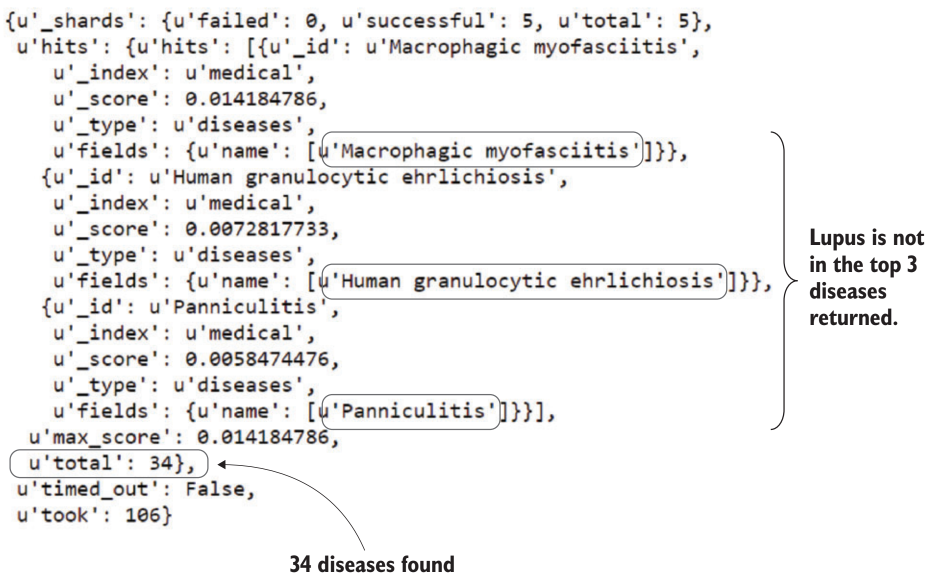
These fields are the fields in which it needs to search. They are not to be confused with the fields it has to return in the search results (specified in the second code line above).

```
searchBody={
"fields":["name"],
"query":{
    "simple_query_string" : {
        "query": '+fatigue+fever+"joint pain"',
        "fields": ["fulltext","title^5","name^10"]
        }
    }
}
client.search(index=indexName,doc_type=docType, body=searchBody, from_ =
     searchFrom, size=searchSize)
```

Like a query on Google the + sign indicates the term is mandatory. Encapsulating two or more words in quotes signals you want to find them exactly like this.

The search is executed. Variables indexName, docType, searchFrom, and searchSize were declared earlier: indexName = "medical" , docType="diseases" , searchFrom = 0 , searchSize = 3.

In searchBody, which has a JSON structure, you specify the fields you'd like to see returned, in this case the name of the disease should suffice. You use the query string syntax to search in all the indexed fields: fulltext, title, and name. By adding ^ you can give each field a weight. If a symptom occurs in the title, it's five times more important than in the open text; if it occurs in the name itself, it's considered ten times as important. Notice how "joint pain" is wrapped in a pair of quotation marks. If you didn't have the "" signs, *joint* and *pain* would have been considered as two separate keywords rather than a single phrase. In Elasticsearch this is called *phrase matching*. Let's look at the results in figure 6.27.

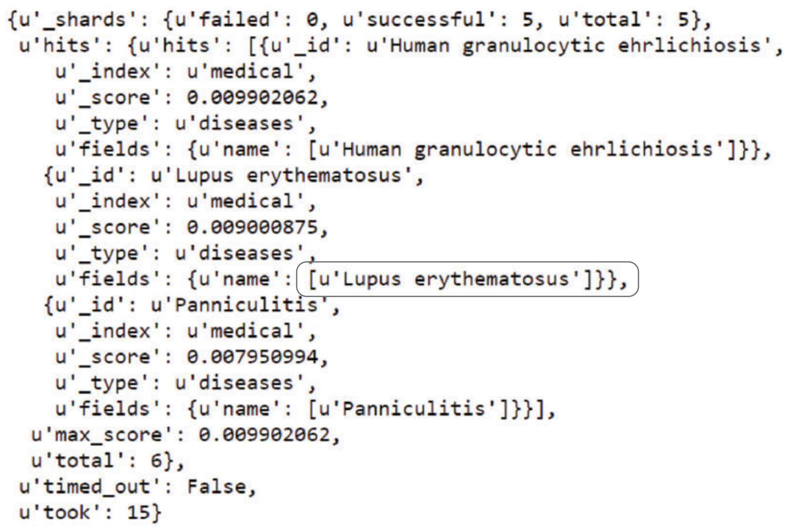


```
{u'_shards': {u'failed': 0, u'successful': 5, u'total': 5},
 u'hits': {u'hits': [{u'_id': u'Macrophagic myofasciitis',
    u'_index': u'medical',
    u'_score': 0.014184786,
    u'_type': u'diseases',
    u'fields': {u'name': [u'Macrophagic myofasciitis']}},
   {u'_id': u'Human granulocytic ehrlichiosis',
    u'_index': u'medical',
    u'_score': 0.0072817733,
    u'_type': u'diseases',
    u'fields': {u'name': [u'Human granulocytic ehrlichiosis']}},
   {u'_id': u'Panniculitis',
    u'_index': u'medical',
    u'_score': 0.0058474476,
    u'_type': u'diseases',
    u'fields': {u'name': [u'Panniculitis']}}],
  u'max_score': 0.014184786,
  u'total': 34},
 u'timed_out': False,
 u'took': 106}
```

Figure 6.27 Lupus first search with 34 results

Figure 6.27 shows the top three results returned out of 34 matching diseases. The results are sorted by their matching score, the variable _score. The matching score is no simple thing to explain; it takes into consideration how well the disease matches your query and how many times a keyword was found, the weights you gave, and so on. Currently, lupus doesn't even show up in the top three results. Luckily for you, lupus has another distinct symptom: a rash. The rash doesn't always show up on the person's face, but it does happen and this is where lupus got its name: the face rash makes people vaguely resemble a wolf. Your patient has a rash but not the signature rash on the face, so add "rash" to the symptoms without mentioning the face.

```
"query": '+fatigue+fever+"joint pain"+rash',
```



```
{u'_shards': {u'failed': 0, u'successful': 5, u'total': 5},
 u'hits': {u'hits': [{u'_id': u'Human granulocytic ehrlichiosis',
    u'_index': u'medical',
    u'_score': 0.009902062,
    u'_type': u'diseases',
    u'fields': {u'name': [u'Human granulocytic ehrlichiosis']}},
   {u'_id': u'Lupus erythematosus',
    u'_index': u'medical',
    u'_score': 0.009000875,
    u'_type': u'diseases',
    u'fields': {u'name': [u'Lupus erythematosus']}},
   {u'_id': u'Panniculitis',
    u'_index': u'medical',
    u'_score': 0.007950994,
    u'_type': u'diseases',
    u'fields': {u'name': [u'Panniculitis']}}],
  u'max_score': 0.009902062,
  u'total': 6},
 u'timed_out': False,
 u'took': 15}
```

Figure 6.28 Lupus second search attempt with six results and lupus in the top three

The results of the new search are shown in figure 6.28.

Now the results have been narrowed down to six and lupus is in the top three. At this point, the search engine says *Human Granulocytic Ehrlichiosis* (HGE) is more likely. HGE is a disease spread by ticks, like the infamous Lyme disease. By now a capable doctor would have already figured out which disease plagues your patient, because in determining diseases many factors are at play, more than you can feed into your humble search engine. For instance, the rash occurs only in 10% of HGE and in 50% of lupus patients. Lupus emerges slowly, whereas HGE is set off by a tick bite. Advanced machine-learning databases fed with all this information in a more structured way could make a diagnosis with far greater certainty. Given that you need to make do with the Wikipedia pages, you need another symptom to confirm that it's lupus. The patient experiences chest pain, so add this to the list.

```
"query": '+fatigue+fever+"joint pain"+rash+"chest pain"',
```

The result is shown in figure 6.29.

Seems like it's lupus. It took a while to get to this conclusion, but you got there. Of course, you were limited in the way you presented Elasticsearch with the symptoms. You used only either single terms ("fatigue") or literal phrases ("joint pain"). This worked out for this example, but Elasticsearch is more flexible than this. It can take regular expressions and do a fuzzy search, but that's beyond the scope of this book, although a few examples are included in the downloadable code.

```
{u'_shards': {u'failed': 0, u'successful': 5, u'total': 5},
 u'hits': {u'hits': [{u'_id': u'Lupus erythematosus',
    u'_index': u'medical',
    u'_score': 0.010452312,
    u'_type': u'diseases',
    u'fields': {u'name': [u'Lupus erythematosus']}}],
  u'max_score': 0.010452312,
  u'total': 1},
 u'timed_out': False,
 u'took': 11}
```

Figure 6.29 Lupus third search: with enough symptoms to determine it must be lupus

HANDLING SPELLING MISTAKES: DAMERAU-LEVENSHTEIN

Say someone typed "lupsu" instead of "lupus." Spelling mistakes happen all the time and in all types of human-crafted documents. To deal with this data scientists often use Damerau-Levenshtein. The Damerau-Levenshtein distance between two strings is the number of operations required to turn one string into the other. Four operations are allowed to calculate the distance:

- *Deletion*—Delete a character from the string.
- *Insertion*—Add a character to the string.
- *Substitution*—Substitute one character for another. Without the substitution counted as one operation, changing one character into another would take two operations: one deletion and one insertion.
- *Transposition of two adjacent characters*—Swap two adjacent characters.

This last operation (transposition) is what makes the difference between traditional Levenshtein distance and the Damerau-Levenshtein distance. It's this last operation that makes our dyslexic spelling mistake fall within acceptable limits. Damerau-Levenshtein is forgiving of these transposition mistakes, which makes it great for search engines, but it's also used for other things such as calculating the differences between DNA strings.

Figure 6.30 shows how the transformation from "lupsu" to "lupus" is performed with a single transposition.

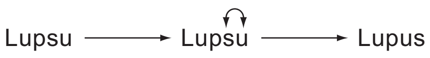


Figure 6.30 Adjacent character transposition is one of the operations in Damerau-Levenshtein distance. The other three are insertion, deletion, and substitution.

With just this you've achieved your first objective: *diagnosing a disease*. But let's not forget about your secondary project objective: *disease profiling*.

PROJECT SECONDARY OBJECTIVE: DISEASE PROFILING

What you want is a list of keywords fitting your selected disease. For this you'll use the significant terms aggregation. The score calculation to determine which words are significant is once again a combination of factors, but it roughly boils down to a comparison of the number of times a term is found in the result set as opposed to all the other documents. This way Elasticsearch profiles your result set by supplying the keywords that distinguish it from the other data. Let's do that on diabetes, a common disease that can take many forms:

Listing 6.4 Significant terms Elasticsearch query for "diabetes"

```
searchBody={
"fields":["name"],
"query":{
    "filtered" : {
        "filter": {
            'term': {'name':'diabetes'}
        }
    }
},
"aggregations" : {
        "DiseaseKeywords" : {
            "significant_terms" : { "field" : "fulltext", "size":30 }
        }
    }
}
client.search(index=indexName,doc_type=docType,
body=searchBody, from_ = searchFrom, size=searchSize)
```

The dictionary named searchBody contains the search request information we'll send.

We want the name field in our results.

The query part.

A filtered query has two possible components: a query and a filter. The query performs a search while the filter matches exact values only and is therefore way more efficient but restrictive.

The filter part of the filtered query. A query part isn't mandatory; a filter is sufficient.

We want to filter the name field and keep only if it contains the term diabetes.

An aggregation can generally be compared to a group by in SQL. It's mostly used to summarize values of a numeric variable over the distinct values within one or more variables.

DiseaseKeywords is the name we give to our aggregation.

A significant term aggregation can be compared to keyword detection. The internal algorithm looks for words that are "more important" for the selected set of documents than they are in the overall population of documents.

You see new code here. You got rid of the query string search and used a filter instead. The filter is encapsulated within the query part because search queries can be combined with filters. It doesn't occur in this example, but when this happens, Elasticsearch will first apply the far more efficient filter before attempting the search. If you know you want to search in a subset of your data, it's always a good idea to add a filter to first create this subset. To demonstrate this, consider the following two snippets of code. They yield the same results but they're not the exact same thing.

A simple query string searching for "diabetes" in the disease name:

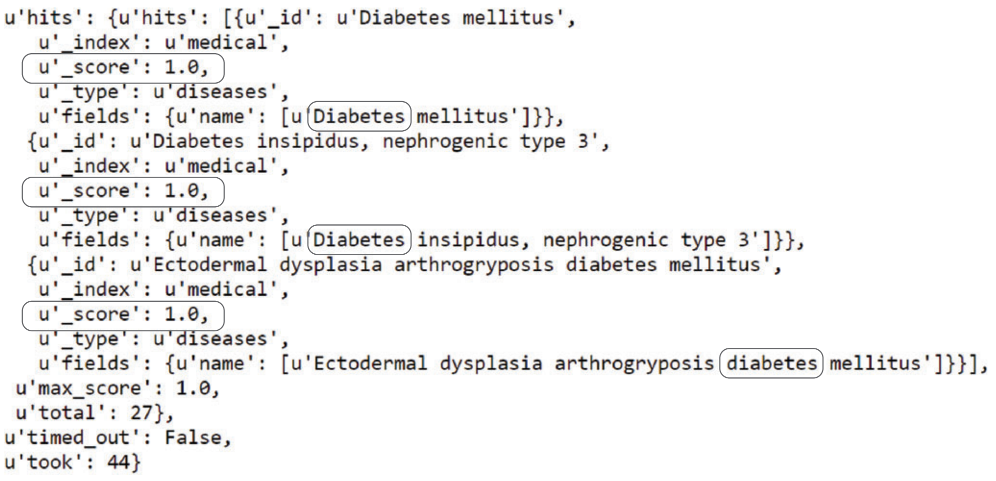
```
"query":{
    "simple_query_string" : {
        "query": 'diabetes',
        "fields": ["name"]
        }
    }
```

A term filter filtering in all the diseases with "diabetes" in the name:

```
"query":{
    "filtered" : {
        "filter": {
            'term': {'name':'diabetes'}
        }
    }
}
```

Although it won't show on the small amount of data at your disposal, the filter is way faster than the search. A search query will calculate a search score for each of the diseases and rank them accordingly, whereas a filter simply filters out all those that don't comply. A filter is thus far less complex than an actual search: it's either "yes" or "no" and this is evident in the output. The score is 1 for everything; no distinction is made within the result set. The output consists of two parts now because of the significant terms aggregation. Before you only had hits; now you have hits and aggregations. First, have a look at the hits in figure 6.31.

This should look familiar by now with one notable exception: all results have a score of 1. In addition to being easier to perform, a filter is cached by Elasticsearch for



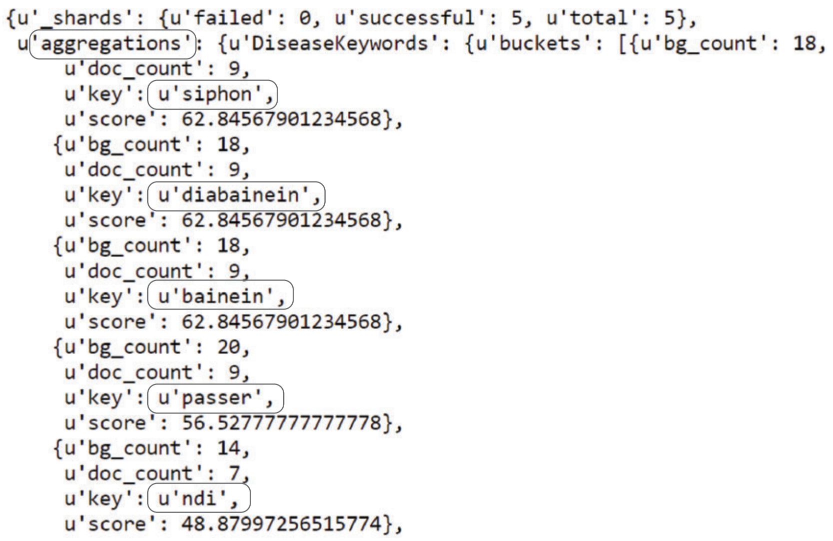
```
u'hits': {u'hits': [{u'_id': u'Diabetes mellitus',
   u'_index': u'medical',
   u'_score': 1.0,
   u'_type': u'diseases',
   u'fields': {u'name': [u'Diabetes mellitus']}},
  {u'_id': u'Diabetes insipidus, nephrogenic type 3',
   u'_index': u'medical',
   u'_score': 1.0,
   u'_type': u'diseases',
   u'fields': {u'name': [u'Diabetes insipidus, nephrogenic type 3']}},
  {u'_id': u'Ectodermal dysplasia arthrogryposis diabetes mellitus',
   u'_index': u'medical',
   u'_score': 1.0,
   u'_type': u'diseases',
   u'fields': {u'name': [u'Ectodermal dysplasia arthrogryposis diabetes mellitus']}}],
 u'max_score': 1.0,
 u'total': 27},
u'timed_out': False,
u'took': 44}
```

Figure 6.31 Hits output of filtered query with the filter "diabetes" on disease name

awhile. This way, subsequent requests with the same filter are even faster, resulting in a huge performance advantage over search queries.

When should you use filters and when search queries? The rule is simple: use filters whenever possible and use search queries for full-text search when a ranking between the results is required to get the most interesting results at the top.

Now take a look at the significant terms in figure 6.32.



```
{u'_shards': {u'failed': 0, u'successful': 5, u'total': 5},
 u'aggregations': {u'DiseaseKeywords': {u'buckets': [{u'bg_count': 18,
    u'doc_count': 9,
    u'key': u'siphon',
    u'score': 62.84567901234568},
   {u'bg_count': 18,
    u'doc_count': 9,
    u'key': u'diabainein',
    u'score': 62.84567901234568},
   {u'bg_count': 18,
    u'doc_count': 9,
    u'key': u'bainein',
    u'score': 62.84567901234568},
   {u'bg_count': 20,
    u'doc_count': 9,
    u'key': u'passer',
    u'score': 56.52777777777778},
   {u'bg_count': 14,
    u'doc_count': 7,
    u'key': u'ndi',
    u'score': 48.87997256515774},
```

Figure 6.32 Diabetes significant terms aggregation, first five keywords

If you look at the first five keywords in figure 6.32 you'll see that the top four are related to the origin of diabetes. The following Wikipedia paragraph offers help:

> The word diabetes (/ˌdaɪ.əˈbiːtiːz/ or /ˌdaɪ.əˈbiːtɨs/) comes from Latin diabētēs, which in turn comes from Ancient Greek διαβήτης (diabētēs) which literally means "a passer through; a siphon" [69]. Ancient Greek physician Aretaeus of Cappadocia (fl. 1st century CE) used that word, with the intended meaning "excessive discharge of urine," as the name for the disease [70, 71, 72]. Ultimately, the word comes from Greek διαβαίνειν (diabainein), meaning "to pass through," [69] which is composed of δια- (dia-), meaning "through" and βαίνειν (bainein), meaning "to go" [70]. The word "diabetes" is first recorded in English, in the form diabete, in a medical text written around 1425.
>
> —Wikipedia page Diabetes_mellitus

This tells you where the word *diabetes* comes from: "a passer through; a siphon" in Greek. It also mentions *diabainein* and *bainein*. You might have known that the most relevant keywords for a disease would be the actual definition and origin. Luckily we asked for 30 keywords, so let's pick a few more interesting ones such as *ndi*. *ndi* is a lowercased version of *NDI*, or "Nephrogenic Diabetes Insipidus," the most common acquired form of diabetes. Lowercase keywords are returned because that's how they're stored in the index when we put it through the standard analyzer when indexing. We didn't specify anything at all while indexing, so the standard analyzer was used by default. Other interesting keywords in the top 30 are *avp*, a gene related to diabetes; *thirst*, a symptom of diabetes; and *Amiloride*, a medication for diabetes. These keywords do seem to profile diabetes, but we're missing multi-term keywords; we stored only individual terms in the index because this was the default behavior. Certain words will never show up on their own because they're not used that often but are still significant when used in combination with other terms. Currently we miss out on the relationship between certain terms. Take *avp*, for example; if *avp* were always written in its full form "Nephrogenic Diabetes Insipidus," it wouldn't be picked up. Storing *n-grams* (combinations of *n* number of words) takes up storage space, and using them for queries or aggregations taxes the search server. Deciding where to stop is a balance exercise and depends on your data and use case.

Generally, bigrams (combination of two terms) are useful because meaningful bigrams exist in the natural language, though 10-grams not so much. Bigram key concepts would be useful for disease profiling, but to create those bigram significant term aggregations you'd need them stored as bigrams in your index. As is often the case in data science, you'll need to go back several steps to make a few changes. Let's go back to the data preparation phase.

6.2.4 Step 3 revisited: Data preparation for disease profiling

It shouldn't come as a surprise that you're back to data preparation, as shown in figure 6.33. The data science process is an iterative one, after all. When you indexed your data, you did virtually no data cleansing or data transformations. You can add data cleansing now by, for instance, stop word filtering. *Stop words* are words that are so common that they're often discarded because they can pollute the results. We won't go into stop word filtering (or other data cleansing) here, but feel free to try it yourself.

To index bigrams you need to create your own token filter and text analyzer. A *token filter* is capable of putting transformations on tokens. Your specific token filter

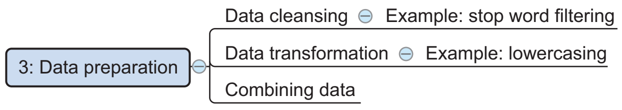


Figure 6.33 Data science process step 3: data preparation. Data cleansing for text can be stop word filtering; data transformation can be lowercasing of characters.

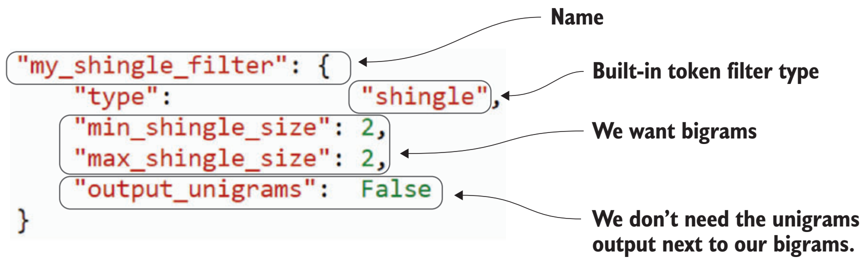
needs to combine tokens to create *n*-grams, also called *shingles*. The default Elasticsearch tokenizer is called the standard tokenizer, and it will look for word boundaries, like the space between words, to cut the text into different tokens or terms. Take a look at the new settings for your disease index, as shown in the following listing.

Listing 6.5 Updating Elasticsearch index settings

```
settings={
    "analysis": {
            "filter": {
                "my_shingle_filter": {
                    "type":               "shingle",
                    "min_shingle_size": 2,
                    "max_shingle_size": 2,
                    "output_unigrams":  False
                }
            },
            "analyzer": {
                "my_shingle_analyzer": {
                    "type":               "custom",
                    "tokenizer":          "standard",
                    "filter": [
                        "lowercase",
                        "my_shingle_filter"
                    ]
                }
            }
        }
    }
client.indices.close(index=indexName)
client.indices.put_settings(index=indexName , body = settings)
client.indices.open(index=indexName)
```

Before you can change certain settings, the index needs to be closed. After changing the settings, you can reopen the index.

You create two new elements: the token filter called "my shingle filter" and a new analyzer called "my_shingle_analyzer." Because *n*-grams are so common, Elasticsearch comes with a built-in shingle token filter type. All you need to tell it is that you want the bigrams `"min_shingle_size" : 2, "max_shingle_size" : 2`, as shown in figure 6.34. You could go for trigrams and higher, but for demonstration purposes this will suffice.

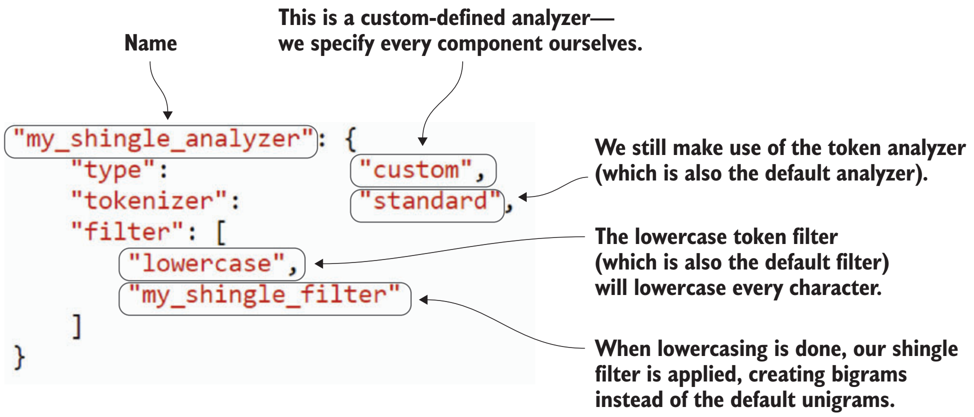


```
"my_shingle_filter": {
    "type":              "shingle",
    "min_shingle_size": 2,
    "max_shingle_size": 2,
    "output_unigrams":   False
}
```

Name
Built-in token filter type
We want bigrams
We don't need the unigrams output next to our bigrams.

Figure 6.34 A shingle token filter to produce bigrams

The analyzer shown in figure 6.35 is the combination of all the operations required to go from input text to index. It incorporates the shingle filter, but it's much more than this. The tokenizer splits the text into tokens or terms; you can then use a lowercase filter so there's no difference when searching for "Diabetes" versus "diabetes." Finally, you apply your shingle filter, creating your bigrams.



```
"my_shingle_analyzer": {
    "type":             "custom",
    "tokenizer":        "standard",
    "filter": [
        "lowercase",
        "my_shingle_filter"
    ]
}
```

Name
This is a custom-defined analyzer—we specify every component ourselves.
We still make use of the token analyzer (which is also the default analyzer).
The lowercase token filter (which is also the default filter) will lowercase every character.
When lowercasing is done, our shingle filter is applied, creating bigrams instead of the default unigrams.

Figure 6.35 A custom analyzer with standard tokenization and a shingle token filter to produce bigrams

Notice that you need to close the index before updating the settings. You can then safely reopen the index knowing that your settings have been updated. Not all setting changes require the index to be closed, but this one does. You can find an overview of what settings need the index to be closed at http://www.elastic.co/guide/en/elasticsearch/reference/current/indices-update-settings.html.

The index is now ready to use your new analyzer. For this you'll create a new document type, `diseases2`, with a new mapping, as shown in the following listing.

Listing 6.6 Create more advanced Elasticsearch doctype mapping

```
docType = 'diseases2'
diseaseMapping = {
        'properties': {
            'name': {'type': 'string'},
            'title': {'type': 'string'},
            'fulltext': {
                "type": "string",
                "fields": {
                    "shingles": {
                        "type":      "string",
                        "analyzer": "my_shingle_analyzer"
                    }
```

The new disease mapping differs from the old one by the addition of the fulltext.shingles field that contains your bigrams.

```
                    }
                }
            }
        }
client.indices.put_mapping(index=indexName,
doc_type=docType,body=diseaseMapping )
```

Within `fulltext` you now have an extra parameter, `fields`. Here you can specify all the different isotopes of `fulltext`. You have only one; it goes by the name `shingles` and will analyze the `fulltext` with your new `my_shingle_analyzer`. You still have access to your original `fulltext`, and you didn't specify an analyzer for this, so the standard one will be used as before. You can access the new one by giving the property name followed by its field name: `fulltext.shingles`. All you need to do now is go through the previous steps and index the data using the Wikipedia API, as shown in the following listing.

Listing 6.7 Reindexing Wikipedia disease explanations with new doctype mapping

```
dl = wikipedia.page("Lists_of_diseases")
diseaseListArray = []
for link in dl.links[15:42]:
    try:
        diseaseListArray.append(wikipedia.page(link))
    except Exception,e:
        print str(e)

checkList = [["0","1","2","3","4","5","6","7","8","9"],
["A"],["B"],["C"],["D"],["E"],["F"],["G"],
["H"],["I"],["J"],["K"],["L"],["M"],["N"],
["O"],["P"],["Q"],["R"],["S"],["T"],["U"],
["V"],["W"],["X"],["Y"],["Z"]]
```

The checklist is an array containing allowed "first characters." If a disease doesn't comply, you skip it.

Loop through disease lists.

```
for diseaselistNumber, diseaselist in enumerate(diseaseListArray):
    for disease in diseaselist.links: #loop through lists of links for every
     disease list
        try:
            if disease[0] in checkList[diseaselistNumber]
and disease[0:3] !="List":
                currentPage = wikipedia.page(disease)
                client.index(index=indexName,
doc_type=docType,id = disease, body={"name": disease,
"title":currentPage.title ,
"fulltext":currentPage.content})
        except Exception,e:
            print str(e)
```

First check if it's a disease, then index it.

There's nothing new here, only this time you'll index doc_type `diseases2` instead of `diseases`. When this is complete you can again move forward to step 4, data exploration, and check the results.

6.2.5 Step 4 revisited: Data exploration for disease profiling

You've once again arrived at data exploration. You can adapt the aggregations query and use your new field to give you bigram key concepts related to diabetes:

Listing 6.8 Significant terms aggregation on "diabetes" with bigrams

```
searchBody={
"fields":["name"],
"query":{
    "filtered" : {
        "filter": {
            'term': {'name':'diabetes'}
        }
    }
},
"aggregations" : {
        "DiseaseKeywords" : {
            "significant_terms" : { "field" : "fulltext", "size" : 30 }
        },
        "DiseaseBigrams": {
            "significant_terms" : { "field" : "fulltext.shingles",
"size" : 30 }
        }
    }
}
client.search(index=indexName,doc_type=docType,
body=searchBody, from_ = 0, size=3)
```

Your new aggregate, called `DiseaseBigrams`, uses the `fulltext.shingles` field to provide a few new insights into diabetes. These new key terms show up:

- *Excessive discharge*—A diabetes patient needs to urinate frequently.
- *Causes polyuria*—This indicates the same thing: diabetes causes the patient to urinate frequently.
- *Deprivation test*—This is actually a trigram, "water deprivation test", but it recognized *deprivation test* because you have only bigrams. It's a test to determine whether a patient has diabetes.
- *Excessive thirst*—You already found "thirst" with your unigram keyword search, but technically at that point it could have meant "no thirst."

There are other interesting bigrams, unigrams, and probably also trigrams. Taken as a whole, they can be used to analyze a text or a collection of texts before reading them. Notice that you achieved the desired results without getting to the modeling stage. Sometimes there's at least an equal amount of valuable information to be found in data exploration as in data modeling. Now that you've fully achieved your secondary objective, you can move on to step 6 of the data science process: presentation and automation.

6.2.6 Step 6: Presentation and automation

Your primary objective, disease diagnostics, turned into a self-service diagnostics tool by allowing a physician to query it via, for instance, a web application. You won't build a website in this case, but if you plan on doing so, please read the sidebar "Elasticsearch for web applications."

Elasticsearch for web applications

As with any other database, it's bad practice to expose your Elasticsearch REST API directly to the front end of web applications. If a website can directly make POST requests to your database, anyone can just as easily delete your data: there's always a need for an intermediate layer. This middle layer could be Python if that suits you. Two popular Python solutions would be Django or the Django REST framework in combination with an independent front end. Django is generally used to build *round-trip applications* (web applications where the server builds the front end dynamically, given the data from the database and a templating system). The Django REST framework is a plugin to Django, transforming Django into a REST service, enabling it to become part of single-page applications. A single-page application is a web application that uses a single web page as an anchor but is capable of dynamically changing the content by retrieving static files from the HTTP server and data from RESTful APIs. Both approaches (round-trip and single-page) are fine, as long as the Elasticsearch server itself isn't open to the public, because it has no built-in security measures. Security can be added to Elasticsearch directly using "Shield," an Elasticsearch payable service.

The secondary objective, disease profiling, can also be taken to the level of a user interface; it's possible to let the search results produce a word cloud that visually summarizes the search results. We won't take it that far in this book, but if you're interested in setting up something like this in Python, use the word_cloud library (`pip install word_cloud`). Or if you prefer JavaScript, D3.js is a good way to go. You can find an example implementation at http://www.jasondavies.com/wordcloud/#%2F%2Fwww.jasondavies.com%2Fwordcloud%2Fabout%2F.

Adding your keywords on this D3.js-driven website will produce a unigram word cloud like the one shown in figure 6.36 that can be incorporated into the presentation

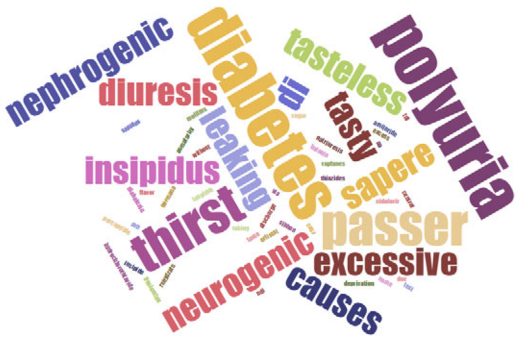


Figure 6.36 Unigram word cloud on non-weighted diabetes keywords from Elasticsearch

of your project results. The terms aren't weighted by their score in this case, but it already provides a nice representation of the findings.

Many improvements are possible for your application, especially in the area of data preparation. But diving into all the possibilities here would take us too far; thus we've come to the end of this chapter. In the next one we'll take a look at streaming data.

6.3 Summary

In this chapter, you learned the following:

- *NoSQL* stands for "Not Only Structured Query Language" and has arisen from the need to handle the exponentially increasing amounts and varieties of data, as well as the increasing need for more diverse and flexible schemas such as network and hierarchical structures.
- Handling all this data requires database partitioning because no single machine is capable of doing all the work. When partitioning, the CAP Theorem applies: you can have availability or consistency but never both at the same time.
- Relational databases and graph databases hold to the ACID principles: atomicity, consistency, isolation, and durability. NoSQL databases generally follow the BASE principles: basic availability, soft state, and eventual consistency.
- The four biggest types of NoSQL databases
  - *Key-value stores*—Essentially a bunch of key-value pairs stored in a database. These databases can be immensely big and are hugely versatile but the data complexity is low. A well-known example is Redis.
  - *Wide-column databases*—These databases are a bit more complex than key-value stores in that they use columns but in a more efficient way than a regular RDBMS would. The columns are essentially decoupled, allowing you to retrieve data in a single column quickly. A well-known database is Cassandra.
  - *Document stores*—These databases are little bit more complex and store data as documents. Currently the most popular one is MongoDB, but in our case study we use Elasticsearch, which is both a document store and a search engine.
  - *Graph databases*—These databases can store the most complex data structures, as they treat the entities and relations between entities with equal care. This complexity comes at a cost in lookup speed. A popular one is Neo4j, but GraphX (a graph database related to Apache Spark) is winning ground.
- Elasticsearch is a document store and full-text search engine built on top of Apache Lucene, the open source search engine. It can be used to tokenize, perform aggregation queries, perform dimensional (faceted) queries, profile search queries, and much more.