Appendix B
Big Data

B.1 What Is Big Data?

In recent years, the amount of data available for analysis has increased tremendously in many fields. This is due to the confluence of several factors: more processes are mediated by computer (think online shopping, automated factory control, etc.) and it is very easy to have the computer keep a record of every single operation it carries out; storage is getting cheaper, and software tools for handling these large datasets have been developed.

As a consequence, many people talk about *Big Data* to refer to this increase in the volume of data. However, this is a very imprecise term that means different things to different people in different contexts. When does data become *big*? Is it just a question of size?

Sometimes people refer to the five (or seven or even ten) Vs of Big Data; the basic three that everyone seems to agree on are:

- Volume: this refers to sheer size. If data is tabular, this usually means the total number of rows in the tables in your database. Nowadays, one can find tables with tens of millions of rows—and more. This means that most analyses, even the simple types described in Chaps. 3 and 4 are challenging to implement in a fast (interactive) manner. This has led to considerable research, and several solutions to this challenge (which we outline below) are available.
- Variety: this refers to the diversity or heterogeneity of the data. When data comes from several sources and it is in several forms (some structured, some semistructured, some unstructured) it usually must be integrated, that is, put together under a common schema, before analysis. Such integration is extremely difficult; in spite of many years of research and many papers in the subject, integrating data is still more art than science, and it usually requires a large investment of time and resources.

A. Badia, *SQL for Data Science*, Data-Centric Systems and Applications,
https://doi.org/10.1007/978-3-030-57592-2

- Velocity: this refers to the rate at which data keeps on coming; you will sometimes hear people refer to *streaming* data or similar terms. If data keeps on coming at high speed, it not only increases rapidly in volume; it also puts the results of analysis at the risk of becoming obsolete quite rapidly. This creates its own challenges if we want to keep our information up-to-date.

However, most of the time when people use the term Big Data they refer to the first characteristic only, Volume. The question, then, is when is Big Data big enough to deserve this name? To explain where the border lies, we have to explain a little bit about computer architecture.

There are *two* distinct memories in any computer. The first one is sometimes called RAM, while the second one is usually referred to as the *disk* (or the drive or the hard drive).[1] The computer moves data and programs between these two memories because each one has a different function. RAM is used for data that the computer needs to operate on; it is very fast, but is also *volatile*—meaning that all the contents of RAM are erased if the computer is turned off or it crashes. The disk, however, is what is called a stable memory: it keeps the data whether the computer is on or off. It is used for long-term storage; that is where files reside. Unfortunately, the computer cannot work with the data in the disk; it needs to move data to RAM before it can make changes to it or even read it for analysis. Because of the current state of technology and manufacturing, disk is much cheaper than RAM, and so most computers have much larger disks than they have RAM memory—both RAM and disks have become cheaper with time, so computers come with larger and larger amounts of both, but the ratio of the size of RAM to the size of disk has not changed that much in a decade. As of 2019, a typical computer will have between 30 and 100 times more disk than RAM.

In summary, RAM is fast, unstable, and (relatively) small, while disk is slow, stable, and big. As a consequence, any activity that requires accessing large amounts of data from disk becomes too tedious and cumbersome; in some cases, it is not even doable. To cope with large data, database systems store data on disk but try to become efficient at accessing it, either by using quick access methods (indices) or by optimizing the execution of queries or a combination of both. They still need to move the data from disk to RAM, so access to large datasets may take a while. Sometimes the data to be analyzed may not fit into RAM; in those cases, database systems are designed so that they still can carry out the required task—at some extra time cost. This is what allows databases to handle very large amounts of data.

In contrast, R works with data in RAM. It reads data from a file in the disk, but it needs to be able to move all the data to RAM to work with it. As a result, R is limited on the size of datasets it can handle.[2] Python, as any program language, works with files and allows a programmer to decide exactly how to handle such files. Hence, Python can also deal with very large datasets, but it is up to the programmer to find

[1] In many modern laptops, hard drives are being substituted by other types of devices, so sometimes one may hear talk of SSDs instead.

[2] New packages are being developed to work around this limitation (see Chap. 6 for more on R).

ways to handle large files efficiently. Database systems, however, take care of this in a manner that it is transparent to the user.

Thus, the line for in-depth analysis is crossed as soon as data exceeds the size of available RAM. However, lightweight or selective analysis can still be done with data that fits on a medium-sized disk, although there will be an initial hit when moving the data.[3] One should be aware of the size of datasets that they need to handle and compare them to their computer's capability.[4] Fortunately, most people work with little (or medium-sized) data, not Big Data. The exceptions are people working at the so-called Internet companies (Google, Facebook, Twitter, Uber, etc.) or at large retailers (Walmart, Target) or at medium or large companies that happen to handle data as part of their regular business. The rest of us rarely will see the amounts of data that qualify as big.

Once this is said, people's jobs and requirements change, so we summarize here some technologies that help deal with large data. It is worth noting that this is an instance where relational databases shine. Because their data model and language are independent of the underlying hardware, it does not matter whether a database is running on a laptop, a computer in a local lab, a remote server, a computer cluster, or the cloud. The only thing a user needs to care about is learning how the relational model stores data and how to use SQL—something that hopefully this book has helped with. Once these skills are acquired, they can be deployed in a wide range of scenarios, as we will see.

B.2 Data Warehouses

A *data warehouse* is a special type of database that is designed to take in large amounts of data as far as such data fits into a certain model, sometimes called the *multidimensional* data model. Data warehouses have several techniques to support fast analysis of very large datasets. However, data warehouses are geared toward business analytics and are not necessarily a good match for general data analytics—in particular, Machine Learning mathematics-based computations. Since they support SQL, though, everything that was explained in this book can be done in a data warehouse and at a large scale.

The multidimensional model looks at data as a collection of basic *facts*; typically, there is only one or very few types of facts in a data warehouse. Each fact describes the basic activity that the warehouse was built to record. For instance, a data warehouse for an e-commerce website usually registers each single visit to the site:

[3] A modern disk can read data at anything between 80 and 160 Mbs per second, so a 1 GB file will take about 13 and 7 s to read, while a 100 GB file will take between 22 and 12 min.

[4] Another practical roadblock is *downloading* data. If the data must be obtained from the Web or a remote site, and downloading speed is 50 Mbs per second, 1 GBs of data will take approximately 21 s to download (assuming no network issues); 100 GBs will take about 35 min (and you can be almost assured of some network issues).

the page visited, the day and time, the IP address of the visitor, the OS and browser of the visiting computer (if available), and what was clicked on (if anything). Another typical example is a telecommunication warehouse that registers phone calls: the basic fact here is the call, including which phone number called, which phone number was called, the day and time, and the length of the conversation. Each fact is described by a series of characteristics, which are divided into *measures* and *dimensions*. Measures are typically simple numerical values (like the length of the phone call), while dimensions are complex attributes for which more information is available (like the phone numbers involved in a conversation, for which associated information may include the person having such number, her address, etc.). The data warehouse is organized as a relational database with one or more *fact tables*, which hold the basic information about the facts, and *dimensions tables*, one for each dimension, holding the information available for that dimension. For instance, in the telecommunications database we would have a `CALLS` fact table, with a schema like `(day, time, caller, callee, call_length)`. We would have at least one dimension table, `ACCOUNTS`, so that each `caller` and `callee` are associated with an entry on this table, which would include attributes like `(phone-number, name, address)` and possibly many others. Note that `(phone-number)` would be the primary key of the `ACCOUNTS` table, and that both `caller` and `callee` are foreign keys to it. This is a typical organization for a data warehouse, with the fact table holding (at least) one foreign key for each dimension. This is many times visualized with the fact table in the middle of a circle-like arrangement of dimension tables; because of this visual, this schema is typically called a *star schema*. As another example, the e-commerce site may have a fact table `VISITS`, registering each visit to a page, and may have dimensions `PAGE` (where the information contained on a web page is described) and `CUSTOMER` (if the visitor is a registered customer and has logged in to the site or can be identified based on IP address). In some data warehouses, the dimensions may contain complex information and therefore keeping all information about a dimension in a single table may lead to redundancy due to some of the reasons discussed in Sect. 2.2. In those cases, dimension tables are *normalized* and broken down into several auxiliary tables to avoid redundancy. When this is done, we talk about a *snowflake schema* instead of a star schema.

Note that in both the example of the e-commerce site and the telecommunications company, there is temporal information involved. Time is almost always one of the dimensions of the data warehouse, as the warehouse tends to accumulate collections of facts for a relatively long period of time (usually, a few years) in order to facilitate temporal analysis. Sometimes, the temporal dimension is explicitly stored in a table, especially when time must be broken down into periods that are not standard: days, weeks, and months can be considered standards, while others (like fiscal quarters) may not be. Sometimes, the temporal dimension is handling implicitly (in SQL, though data manipulations with date functions, without having an explicit table for it).

The fact table tends to be very large, as it registers a large collection of basic facts. Think of a retailer like Amazon, which does log (collect) all visits to all the pages in its site, when such visits may number in the millions each day; or a large

phone company, which also handles millions of calls each day. It is not unknown to have fact tables with hundreds of millions of rows on it. By contrast, dimension tables tend to be considerably shorter, from hundreds to a few thousand rows.

Data warehouses typically get their data from other databases, usually called *transaction* databases because they handle transactions, that is, changes to the database in the form of INSERT statements that keep data in such databases current. The typical example is a brick-and-mortar retailer chain, which may have a database running on each store, registering the sales that are taking place in that store. Every so often, all data in the store database is transferred to data warehouse which collects all data for the whole chain—that is, all stores are connected to this data warehouse and transfer their data to the warehouse. This "gathering" of data is one of the reasons that the fact table in a warehouse grows so large (another one is that data, as stated before, is kept for quite some time).

The transfer of data between the transaction databases and the data warehouse is usually carried out at pre-determined intervals (typically, anything from 6 to 24 h is used). During this transfer, data from all sources is typically cleaned, standardized, and integrated, so that it all fits nicely in the schema of the warehouse. This is typically called the *ETL (Extract-Transform-Load)* process. When data comes from heterogeneous, highly variable sources, this can be quite a complex process which requires large amounts of code. In our example of a retail chain, chances are that all stores use the same schema for their databases—since it was likely designed by a group within the company for all stores to use. In such cases, ETL is much simpler.

Data warehouses are typically used by business analyst which take advantage of the fact that the warehouse contains data from the whole company to ask *global* questions (i.e. how are the stores in the Northwest area doing with respect to other areas? Or, which were the top 10 sellers throughout the whole company in the last week?). They also use the fact that the warehouse stores data for a long time to examine patterns of evolution and change (i.e. how are sales of perishable goods done over the last year? What was the month with the highest (and/or lowest) monthly sales of such items?). Clearly, such questions could not be asked of the database in any particular store.

The typical queries asked of a data warehouse are sometimes called *Decision Support* queries, and warehouses are sometimes referred to as *Decision Support Systems (DSS)*. The reason is that the information obtained through data warehouse queries are used by middle- and upper-level managers to make decisions about the company's future actions, i.e. purchases or discount campaigns. These queries are sometimes described using a specialized vocabulary (*slice and dice*, *roll-up*), but they are simply SQL queries that use certain patterns over the multidimensional data in the warehouse. To explain them, it is useful to visualize multidimensional data as points in a multidimensional space (just like we did in Sect. 4.3.1). As a simple example, assume a `SALES` fact table with dimensions `PRODUCT`, `CUSTOMER`, and `STORE` (i.e. what was sold, to whom, and on what store). We can see each record in `SALES` as representing a point in three-dimensional space: given a product `p`, a customer `c`, and a store `s`, the point at `(p, c, s)` is associated with certain measures (say, the number of products bought `n` and the total amount spent `a`).

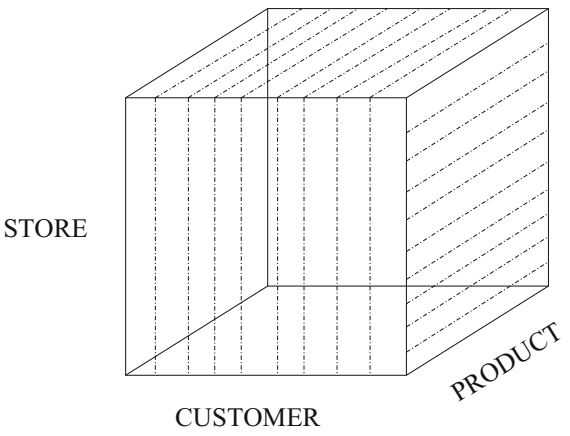

Fig. B.1 Three-dimensional data points as a cube

This point and its associated measures are represented the record `(p, c, s, n, a)` in fact table `SALES`. In a relational database, this would be implemented giving each dimension a primary key (say, `storeid` for `STORE`, `custid` for `CUSTOMER`, and `productid` for `PRODUCT`) and using them as foreign keys in the `SALES` table. We can also visualize the space as a cube since we conveniently made it three-dimensional (see Fig. B.1).

We can now describe typical data warehouse queries as follows:

- *Slicing*: to cut a slice of the cube, we take one of the dimensions and we fix it by using an equality condition. For instance, using `storeid = 112` cuts a slice of the cube along the `STORE` dimension. Slicing allows us to reduce the number of dimensions of what we are observing; it can be done using more than one dimension. Sometimes slicing refers to fixing an attribute of a dimension; for instance, assume that table `STORE` has attribute `city`; we use the condition `STORE.city = 'Lexington'` and join tables `STORE` and `SALES` in order to retrieve measures and other information related to sales in stores in that city.
- *Dicing*: to dice up the cube, that is, to get a smaller cube of the larger one. This is usually done by selecting on one or more dimensions with a range condition, that is, one that obtains a range of values instead of a single one, or with a condition on a categorical attribute. As with slicing, this can be done on the dimension itself or on an attribute of the dimension. For instance, using `PRODUCT.price BETWEEN 10 and 100` would take a 'chunk' of the `PRODUCT` dimension; using `STORE.state = 'NY'` would do the same (assuming there are several stores in the same state).
- *Roll-up*: facts in a fact table are usually not interesting on their own; what the analyst is really after is any trends and patterns in the collection of facts. Hence, when facts (dimensions or measures) are analyzed, they are usually *grouped and summarized* (using GROUP BY and aggregates). This also results in a small (hence easy to interpret) answer—it is difficult to make sense of an answer with

hundreds of thousands of records. The typical approach aggregates facts by some characteristics of one or more dimensions; for instance, we could look at sales by each state. In many instances, we can look at categories at several levels, as the attributes are naturally organized in a hierarchy: for instance, geographical and temporal hierarchies are very common. Geographically, we can look at all sales in a city, a county (grouping several cities together), a state (grouping several counties together), a region (grouping several states together), or a country (grouping several regions together). Temporally, we could look at all sales on a day, a week, a month, a quarter, or a year. In such cases, to *roll-up* is to summarize facts by one of the low levels in the hierarchy and then move up the hierarchy, each time getting a smaller, more coarse answer (but ones where patterns may appear more clearly). For instance, we may start to look at sales per city, then roll up by county, and then by state. In SQL, this can be accomplished with `ROLLUP` and `CUBE`, as described in Sect. 5.3.

- *Drill down*: this is the opposite of roll-up; it consists of going down in a hierarchy to get more fine grained answers. Sometimes we may notice something in an answer and we may need more detail; for instance, when looking at sales per quarter, we notice that one of them has substantially higher sales than the others. We then drill down to the months making up this quarter, and we find one of them that stands out. We again drill down to the weeks of that month, and we find a particular week that explains most of the surge in sales.

In most data warehouse queries, the fact table is joined with one or more dimensions, conditions are placed on such dimensions, and then a grouping and several aggregates are applied. Moving across dimensions and conditions allows us to examine those facts from several angles.

The important lesson to remember is that, in spite of all the new vocabulary, ultimately the data warehouse is basically a specialized relational database and the analysis of its data is done using SQL. However, there are also many software products that connect to a data warehouse, extract some of the data from it, bring it to memory, and analyze it there in order to achieve fast response times. Many times, such products have their own query languages, usually with ad hoc constructs that allow simulating spreadsheet-like operations on the data. Unfortunately, since there are many such languages, and each one has its own idiosyncrasies and peculiarities, we will not be describing them here. Most of them do not do anything that cannot be done in SQL.

We close this very brief description by pointing out that, in some cases, data warehouses may have complex schemas, with more than one fact table and many dimension tables for each. Being a central repository of information for the enterprise, data warehouses can grow in scope, complexity, and size. In some cases, a group of users that are only interested in part of what is available in the warehouse create a *data mart*. This is a database that contains only part of the data in the warehouse, either by restricting the schema (usually, picking one of the fact tables if there are several, and a subset of dimension tables) or by restricting the data (usually, picking some of the data in the fact table; for instance, the facts for the last

6 months). A data mart simplifies analysis and allows for faster response time since it does not have the size of the data warehouses. Usually, one or more data marts may be created after a data warehouse is up and running to service the needs of users with interest in only some parts of the warehouse. Again, a data mart is essentially a relational database and is managed using SQL (although, as in the case of the data warehouse, specialized tools may be used with the data mart to facilitate analysis).

B.3 Cluster Databases

A *computer cluster* is a collection of computers that are connected to each other, usually by a high-speed network. Many times, the name cluster is reserved for computers that are physically close (in the same room or perhaps the same building), so that the network used is a fast local area network, which is very fast and under the control of the cluster. Each computer in the cluster is called a *node* and is a complete computing unit (i.e. with its own CPU, memory, and disk, running its own operating system independently of others). However, all computers in the cluster are run in coordination by a special software, usually with all computers in the cluster running the same task. For many processes, the cluster appears as one large computer.

Clusters are used to deal with large amounts of data. Usually, the data is distributed across all the nodes in the cluster. In the case of a relational database, the data is usually distributed by *horizontal partitioning*: the tables are broken into chunks, disjoint collections of records. All chunks for a table share the same schema with the original table. Computer in the cluster chunk is given to each one.

Database systems developed to run in clusters take SQL, just like a traditional database, but then internally break down the tasks needed to execute the query into pieces that are sent to each node. A node runs the piece received on its own chunk of data, perhaps exchanging data with other nodes if necessary at some point. Because all nodes work in parallel, and each one of them handles only a chunk of the whole dataset, clusters are able to handle extremely large datasets.

The best well-known example of open-source software that controls a cluster is Apache Hadoop.[5] Hadoop provides a set of facilities to control the cluster. For instance, Hadoop has tools to deal with *fault tolerance* and with *load balancing*. Fault tolerance refers to the problems caused by one of the nodes in the cluster going down or crashing. This could prevent the whole cluster from working if no provisos are made for this situation. Load balancing refers to the fact that, for a cluster to be as effective as possible, ideally all nodes should have about the same amount of data to deal with. An uneven distribution of the data could cause some nodes to struggle with a large dataset, while others are idle because they have little data to work on. Distributing the data in a balanced manner clearly helps performance, but is not a trivial task.

[5] https://hadoop.apache.org/.

Cluster databases support the analysis of large datasets by providing algorithms that execute most SQL queries using all the machines in the cluster. The work is automatically divided up so that all machines collaborate in obtaining an answer. This can be done automatically in SQL, thanks to its declarativeness and relative simplicity. The great advantage of this is that a user can use SQL for analysis and not worry about how the system will actually carry out the queries. Thus, most of what we have seen in Chaps. 3 and 4 can be used in these systems. The open-source data warehouse Apache Hive[6] has been developed on top of Hadoop. Hive supports most of the SQL standard; all the approaches explained in this book can be used with Hive.

Most commercial database systems (and a few open-source ones) have 'cluster' versions. Installing and maintaining these versions can get quite complicated due to the number of parameters that must be set. In many systems, it is necessary that the user decides which tables must be partitioned, how they should be partitioned, how many replicas (copies) should be kept, and so on. It is a goal of current research to make these tasks simpler by using algorithms (including machine learning algorithms) to simplify them. Still, given their added complexity, these services should only be used when the volume of data justifies it. As a rule of thumb, the volume of data should be considerably larger than what one can fit in the hard drive of a powerful computer. If a cluster is needed, it is probably a good idea for a data scientist to team up with a *data engineer*, someone who is knowledgeable about cluster deployment and maintenance.

As a brief example, the CREATE TABLE command in Apache Hive looks and behaves pretty much like the standard SQL CREATE TABLE defined in Sect. 2.1. However, it has some additional, optional parameters:

```
PARTITIONED BY (col_name data_type
CLUSTERED BY (col_name, col_name, ...)
SORTED BY (col_name [ASC|DESC], ...) INTO num_buckets BUCKETS
SKEWED BY (col_name, col_name, ...)
     ON ((col_value, col_value, ...), ...)
```

All these refer to how the data is laid out in the cluster. The `PARTITIONED BY` indicates a way to partition and distribute the data in the table among nodes in the cluster (essentially, all records with the same values for the partition columns are sent to the same node). Within each node, the data can be *bucketed* using the `CLUSTERED BY` clause; again, all records with the same values for the attributes named in this clause are stored together. Within a bucket, records may be kept in sorted order according to the attributes mentioned in `SORTED BY`. To improve load balancing, Hive allows the user to specify attributes that are heavily skewed (i.e. some values appear very often, while some others appear sparsely). The system will use the information on `SKEWED BY` (which provides not only skewed attributes, but also a list of frequent values for each) to even out data distribution by splitting the collection of records associated with frequent values.

[6] http://hive.apache.org/.

The purpose of all these optional, extra parameters is to facilitate very fast retrieval of data. It is easy to see that all these extra parameters are only useful in the context of a cluster; that is why they are absent from the standard definition. It is also clear that, in order to make good use of them, the user must have detailed information about how the data is distributed—not all attributes are good candidates for `PARTITION BY`, `CLUSTERED BY`, or `SORTED BY`. That is why it is a good idea to work with a data engineer for these decisions. The good news is that all these are irrelevant for data analysis; in a `SELECT` statement, only the table name is needed in the `FROM` clause: queries can be written just as they would over a regular database—just as it has been described in this book.

B.4 The Cloud

As explained in the previous section, cluster computing is very powerful but it also requires quite a bit of expertise in computers to implement properly. This has discouraged users that are not computer experts from working with large datasets. Lately, an idea has come to the forefront for attacking this problem: the idea of *computing-as-a-service*. This idea is based on the following premise: someone (a "cloud provider") sets up one or several computer clusters and installs software on them (in particular, database systems). This set of clusters is referred to as *a cloud*. The cloud provider sells access to these capabilities to the public. A customer can buy access to a certain amount of resources (disk, memory, CPU). Once a customer creates an account, she can access the resources needed, usually through a Web-based interface. In particular, in the case of databases, once connected into the system the user can create a schema, create tables on it, upload data to this schema, and run queries on it. The cloud provider makes sure that everything runs smoothly. If the customer needs to deal with large amounts of data, she can buy more resources (more disk, more memory, more CPU). This way, the user can buy as few or as many resources as needed by the data and does not have to deal with maintaining its own computer resources, upgrading them, etc. That is, the user can concentrate on gathering and analyzing the data.

The most famous examples of cloud computing are AWS (Amazon Web Services), Microsoft Azure, and Google Cloud services. All of them offer multiple services, including relational databases. For instance, AWS offers MySQL and Postgres, as well as Amazon Aurora (a database that is MySQL and Postgres compatible), Amazon RDS, and Amazon Redshift (a data warehouse). Microsoft Azure offers Azure SQL database, Azure database for MySQL, Azure database for PostgreSQL, and SQL Server. Google Cloud offers Cloud SQL for Postgres, Cloud SQL for MySQL and Cloud Spanner. As it can easily be seen, all the main providers support MySQL and Postgres, besides their own offerings. All that has been learned in this book can be ported to this new environment.

As a simple example, assume we have created an account with AWS to manage a MySQL database. The only extra step needed is to create an AWS *instance*. This is

simply a specification of what services we want to use and how many resources we need. AWS Cloud resources are housed in several data centers, each one in certain areas of the world (called a *region*); within each region, there are different locations (called *Availability Zones (AZ)*).[7] Because of this, the user has to start by choosing a region and an AZ for the database.[8] Once this is done, the user can create a database. A choice of databases is available; the user in this example would pick MySQL. Then the choice needs to be configured by telling AWS the level of resources needed and a few more details. Next, the user picks a database name, a username, and a password and provides AWS with some more information, this time concerning the database, not the instance (for instance, one can choose whether to encrypt communication with the database for security; whether the database should be periodically backed up; whether database access should be monitored, etc.). Once this is done, the particular database (called a *database instance*) is created. From this point on, the user can communicate with this database instance using a GUI (for instance, MySQL Workbench; see the previous Appendix) or the CLI (Command Line Interface), just like one would do with a regular database. For instance, when connecting to the database the user would enter as hostname the value provided by the AWS Console (it is called an *endpoint* there), and as username and password the values entered when creating the database instance. Now the user can do everything that she would do with a local database. To load large amounts of data, the easiest way is to use AWS DataSync, which is a service for transferring data between a user's computer (or any other data repository) and AWS computers.[9] For smaller amounts of data, one can use `INSERT INTO` statements. Data manipulation and analysis happens through SQL, and everything we have seen in this book applies.

[7] For instance, at the time of writing this, the USA has an East Region (with AZ in Virginia and Ohio) and a West Region (with AZ in California and Oregon). There is also a region for Asian Pacific (with multiple AZ in Japan, one in India, one in Singapore, and one in Australia), one for Europe, one for Canada, and one for South America.

[8] All this process is traditionally done through a Wed, form-based interface called the *AWS Console*, so it is a matter of clicking away (although one can also use the CLI).

[9] See https://docs.aws.amazon.com/datasync/latest/userguide/what-is-datasync.html for documentation.