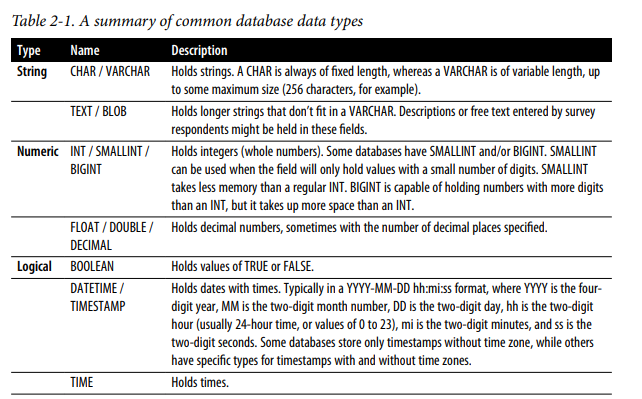
Data scientists spend from 50% to 80% of their time cleaning and wrangling their data. Preparing data is such a common task that terms have sprung up to describe it, such as data munging, data wrangling, and data prep. (“Mung” is an acronym for Mash Until No Good)

Data preparation is easier when a data set has a data dictionary, a document or repository that has clear descriptions of the fields, possible values, how the data was collected, and how it relates to other data.

Unfortunately, this is frequently not the case. Documentation often isn’t prioritized, even by people who see its value, or it becomes out-of-date as new fields and tables are added or the way data is populated changes. **Data profiling creates many of the elements of a data dictionary**, so if your organization already has a data dictionary, this is a good time to use it and contribute to it. If no data dictionary exists currently, consider starting one! This is one of the most valuable gifts you can give to your team and to your future self. An up-to-date data dictionary allows you to speed up the data-profiling process by building on profiling that’s already been done rather than replicating it. It will also improve the quality of your analysis results, since you can verify that you have used fields correctly and applied appropriate filters.

1. I’ll start with a review of data types you are likely to encounter
   1. I’ll start with the database data types most frequently encountered in analysis
   2. Then I’ll move on to some conceptual groupings that can help us understand the source, quality, and possible appli‐ cations of the data.
2. This is followed by a review of SQL query structure.
3. Next, I will talk about profiling the data as a way to get to know its contents and check for data quality
4. Then, I’ll walk through some useful tools for cleaning data to deal with any quality issues
5. Finally, I’ll talk about some data-shaping techniques that will return the columns and rows needed for further analysis

**Database Data Types**



1. **String data types** are the most versatile. These can hold letters, numbers, and special characters, including unprintable characters like tabs and newlines. String fields can be defined to hold a fixed or variable number of characters. A CHAR field could be defined to allow only two characters to hold US state abbreviations, for example, whereas a field storing the full names of states would need to be a VARCHAR to allow a variable number of characters. Fields can be defined as TEXT, CLOB (Character Large Object), or BLOB (Binary Large Object, which can include additional data types such as images), depending on the database to hold very long strings, though since they often take up a lot of space, these data types tend to be used sparingly. When data is loaded, if strings arrive that are too big for the defined data type, they may be truncated or rejected entirely. SQL has a number of string functions that we will make use of for various analysis purposes.
2. **Numeric data types** are all the ones that store numbers, both positive and negative. Mathematical functions and operators can be applied to numeric fields. Numeric data types include the INT types as well as FLOAT, DOUBLE, and DECIMAL types that allow decimal places. Integer data types are often implemented because they use less memory than their decimal counterparts. In some databases, such as Postgres, dividing integers results in an integer, rather than a value with decimal places as you might expect.
3. **The logical data type** is called BOOLEAN. It has values of TRUE and FALSE and is an efficient way to store information where these options are appropriate. Operations that compare two fields return a BOOLEAN value as a result. **This data type is often used to create flags, fields that summarize the presence or absence of a property in the data**. For example, a table storing email data might have a BOOLEAN has\_opened field.
4. **The datetime data types** include DATE, TIMESTAMP, and TIME. Date and time data should be stored in a field of one of these database types whenever possible, since SQL has a number of useful functions that operate on them. Timestamps and dates are very common in databases and are critical to many types of analysis, particularly time series analysis and cohort analysis.
5. **Other data types**, such as JSON and geographical types, are supported by some but not all databases

**Structured Versus Unstructured**

Data is often described as **structured** or **unstructured**, or sometimes as **semistructured**.

1. Most databases were designed to handle **structured data**, where each attribute is stored in a column, and instances of each entity are represented as rows. A data model is first created, and then data is inserted according to that data model. For example, an address table might have fields for street address, city, state, and postal code. Each row would hold a particular customer’s address. Each field has a data type and allows only data of that type to be entered. When structured data is inserted into a table, each field is verified to ensure it conforms to the correct data type. Structured data is easy to query with SQL.
2. **Unstructured data** is the opposite of structured data. There is no predetermined structure, data model, or data types. Unstructured data is often the “everything else” that isn’t database data. Documents, emails, and web pages are unstructured. Photos, images, videos, and audio files are also examples of unstructured data. They don’t fit into the traditional data types, and thus they are more difficult for relational databases to store efficiently and for SQL to query. Unstructured data is often stored outside of relational databases as a result. This allows data to be loaded quickly, but lack of data validation can result in low data quality.
3. **Semistructured** data falls in between these two categories. Much “unstructured” data has some structure that we can make use of. For example, emails have from and to email addresses, subject lines, body text, and sent timestamps that can be stored separately in a data model with those fields. Metadata, or data about data, can be extracted from other file types and stored for analysis. For example, music audio files might be tagged with artist, song name, genre, and duration. Generally, the structured parts of semistructured data can be queried with SQL, and SQL can often be used to parse or otherwise extract structured data for further querying.

**Quantitative Versus Qualitative Data**

1. **Quantitative data is numeric**. It measures people, things, and events. Quantitative data can include descriptors, such as customer information, product type, or device configurations, but it also comes with numeric information such as price, quantity, or visit duration. Counts, sums, average, or other numeric functions are applied to the data. Quantitative data is often machine generated these days, but it doesn’t need to be. Height, weight, and blood pressure recorded on a paper patient intake form are quantitative, as are student quiz scores typed into a spreadsheet by a teacher.
2. **Qualitative data is usually text based and includes opinions, feelings, and descriptions that aren’t strictly quantitative**. Temperature and humidity levels are quantitative, while descriptors like “hot and humid” are qualitative. The price a customer paid for a product is quantitative; whether they like or dislike it is qualitative. Survey feedback, customer support inquiries, and social media posts are qualitative. There are whole professions that deal with qualitative data. In a data analysis context, we usually try to quantify the qualitative. One technique for this is to extract keywords or phrases and count their occurrences. Another technique is sentiment analysis, in which the structure of language is used to interpret the meaning of the words used, in addition to their frequency. Sentences or other bodies of text can be scored for their level of positivity or negativity, and then counts or averages are used to derive insights that would be hard to summarize otherwise. There have been exciting advances in the field of natural language processing, or NLP, though much of this work is done with tools such as Python.

**First-, Second-, and Third-Party Data**

1. **First-party data** is collected by the organization itself. This can be done through server logs, databases that keep track of transactions and customer information, or other systems that are built and controlled by the organization and generate data of interest for analysis. Since the systems were created in-house, finding the people who built them and learning about how the data is generated is usually possible. Data analysts may also be able to influence or have control over how certain pieces of data are created and stored, particularly when bugs are responsible for poor data quality
2. **Second-party data** comes from vendors that provide a service or perform a business function on the organization’s behalf. These are often software as a service (SaaS) products; common examples are CRM, email and marketing automation tools, ecommerce-enabling software, and web and mobile interaction trackers. The data is similar to first-party data since it is about the organization itself, created by its employees and customers. However, both the code that generates and stores the data and the data model are controlled externally, and the data analyst typically has little influence over these aspects. Second-party data is increasingly imported into an organization’s data warehouse for analysis. This can be accomplished with custom code or ETL connectors, or with SaaS vendors that offer data integration.
3. **Third-party data** may be purchased or obtained from free sources such as those published by governments. Unless the data has been collected specifically on behalf of the organization, data teams usually have little control over the format, frequency, and data quality. This data often lacks the granularity of first- and second-party data. For example, most third-party sources do not have user-level data, and instead data might be joined with first-party data at the postal code or city level, or at a higher level. Third-party data can have unique and useful information, however, such as aggregate spending patterns, demographics, and market trends that would be very expensive or impossible to collect otherwise.

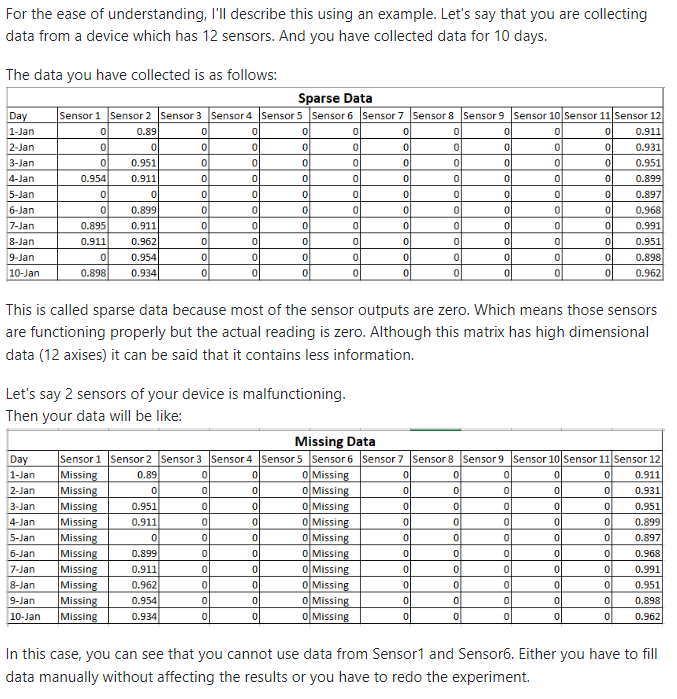
**Sparse Data**

**Sparse data occurs when there is a small amount of information within a larger set of empty or unimportant information**. Sparse data can occur when events are rare, such as software errors or purchases of products in the long tail of a product catalog. It can also occur in the early days of a feature or product launch, when only testers or beta customers have access. JSON is one approach that has been developed to deal with sparse data from a writing and storage perspective, as it stores only the data that is present and omits the rest. This is in contrast to a row-store database, which has to hold memory for a field even if there is no value in it.

Sparse data can be problematic for analysis. When events are rare, trends aren’t necessarily meaningful, and correlations are hard to distinguish from chance fluctuations. It’s worth profiling your data, to understand if and where your data is sparse. Some options are to group infrequent events or items into categories that are more common, exclude the sparse data or time period from the analysis entirely, or show descriptive statistics along with cautionary explanations that the trends are not necessarily meaningful.

There are a number of different types of data and a variety of ways that data is described, many of which are overlapping or not mutually exclusive. Familiarity with these types is useful not only in writing good SQL but also for deciding how to analyze the data in appropriate ways. You may not always know the data types in advance, which is why data profiling is so critical.

**Sparse Data vs Missing Data**



**SQL Query Structure**

SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY

LIMIT

**Profiling**

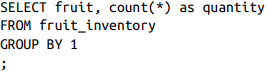
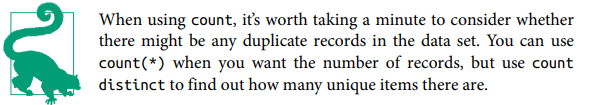
**Distributions**

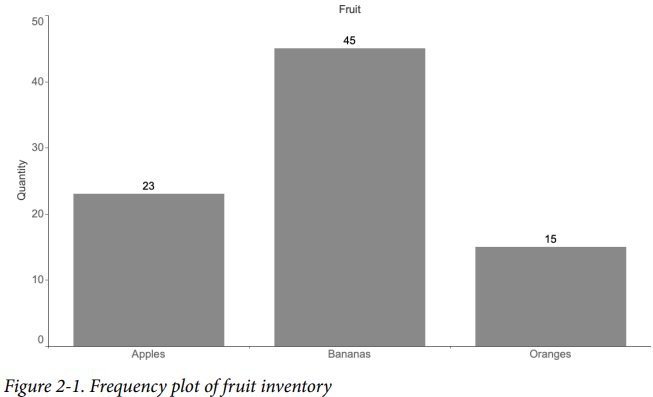
1. Profiling is the first thing I do when I start working with any new data set. I **look at how the data is arranged into schemas and tables**. I look at the table names to get familiar with the topics covered, such as customers, orders, or visits. I check out the column names in a few tables and start to construct a mental model of how the tables relate to one another. For example, the tables might include an order\_detail table with line-item breakouts that relate to the order table via an order\_id, while the order table relates to the customer table via a customer\_id. If there is a data dictionary, I review that and compare it to the data I see in a sample of rows.
2. The **tables generally represent the operations of an organization, or some subset of the operations**, so I think about what domain or domains are covered, such as ecommerce, marketing, or product interactions. Working with data is easier when we have knowledge of how the data was generated. Profiling can provide clues about this, or about what questions to ask of the source, or of people inside or outside the organization responsible for the collection or generation of the data. Even when you collect the data yourself, profiling is useful.
3. Another detail I check for is **how history is represented**, if at all. Data sets that are replicas of production databases may not contain previous values for customer addresses or order statuses, for example, whereas a well-constructed data warehouse may have daily snapshots of changing data fields
4. After checking a few samples of data, I **start looking at distributions**. Distributions allow me to understand the range of values that exist in the data and how often they occur, whether there are nulls, and whether negative values exist alongside positive ones. Distributions can be created with continuous or categorical data and are also called frequencies.

**Histograms and Frequencies**

One of the best ways **to get to know a data set**, and to know particular fields within the data set, is to **check the frequency** of values in each field. Frequency checks are also useful whenever you have a question about whether certain values are possible or if you spot an unexpected value and want to know how commonly it occurs. Frequency checks can be done on any data type, including strings, numerics, dates, and booleans. **Frequency queries are a great way to detect sparse data as well**.

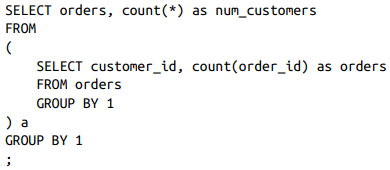
***For example, we can check the frequency of each type of fruit in a fictional fruit\_inventory table:***



* A histogram is a way to visualize **the distribution of numerical values in a data set** (A basic histogram might show the distribution of ages across a group of customers

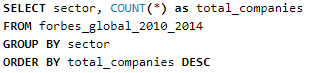
Another technique involves an aggregation followed by a frequency count. There is a table called orders, which has a date, customer identifier, order identifier, and an amount. ***Write an SQL query that returns the distribution of orders per customer***

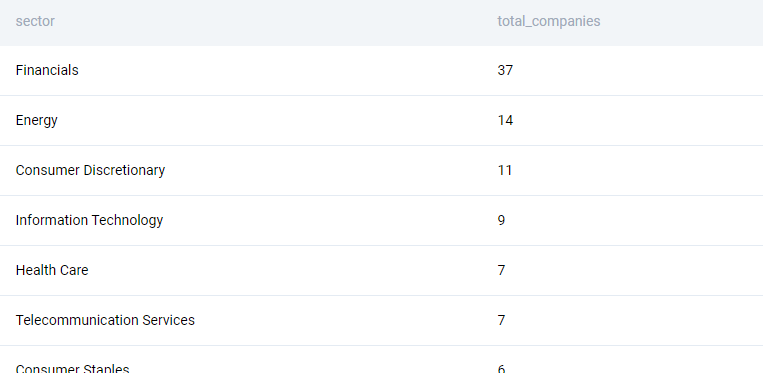


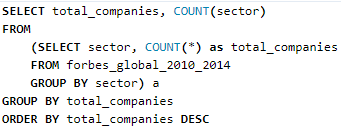
This can’t be solved with a simple query; it requires an intermediate aggregation step, which can be accomplished with a subquery.

1. Count the number of orders placed by each customer\_id in the subquery.
2. The outer query uses the number of orders as a category and counts the number of customers

Another Example. **Write an SQL query that returns the distribution of sector per company**





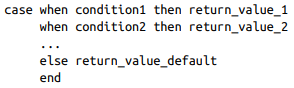




This type of profiling can be applied whenever you need to see how frequently certain entities or attributes appear in the data. In these examples, count has been used, but the other basic aggregations (sum, avg, min, and max) can be used to create histograms as well. For instance, we might want to profile customers by the sum of all their orders, their avg order size, their min order date, or their max (most recent) order date

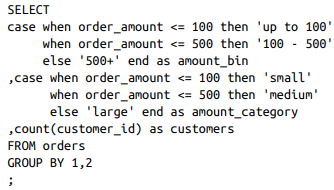
**Binning (grouping)**

Binning is useful when working with continuous values. Rather than the number of observations or records for each value being counted, **ranges of values are grouped together**, **and these groups are called bins or buckets**. The number of records that fall into each interval is then counted. Bins can be variable in size or have a fixed size, depending on whether your goal is to group the data into bins that have particular meaning for the organization, are roughly equal width, or contain roughly equal numbers of records. Bins can be created with CASE statements, rounding, and logarithms.



A CASE statement is a flexible way to control the number of bins, the range of values that fall into each bin, and how the bins are named. I find them particularly useful when there is a long tail of very small or very large values that I want to group together rather than have empty bins in part of the distribution. Certain ranges of values have a business meaning that needs to be re-created in the data. Many B2B companies separate their customers into “enterprise” and “SMB” (small- and medium-sized businesses) categories based on number of employees or revenue, because their buying patterns are different.

**As an example, imagine we are considering discounted shipping offers and we want to know how many customers will be affected. We can group order\_amount into three buckets using a CASE statement:**



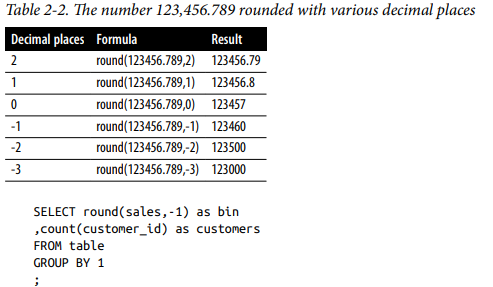
Тем самым мы создадим отдельные группы по которым после будет группировать

Arbitrary-sized bins can be useful, but at other times **bins of fixed size are more appropriate for the analysis**. Fixed-size bins can be accomplished in a few ways,

* Rounding
* Logarithms
* n-tiles

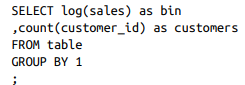
**Rounding**

To create equal-width bins, rounding is useful. Rounding reduces the precision of the values, and we usually think about rounding as reducing the number of decimal places or removing them altogether by rounding to the nearest integer



**Logarithms**

Logarithms are another way to create bins, particularly in data sets in which the largest values are orders of magnitude greater than the smallest values. The distribution of household wealth, the number of website visitors across different properties on the internet, and the shaking force of earthquakes are all examples of phenomena that have this property. While they don’t create bins of equal width, logarithms create bins that increase in size with a useful pattern



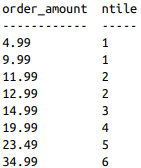
The log function can be used on any positive value. However, the logarithm function does not work when values can be less than or equal to 0; it will return null or an error, depending on the database.

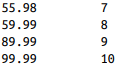
**N-tiles**

You’re probably familiar with the median, or middle value, of a data set. This is the 50th percentile value. Half of the values are larger than the median, and the other half are smaller. With quartiles, we fill in the 25th and 75th percentile values. A quarter of the values are smaller and three quarters are larger for the 25th percentile; three quarters are smaller and one quarter are larger at the 75th percentile. Deciles break the data set into 10 equal parts. Making this concept generic, n-tiles allow us to calculate any percentile of the data set: 27th percentile, 50.5th percentile, and so on.

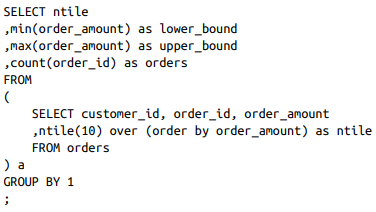


As an example, imagine we had 12 transactions with order\_amounts of $19.99, $9.99, $59.99, $11.99, $23.49, $55.98, $12.99, $99.99, $14.99, $34.99, $4.99, and $89.99. Per‐ forming an ntile calculation with 10 bins sorts each order\_amount and assigns a bin from 1 to 10:





This can be used to bin records in practice by first calculating the ntile of each row in a subquery and then wrapping it in an outer query that uses min and max to find the upper and lower boundaries of the value range:



Тем самым мы проставим ntile, а после в верхнем запросе сгруппируем по ntile и выведем для каждой группы минимум и максимум

**Data Quality**

Data quality is absolutely critical when it comes to creating good analysis. It’s easy to get overly focused on the mechanics of processing the data, finding clever query techniques and just the right visualization, only to have stakeholders ignore all of that and point out the one data inconsistency

**Comparing data against ground truth, or what is otherwise known to be true, is ideal though not always possible**. For example, if you are working with a replica of a pro‐ duction database, you could compare the row counts in each system to verify that all rows arrived in the replica database. In other cases, you might know the dollar value and count of sales in a particular month and thus can query for this information in the database to make sure the sum of sales and count of records match. Often the difference between your query results and the expected value comes down to whether you applied the correct filters, such as excluding cancelled orders or test accounts; how you handled nulls and spelling anomalies; and whether you set up correct JOIN conditions between tables.

**Profiling is a way to uncover data quality issues early on**, before they negatively impact results and conclusions drawn from the data. Profiling reveals

* Nulls
* categorical codings that need to be deciphered
* fields with multiple values that need to be parsed
* unusual datetime formats

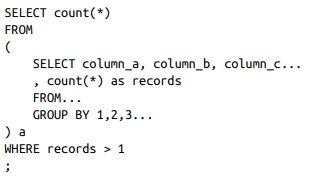
Profiling can also uncover gaps and step changes in the data that have resulted from tracking changes or outages. Data is rarely perfect, and it’s often only through its use in analysis that data quality issues are uncovered.

**Detecting Duplicates**

A duplicate is when you have two (or more) rows with the same information. Duplicates can exist for any number of reasons.

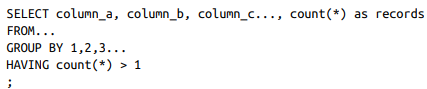
* A mistake might have been made during data entry, if there is some manual step.
* A tracking call might have fired twice.
* A processing step might have run multiple times. You might have created it accidentally with a hidden many-to-many JOIN.
* …

However they come to be, duplicates can really throw a wrench in your analysis.

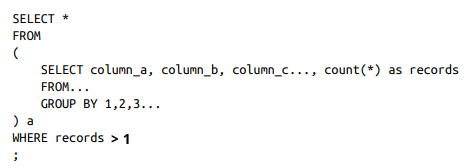


This will tell you whether there are any cases of duplicates. If the query returns 0, you’re good to go

As an alternative to a subquery, you can use a HAVING clause

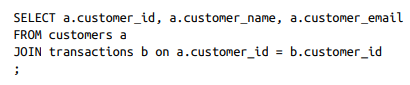


For full detail on which records have duplicates, you can list out all the fields and then use this information to chase down which records are problematic:

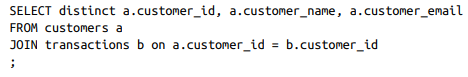


**Deduplication with GROUP BY and DISTINCT**

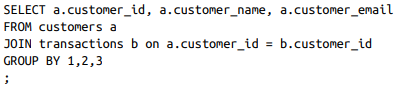
Duplicates happen, and they’re not always a result of bad data. For example, imagine we want to find a list of all the customers who have successfully completed a transaction so we can send them a coupon for their next order. We might JOIN the customers table to the transactions table, which would restrict the records returned to only those customers that appear in the transactions table:



This will return a row for each customer for each transaction, however, and **there are** hopefully **at least a few customers who have transacted more than once**. **We have accidentally created duplicates**, not because there is any underlying data quality problem but because we haven’t taken care to avoid duplication in the results. Fortunately, there are several ways to avoid this with SQL. One way to remove duplicates is to use the keyword DISTINCT:



Another option is to use a GROUP BY, which, although typically seen in connection with an aggregation, will also deduplicate in the same way as DISTINCT



Duplicate data, or data that contains multiple records per entity even if they techni‐ cally are not duplicates, is one of the most common reasons for incorrect query results. You can suspect duplicates as the cause if all of a sudden the number of cus‐ tomers or total sales returned by a query is many times greater than what you were expecting

**Data Cleaning**

Profiling often reveals where changes can make the data more useful for analysis. Some of the steps are

* CASE transformations
* adjusting for null
* changing data types.

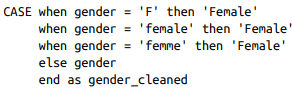
**Cleaning Data with CASE Transformations**

CASE statements can be used to perform a variety of cleaning, enrichment, and summarization tasks. Sometimes the data exists and is accurate, but it would be more useful for analysis if values were standardized or grouped into categories.

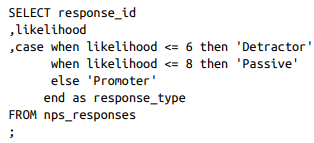
Nonstandard values occur for a variety of reasons:

* Values might come from different systems with slightly different lists of choices
* System code might have changed
* Options might have been presented to the customer in different languages
* The customer might have been able to fill out the value rather than pick from a list.

Imagine a field containing information about the gender of a person. Values indicating a female person exist as “F,” “female,” and “femme.” **We can standardize the values** like this:

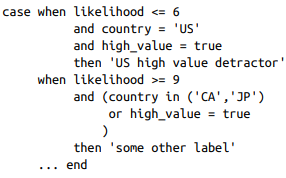


**CASE statements can also be used to add categorization or enrichment that does not exist in the original data**. As an example, many organizations use a Net Promoter Score, or NPS, to monitor customer sentiment. NPS surveys ask respondents to rate, on a scale of 0 to 10, how likely they are to recommend a company or product to a friend or colleague. Scores of 0 to 6 are considered detractors, 7 and 8 are passive, and 9 and 10 are promoters. The final score is calculated by subtracting the percentage of detractors from the percentage of promoters. Survey result data sets usually include optional free text comments and are sometimes enriched with information the organization knows about the person surveyed. Given a data set of NPS survey responses, the first step is to group the responses into the categories of detractor, passive, and promoter:



Note that the data type can differ between the field being evaluated and the return data type.

**CASE statements can consider multiple columns and can contain AND/OR logic**. They can also be nested, though often this can be avoided with AND/OR logic:



Cleaning or enriching data with a CASE statement works well as long as there is a relatively short list of variations, you can find them all in the data, and the list of val‐ ues isn’t expected to change. For longer lists and ones that change frequently, a lookup table can be a better option

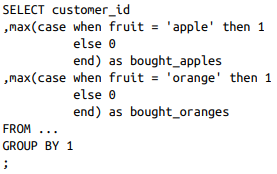
**Another useful thing you can do with CASE statements is to create flags indicating whether a certain value is present, without returning the actual value**

This can be useful during profiling for understanding how common the existence of a particular attribute is. Another use for flagging is during preparation of a data set for statistical analysis. In this case, a flag is also known as a dummy variable, taking a value of 0 or 1 and indicating the presence or absence of some qualitative variable. For example, we can create is\_female and is\_promoter flags with CASE statements on gender and likelihood (to recommend) fields:



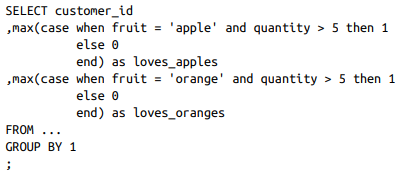


If you are working with a data set that has multiple rows per entity, such as with line items in an order, **you can flatten the data with a CASE statement wrapped in an aggregate and turn it into a flag at the same time by using 1 and 0 as the return value**. We saw previously that a BOOLEAN data type is often used to create flags (fields that represent the presence or absence of some attribute). Here, 1 is substituted for TRUE and 0 is substituted for FALSE so that a max aggregation can be applied. The way this works is that for each customer, the CASE statement returns 1 for any row with a fruit type of “apple.” Then max is evaluated and will return the largest value from any of the rows. As long as a customer bought an apple at least once, the flag will be 1; if not, it will be 0



Тут каждый покупатель группируется, и если он купил яблоки, то MAX() выведет 1, если не купил, то будет 0. То же самое и с апельсинами. Тут покажет покупал ли покупатель вообще яблоки или апельсины, так как для 1 покупателя может быть несколько записей с его покупками

**You can also construct more complex conditions for flags, such as requiring a threshold or amount of something before labeling with a value of 1:**



**Type Conversions and Casting**

Every field in a database is defined with a data type. When data is inserted into a table, values that aren’t of the field’s type are rejected by the database. Strings can’t be inserted into integer fields, and booleans are not allowed in date fields. Occasionally, however, we need to override the data type of the field and force it to be something else.

**Type conversion functions** allow pieces of data with the appropriate format to be changed from one data type to another.

There are 2 options. One way to change the data type is with the cast function, **cast (input as data\_type)**, or two colons, **input :: data\_type**. Both of these are equivalent and convert the integer 1,234 to a string



Converting an integer to a string can be useful in CASE statements when categorizing numeric values with some unbounded upper or lower value. For example, **in the following code, leaving the values that are less than or equal to 3 as integers while returning the string “4+” for higher values would result in an error:**



Casting the integers to the VARCHAR type solves the problem:

