CHAPTER 1

A Data Miner Looks at SQL

Data is being collected everywhere. Every transaction, every web page visit, every payment—and much more—is filling databases, relational and otherwise, with raw data. Computing power and storage have grown to be cost effective, a trend where today's smart phones are more powerful than supercomputers of yesteryear. Databases are no longer merely platforms for storing data; they are powerful engines for transforming data into useful information about customers and products and business practices.

The focus on data mining has historically been on complex algorithms developed by statisticians and machine-learning specialists. Once upon a time, data mining required downloading source code from a research lab or university, compiling the code to get it to run, and sometimes even debugging it. By the time the data and software were ready, the business problem had lost urgency.

This book takes a different approach because it starts with the data. The billions of transactions that occur every day—credit cards swipes, web page visits, telephone calls, and so on—are now often stored in relational databases. Relational database engines count among the most powerful and sophisticated software products in the business world, so they are well suited for the task of extracting useful information. And the lingua franca of relational databases is SQL.

The focus of this book is more on data and what to do with data and less on theory. Instead of trying to squeeze every last iota of information from a small sample—the goal of much statistical analysis—the goal is instead to find something useful in the gigabytes and terabytes of data stored by the business. Instead of asking programmers to learn data analysis, the goal

is to give data analysts—and others—a solid foundation for using SQL to learn from data.

This book strives to assist anyone facing the problem of analyzing data stored in large databases, by describing the power of data analysis using SQL and Excel. SQL, which stands for Structured Query Language, is a language for extracting information from data. Excel is a popular and useful spreadsheet for analyzing smaller amounts of data and presenting results.

The various chapters of this book build skill in and enthusiasm for SQL queries and the graphical presentation of results. Throughout the book, the SQL queries are used for more and more sophisticated types of analyses, starting with basic summaries of tables, and moving to data exploration. The chapters continue with methods for understanding time-to-event problems, such as when customers stop, and market basket analysis for understanding what customers are purchasing. Data analysis is often about building models, and—perhaps surprisingly to most readers—some models can be built directly in SQL, as described in Chapter 11, "Data Mining in SQL." An important part of any analysis, though, is constructing the data in a format suitable for modeling—customer signatures.

The final chapter takes a step back from analysis to discuss performance. This chapter is an overview of a topic, concentrating on good performance practices that work across different databases.

This chapter introduces SQL for data analysis and data mining. Admittedly, this introduction is heavily biased because the purpose is for querying databases rather than building and managing them. SQL is presented from three different perspectives, some of which may resonate more strongly with different groups of readers. The first perspective is the structure of the data, with a particular emphasis on entity-relationship diagrams. The second is the processing of data using dataflows, which happen to be what is "under the hood" of most relational database engines. The third, and strongest thread through subsequent chapters, is the syntax of SQL itself. Although data is well described by entities and relationships, and processing by dataflows, the ultimate goal is to express the transformations in SQL and present the results often through Excel.

Databases, SQL, and Big Data

Collecting and analyzing data is a major activity, so many tools are available for this purpose. Some of these focus on "big data" (whatever that might mean). Some focus on consistently storing the data quickly. Some on deep analysis. Some have pretty visual interfaces; others are programming languages.

SQL and relational databases are a powerful combination that is useful in any arsenal of tools for analysis, particularly ad hoc analyses:

- A mature and standardized language for accessing data
- Multiple vendors, including open source
- Scalability over a very broad range of hardware
- A non-programming interface for data manipulations

Before continuing with SQL, it is worth looking at SQL in the context of other tools.

What Is Big Data?

Big data is one of those concepts whose definition changes over time. In the 1800s, when statistics was first being invented, researchers worked with dozens or hundreds of rows of data. That might not seem like a lot, but if you have to add everything up with a pencil and paper, and do long division by hand or using a slide rule, then it certainly seems like a lot of data.

The concept of big data has always been relative, at least since data processing was invented. The difference is that now data is measured in gigabytes and terabytes—enough bytes to fit the text in all the books in the Library of Congress—and we can readily carry it around with us. The good news is that analyzing "big data" no longer requires trying to get data to fit into very limited amounts of memory. The bad news is that simply scrolling through "big data" is not sufficient to really understand it.

This book does not attempt to define "big data." Relational databases definitely scale well into the tens of terabytes of data—big by anyone's definition. They also work efficiently on smaller datasets, such as the ones accompanying this book.

Relational Databases

Relational databases, which were invented in the 1970s, are now the storehouse of mountains of data available to businesses. To a large extent, the popularity of relational databases rests on what are called ACID properties of transactions:

- Atomicity
- Consistency
- Isolation
- Durability

These properties basically mean that when data is stored or updated in a database, it really is changed. The databases have transaction logs and other capabilities to ensure that changes really do happen and that modified data is visible when the data modification step completes. (The data should even survive major failures such as the operating system crashing.) In practice, databases support

transactions, logs, replication, concurrent access, stored procedures, security, and a host of features suitable for designing real-world applications.

From our perspective, a more important attribute of relational databases is their ability to take advantage of the hardware they are running on—multiple processors, memory, and disk. When you run a query, the *optimization engine* first translates the SQL query into the appropriate lower-level algorithms that exploit the available resources. The optimization engine is one of the reasons why SQL is so powerful: the same query running on a slightly different machine or slightly different data might have very different execution plans. The SQL remains the same; it is the optimization engine that chooses the best way to execute the code.

Hadoop and Hive

One of the technologies highly associated with big data is Hadoop in conjunction with MapReduce. Hadoop is an open-source project, meaning that the code is available for free online, with the goal of developing a framework for "reliable, scalable, distributed computing." (The SQL world has free open-source databases such as MySQL, Postgres, and SQLite; in addition, several commercial databases have free versions.) In practice, Hadoop is a platform for processing humongous amounts of data, particularly data from sources such as web logs, high-energy physics, high volumes of streaming images, and other voluminous data sources.

The roots of MapReduce go back to the 1960s and a language called Lisp. In the late 1990s, Google developed a parallel framework around MapReduce, and now it is a framework for programming data-intensive tasks on large grid computers. It became popular because both Google and Yahoo developed MapReduce engines; and, what big successful internet companies do must be interesting.

Hadoop actually has a family of technologies and MapReduce is only one application. Built on Hadoop are other tools, all with colorful names such as Hive, Mahout, Cassandra, and Pig. Although the underlying technology is different from relational databases, there are similarities in the problems these technologies are trying to solve. Within the Hadoop world are languages, such CQL, which is based on SQL syntax. Hive, in particular, is being developed into a fully functional SQL engine and can run many of the queries in this book.

NoSQL and Other Types of Databases

NoSQL refers to a type of database that, at first sight, might seem to be the antithesis of SQL. Actually, the "No" stands for "Not Only." This terminology can be used to refer to a variety of different database technologies:

- Key-value pairs, where the columns of data can vary between rows—and, quite importantly—the columns themselves can contain lists of things

- Graph-based databases, which specialize in representing and handling problems from graph theory
- Document databases, which are used for analyzing documents and other texts
- Geographic information systems (GIS), which are used for geographic analysis

These types of databases are often specialized for particular functions. For instance, key-value pair databases provide excellent performance in a web environment for managing data about online sessions.

These technologies are really complementary technologies to traditional relational databases rather than replacement technologies. For instance, key-value databases are often used on a website in conjunction with relational databases that store history. Graph and document databases are often used in conjunction with data warehouses that support more structured information.

Further, good ideas are not limited to a single technology. One of the motivations for writing a second edition of this book is that database technology is improving. SQL and the underlying relational database technology increasingly support functionality similar to NoSQL databases. For example, recursive common table expressions provide functionality for traversing graphs. Full text indexes provide functionality for working with text. Most databases offer extensions for geographic data. And, increasingly databases are providing better functionality for nested tables and portable data formats, such as XML and JSON.

SQL

SQL was designed to work on structured data—think tables with well-defined columns and rows, much like an Excel spreadsheet. Much of the power of SQL comes from the power of the underlying database engine and the optimizer. Many people use databases running on powerful computers, without ever thinking about the underlying hardware. That is the power of SQL: The same query that runs on a mobile device can run on the largest grid computer, taking advantage of all available hardware with no changes to the query.

The part of the SQL language used for analysis is the `SELECT` statement. Much of the rest of the language is about getting data *in to* databases. Our concern is getting information *out of* them to solve business problems. The `SELECT` statement describes what the results look like, freeing the analyst to think about *what* to do, instead of *how* to do it.

TIP **SQL (when used for querying) is a *descriptive* language rather than a *procedural* language. It describes what needs to be done, letting the SQL engine optimize the code for the particular data, hardware, and database layout where the query is running, and freeing the analyst to think more about the business problem.**

Picturing the Structure of the Data

In the beginning, there is data. Although data may seem to be without form and chaotic, there is an organization to it, an organization based on tables and columns and relationships between and among them. Relational databases store *structured* data—that is, tables with well-defined rows and columns.

This section describes databases by the data they contain. It introduces *entity-relationship diagrams*, in the context of the datasets (and associated data models) used with this book. These datasets are not intended to represent all the myriad different ways that data might be stored in databases; instead, they are intended as practice data for the ideas in the book. They are available on the companion website, along with all the examples in the book.

What Is a Data Model?

The definition of the tables, the columns, and the relationships among them constitute the *data model* for the database. A well-designed database actually has two data models. The *logical data model* explains the database in terms that business users understand. The logical data model communicates the contents of the database because it defines many business terms and how they are stored in the database.

The *physical data model* explains how the database is actually implemented. In many cases, the physical data model is identical to or very similar to the logical data model. That is, every entity in the logical data model corresponds to a table in the database; every attribute corresponds to a column. This is true for the datasets used in this book.

On the other hand, the logical and physical data models can differ. For instance, in more complicated databases, certain performance issues might drive physical database design. A single entity might have rows split into several tables to improve performance, enhance security, enable backup-restore functionality, or facilitate database replication. Multiple similar entities might be combined into a single table, especially when they have many attributes in common. Or, a single entity could have different columns in different tables, with the most commonly used columns in one table and less commonly used ones in another table (this is called *vertical partitioning*, which some databases support directly without having to resort to multiple tables). Often these differences are masked through the use of *views* and other database constructs.

The logical model is quite important for analytic purposes because it provides an understanding of the data from the business perspective. However, queries actually run on the database represented by the physical model, so it is convenient that the logical and physical structures are often quite similar.

What Is a Table?

A table is a set of rows and columns that describe multiple instances of something. Each row represents one instance—such as a single purchase made by a customer, or a single visit to a web page, or a single zip code with its demographic details. Each column contains one attribute for one instance. SQL tables represent unordered sets, so the table does not have a first row or a last row—unless a specific column such as an id or creation date provides that information.

Any given column contains the same genre of information for all rows. So a zip code column should not be the "sent-to" zip code in one row and the "billed-to" zip code in another. Although these are both zip codes, they represent two different uses, so they belong in two different columns.

Columns, unless declared otherwise, are permitted to take on the value `NULL`, meaning that the value is not available or is unknown. For instance, a row describing customers might contain a column for birthdate. This column would take on the value of `NULL` for all rows where the birthdate is not known.

A table can have as many columns as needed to describe an instance, although for practical purposes tables with more than a few hundred columns are rare (and most relational databases do have an upper limit on the number of columns in a single table, typically in the low thousands). A table can have as many rows as needed; here the numbers easily rise to the millions and even billions.

As an example, Table 1-1 shows a few rows and columns from `ZipCensus` (which is available on the companion website). This table shows that each zip code is assigned to a particular state, which is the abbreviation in the `stab` column ("`ST`ate `AB`breviation") . The `pctstate` column is an indicator that zip codes sometimes span state boundaries. For instance, 10004 is a zip code in New York City that covers Ellis Island. In 1998, the Supreme Court split jurisdiction of the island between New York and New Jersey, but the Post Office did not change the zip code. So, 10004 has a portion in New York and a smaller, unpopulated portion in New Jersey.

Each zip code also has an area, measured in square miles and recorded in the `landsqmi` column. This column contains a number, and the database does not

Table 1-1: Some Rows and Columns from ZipCensus

ZCTA5	STAB	PCTSTATE	TOTPOP	LANDSQMI
10004	NY	100%	2,780	0.56
33156	FL	100%	31,537	13.57
48706	MI	100%	40,144	66.99
55403	MN	100%	14,489	1.37
73501	OK	100%	19,794	117.34
92264	CA	100%	20,397	52.28

know what this number means. It could be area in acres, or square kilometers, or square inches, or pyongs (a Korean unit for area). What the number really means depends on information not stored in the tables. The term *metadata* is used to describe such information about what the values in columns mean. Similarly, `fipco` is a numeric value that encodes the state and county, with the smallest value being 1001, for Alabaster County in Alabama.

Databases typically have some metadata information about each column. Conveniently, there is often a label or description (and it is a good idea to fill this in when creating a table). More importantly, each column has a data type and a flag specifying whether `NULL` values are allowed. The next two sections discuss these two topics because they are quite important for analyzing data.

Allowing NULL Values

Nullability is whether or not a column may contain the `NULL` value. By default in SQL, a column in any row can contain a special value that says that the value is empty or unknown. Although this is quite useful, `NULL`s have unexpected side effects. Almost every comparison returns "unknown" if any argument is `NULL`, and "unknown" is treated as false.

The following very simple query looks like it is counting all the rows in the `ZipCensus` table where the `FIPCo` column is not `NULL`. (`<>` is the SQL operator for "not equals.")

```
SELECT COUNT(*)
FROM ZipCensus zc
WHERE zc.fipco <> NULL
```

Alas, this query always returns zero. When a `NULL` value is involved in a comparison—even "not equals"—the result is almost always `NULL`, which is treated as false.

Of course, determining which rows have `NULL` values is quite useful, so SQL provides the special operators `IS NULL` and `IS NOT NULL`. These behave as expected, with the preceding query returning 32,845 instead of 0.

The problem is more insidious when comparing column values, either within a single table or between tables. For instance, the column `fipco` contains the primary county of a zip code and `fipco2` contains the second county, if any. The following query counts the number of zip codes in total and the number where these two county columns have different values. This query uses conditional aggregation, which is when a conditional statement (`CASE`) is the argument to an aggregation function such as `SUM()`:

```
SELECT COUNT(*),
       SUM(CASE WHEN fipco <> fipco2 THEN 1 ELSE 0 END) as numsame
FROM ZipCensus zc
```

Or does it? The columns `fipco` and `fipco2` should always have different values, so the two counts should be the same. In reality, the query returns the values 32,989 and 8,904. And changing the not-equals to equals shows that there are 0 rows where the values are equal. What is happening on the other 32,989 – 8,904 rows? Once again, the "problem" is `NULL` values. When `fipco2` is `NULL`, the test always fails.

When a table is created, there is the option to allow `NULL` values on each column in the table. This is a relatively minor decision when creating the table. However, making mistakes on columns with `NULL` values is easy.

WARNING **Designing databases is different from analyzing the data inside them. For example, `NULL` columns can cause unexpected—and inaccurate—results when analyzing data and make reading queries difficult. Be very careful when using columns that allow them.**

`NULL` values may seem troublesome, but they solve an important problem: how to represent values that are not present. One alternative method is to use a special value, such as -99 or 0. However, the database would just treat this as a regular value, so calculations (such as `MIN()`, `MAX()`, and `SUM()`) would be incorrect.

Another alternative would be to have separate flags indicating whether or not a value is `NULL`. That would make even simple calculations cumbersome. "A + B", for instance, would have to be written as something like "`(CASE WHEN A_flag = 1 AND B_flag = 1 THEN A + B END)`". Given the alternatives, having `NULL`s in the database is a practical approach to handling missing values.

Column Types

The second important attribute of a column is its type, which tells the database exactly how to store values. A well-designed database usually has parsimonious columns, so if two characters suffice for a code, there is no reason to store eight. There are a few important aspects of column types and the roles that columns play.

Primary key columns uniquely identify each row in the table. That is, no two rows have the same value for the primary key and the primary key is never `NULL`. Databases guarantee that primary keys are unique by refusing to insert rows with duplicate primary keys. Chapter 2, "What's in a Table? Getting Started with Data Exploration," shows techniques to determine whether this condition holds for any given column. Typically the primary key is a single column, although SQL does allow *composite* primary keys, which consist of multiple columns.

Numeric values are values that support arithmetic and other mathematical operations. In SQL, these can be stored in different ways, such as floating-point numbers, integers, and decimals. The details of how these formats differ are much less important than what can be done with numeric data types.

Within the category of numeric types, one big difference is between integers, which have no fractional part, and real numbers, which do. When doing arithmetic on integers, the result might be an integer or it might be a real number, depending on the database. So 5/2 might evaluate to 2 rather than 2.5, and the average of 1 and 2 might turn out to be 1 instead of 1.5, depending on the database. To avoid this problem, examples in this book multiply integer values by 1.0 to convert them to decimal values when necessary.

Of course, just because it walks like a duck and talks like a duck does not mean that it is a duck. Some values look like numbers, but really are not. Zip codes (in the United States) are an example, as are primary key columns stored as numbers. What is the sum of two zip codes? What does it mean to multiply a primary key value by 2? These questions yield nonsense results (although the values can be calculated). Zip codes and primary keys happen to look like numbers, but they do not behave like numbers.

The datasets used in this book use character strings for zip codes and numbers for primary keys. To distinguish such false numeric values from real numbers, the values are often left padded with zeros to get a fixed length. After all, the zip code for Harvard Square in Cambridge, MA, is 02138, not 2,138.

Dates and *date-times* are exactly what their names imply. SQL provides several functions for common operations, such as determining the number of days between two dates, extracting the year and month, and comparing two times. Unfortunately, these functions often differ between databases. The Appendix provides a list of equivalent functions in different databases for functions used in this book, including date and time functions.

Another type of data is character string data. These are commonly codes, such as the state abbreviation in the zip code table, or a description of something, such as a product name or the full state name. SQL has some very rudimentary functions for handling character strings, which in turn support rudimentary text processing. Spaces at the end of a character string are ignored, so the condition `'NY' = 'NY '` evaluates to TRUE. However, spaces at the beginning of a character string are counted, so `'NY' = ' NY'` evaluates to FALSE. When working with data in character columns, it might be worth checking out whether there are spaces at the beginning, a topic discussed in Chapter 2.

What Is an Entity-Relationship Diagram?

The "relational" in the name "relational databases" refers to the fact that different tables relate to each other via keys, and to the fact that columns in a given row relate to the values for that column via the column name. For instance, a zip code column in any table can link (that is "relate") to the zip code table. The key makes it possible to look up information available in the zip code table. Figure 1-1 shows the relationships between tables in the purchases dataset.

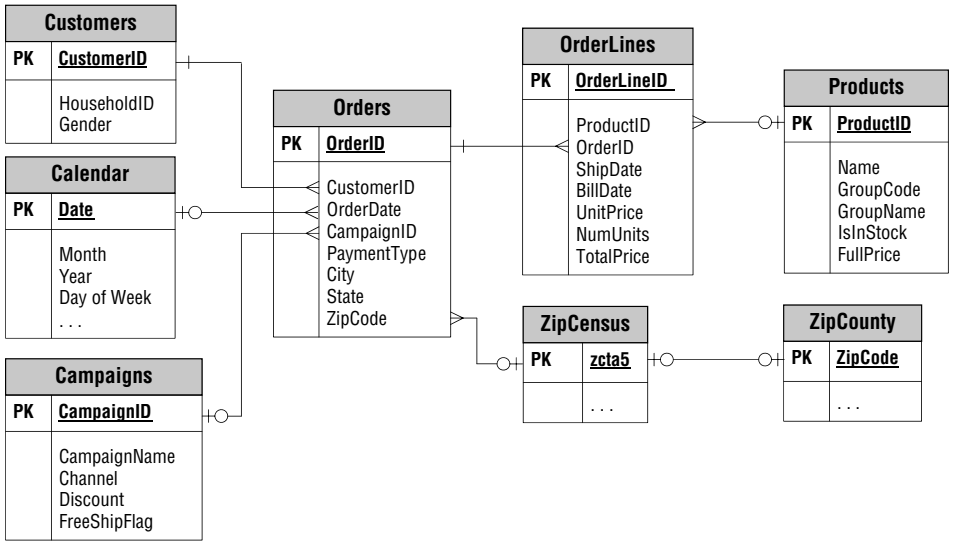


Figure 1-1: This entity-relationship diagram shows the relationship among entities in the purchase dataset. Each entity corresponds to one table.

These relationships have a characteristic called *cardinality*, which is the number of items related on each side. For instance, the relationship between `Orders` and `ZipCensus` is a zero/one-to-many relationship. This specifies every row in `Orders` has at most one zip code. And, every zip code has zero, one, or more orders. Typically, this relationship is implemented by having a column in the first table contain the zip code, which is called a *foreign key*. A foreign key is just a column whose contents are the primary key of another table (`ZipCode` in `Orders` is a foreign key; `zcta5` in `ZipCensus` is a primary key). To indicate no match, the foreign key column would typically be `NULL`.

The zero/one-to-one relationship says that there is at most one match between two tables. This is often a subsetting relationship. For instance, a database might contain sessions of web visits, some of which result in a purchase. Any given session would have zero or one purchases. Any given purchase would have exactly one session.

Another relationship is a many-to-many relationship. A customer might purchase many different products and any given product might be purchased by many different customers. In fact, the purchase dataset does have a many-to-many relationship between `Orders` and `Products`; this relationship is represented by the `OrderLines` entity, which has a zero/one-to-many relationship with each of those.

An example of the one-at-a-time relationship is a customer who resides in a particular zip code. The customer might move over time. Or, at any given time, a customer might have a particular handset or billing plan, but these can change over time.

With this brief introduction to entity-relationship diagrams, the following sections describe the datasets used in this book.

The Zip Code Tables

The `ZipCensus` table consists of more than one hundred columns describing each zip code, or, strictly speaking, each zip code tabulation area (ZCTA) defined by the Census Bureau. The column `zcta5` is the zip code. This information was gathered from the Missouri Census Data Center, based on US Census data, specifically the American Community Survey.

The first few columns consist of overview information about each zip code, such as the state, the county, population (`totpop`), latitude, and longitude. There is a column for additional zip codes because the zip-code tabulation area does not necessarily match 100% with actual zip codes. In addition to population, there are four more counts: the number of households (`tothhs`), the number of families (`famhhs`), the number of housing units (`tothus`), and the number of occupied housing units (`occhus`).

The following information is available for the general population:

- Proportion and counts in various age groups
- Proportion and counts by gender
- Proportion and counts in various racial categories
- Proportion and counts of households and families by income
- Information about occupation categories and income sources
- Information about marital status
- Information about educational attainment
- And more

Information on the columns and exact definitions of terms such as ZCTA are available at http://mcdc.missouri.edu/data/georef/zcta_master.Metadata.html.

The second zip code table is `ZipCounty`, a companion table that maps zip codes to counties. It contains information such as the following:

- County name
- Post office name
- Population of county
- Number of households in county
- County land area

This table has one row for each zip code, so it can be joined to `ZipCensus` and to other tables using the `ZipCode` column. The two tables are from different

time frames and sources so not all zip codes match between the two tables—a common problem when working with data.

Subscription Dataset

The subscription data has only two entities, shown in Figure 1-2. This dataset paints a picture of a subscriber at a given point in time (the date when the snapshot was created).

The `Subscribers` table describes customers in a telephone company. It is a snapshot that shows what customers (and former customers) look like as of a particular date. The columns in this table describe customers as they start and as they stop. This particular snapshot table has no intermediate behavior information.

The `Calendar` table is a general-purpose table that has information about dates, including:

- Year
- Month number
- Month name
- Day of month
- Day of week
- Day of year
- Holiday information

This table has the date as a primary key, and covers dates from 1950 through 2050.

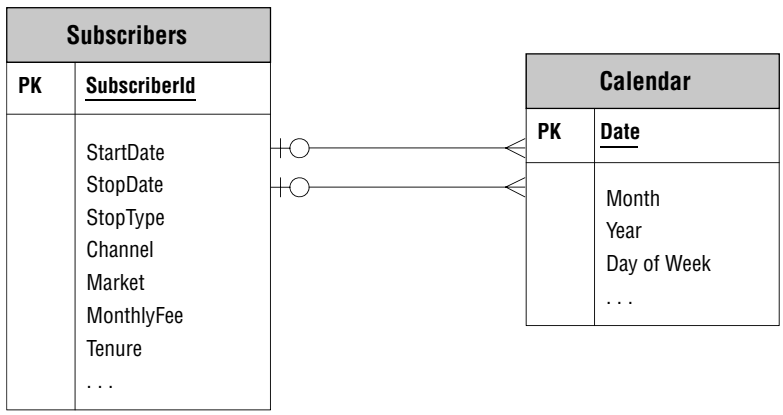


Figure 1-2: An entity-relationship diagram with only two entities describes the data in the customer snapshot dataset.

Purchases Dataset

The purchases dataset contains entities typical of retail purchases; the entities in this dataset and their relationships are shown in Figure 1-1 (page 11) :

- `Customers`
- `Orders`
- `OrderLines`
- `Products`
- `Campaigns`
- `ZipCensus`
- `ZipCounty`
- `Calendar`

This data captures the important entities associated with retail purchases. The most detailed information is in `OrderLines`, which describes each of the items in an order. To understand the name of the table, think of a receipt. Each line on the receipt represents a different item in the purchase. In addition, the line has other information such as the product id, the price, and the number of items, which are all in this table.

The `Products` table provides information such as the product group name and the full price of a product. The table does not contain detailed product names. These were removed as part of the effort to anonymize the data.

To tie all the items in a single purchase together, each row of `OrderLines` has an `OrderId`. Each `OrderId`, in turn, represents one row in the `Orders` table, which has information such as the date and time of the purchase, where the order was shipped to, and the type of payment. It also contains the total dollar amount of the purchase, summed up from the individual items. Each order line is in exactly one order and each order has one or more order lines. This relationship is described as a one-to-many relationship between these tables.

Just as the `OrderId` ties multiple order lines into an order, the `CustomerId` assigns orders made at different points in time to the same customer. The existence of the `CustomerId` prompts the question of how it is created. In one sense, it makes no difference how it is created; the `CustomerId` is simply a given, defining the customer in the database. Is it is doing a good job? That is, are a single customer's purchases being tied together most of the time? The aside "The Customer ID: Identifying Customers Over Time," discusses the creation of customer IDs.

Tips on Naming Things

The datasets provided with this book have various original sources, so they have different naming conventions. In general, there are some things that should always be avoided and some things that are good practice:

THE CUSTOMER ID: IDENTIFYING CUSTOMERS OVER TIME

The `CustomerId` column combines transactions over time into a single grouping, the customer (or household or similar entity). How is this accomplished? It all depends on the business and the business processes:

- **The purchases might contain name and address information. So, purchases with matching names and addresses would have the same customer ID.**
- **The purchases might all have telephone numbers or email address, so these could provide the customer ID.**
- **Customers may have loyalty cards or account numbers which provide the customer ID.**
- **The purchases might be on the web, so browser cookies and logins could identify customers over time.**
- **The purchases might all be by credit card, so purchases with the same credit card number would have the same customer ID.**

Of course, any combination of these or other methods might be used to generate an internal customer id. And, because any one of these ids could change over time, the problem has a time component as well.

And all these approaches have their challenges. What happens when a customer browses on a tablet as well as a laptop (and different cookies are stored on different machines) or deletes her web cookies? Or when customers forget their loyalty cards (so the loyalty numbers are not attached to the purchases)? Or move? Or change phone numbers or email addresses? Or change their names? Keeping track of customers over time can be challenging.

- Always use only alphanumeric characters and underscores for table and column names. Other characters, such as spaces, require that the name be escaped when referenced. The escape characters, typically double quotes or square braces, make it hard to write and read queries.
- Never use SQL reserved words. Databases have their own special words, but words like `Order`, `Group`, and `Values` are keywords in the language and should be avoided.

Additional good practices include the following:

- Table names are usually in plural (this helps avoid the problem with reserved words) and reinforces the idea that tables contain multiple instances of the entity.
- The primary key is the singular of the table name followed by "Id." Hence, `OrderId` and `SubscriberId`. When a column references another table such as the `OrderId` column in `OrderLines` (a *foreign key relationship*) use the exact same name, making it easy to see relationships between tables.

- "CamelBack" case is used (upper case for each new word, lowercase for the rest). Hence, `OrderId` instead of `Order_Id`. In general, table names and column names are not case sensitive. The CamelBack method is to make it easier to read the name, while at the same time keeping the name shorter (than if using underscores).
- The underscore is used for grouping common columns together. For instance, in the `Calendar` table, the indicators for holidays for specific religions start with `hol_`.

Of course, the most important practice is to make the column and table names understandable and consistent, so you (and others) recognize what they mean.

Picturing Data Analysis Using Dataflows

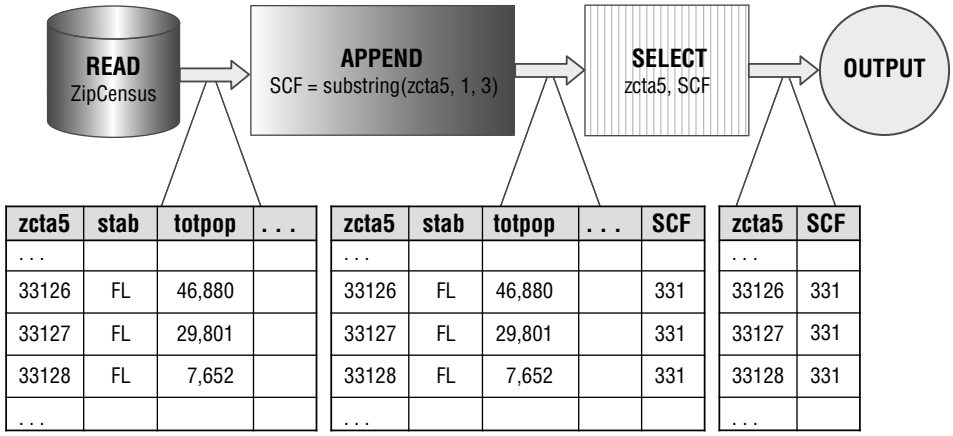
Tables store data, but tables do not actually do anything. Tables are nouns; queries are verbs. This book mates SQL and Excel for data manipulation, transformation, and presentation. The differences between these tools are exacerbated because they often support the same operations, although in very different ways. For instance, SQL uses the `GROUP BY` clause to summarize data in groups. An Excel user, on the other hand, might use pivot tables, use the subtotal wizard, or manually do calculations using functions such as `SUMIF()`; however, nothing in Excel is called "group by."

Because this book intends to combine the two technologies, it is useful to have a common way of expressing data manipulations and data transformations, a common language independent of the tools being used. Dataflows provide this common language by showing the transformation operations fitting together like an architecture blueprint for data processing, a blueprint that describes what needs to be done, without saying which tool is going to do the work. This makes dataflows a powerful mechanism for thinking about data transformations.

What Is a Dataflow?

A dataflow is a graphical way of visualizing data transformations. Dataflows have two important elements. The *nodes* in a dataflow diagram transform data, taking zero or more inputs and producing output. The *edges* in a dataflow diagram are pipes connecting the nodes. Think of the data flowing through the pipes and getting banged and pushed and pulled and flattened into shape by the nodes. In the end, the data has been transformed into information.

Figure 1-3 shows a simple dataflow that adds a new column, called SCF for Sectional Center Facility (something the U.S. Post Office uses to route mail).



zcta5	stab	totpop	. . .
. . .			
33126	FL	46,880	
33127	FL	29,801	
33128	FL	7,652	
. . .			

zcta5	stab	totpop	. . .	SCF
. . .				
33126	FL	46,880		331
33127	FL	29,801		331
33128	FL	7,652		331
. . .				

zcta5	SCF
. . .	
33126	331
33127	331
33128	331
. . .	

Figure 1-3: A simple dataflow reads the ZIPCODE, calculates and appends a new field called SCF, and outputs the SCF and ZIPCODE.

This column is the first three digits of a zip code. The output is each zip code with its SCF. The dataflow has four nodes, connected by three edges. The first, shaped like a cylinder, represents a database table or file and is the source of the data. The edge leaving this node shows some of the records being passed from it, records from the `ZipCensus` table.

The second node appends the new column to the table, which is also visible along the edge leading out from the node. The third selects two columns for output—in this case, `zcta5` and `SCF`. And the final node simply represents the output. On the dataflow diagram, imagine a magnifying glass that makes it possible to see the data moving through the flow. Seeing the data move from node to node shows what is happening in the flow.

The actual processing could be implemented in either SQL or Excel. The SQL code corresponding to this dataflow is:

```
SELECT zc.zcta5, LEFT(zc.zcta5, 3) as scf
FROM ZipCensus zc
```

Alternatively, if the data were in an Excel worksheet with the zip codes in column A, the following formula would extract the SCF:

```
=MID(A1, 1, 3)
```

Of course, the formula would have to be copied down the column.

Excel, SQL, and dataflows are three different ways of expressing similar transformations. The advantage of dataflows is that they provide an intuitive way of visualizing and thinking about data manipulations, independent of the

tool used for the processing. Dataflows facilitate understanding, but in the end, the work described in this book is in SQL or Excel.

TIP **When column A has a column of data and we want to copy a formula down column B, the following is a handy method based on keyboard shortcuts:**

1. **Type the formula in the first cell in column B where there is data in column A.**
2. **Move the cursor to column A.**
3. **Hit Ctrl+down arrow to go to the end of the data in column A (Command+down arrow on a Mac)**
4. **Hit the right arrow to move to column B.**
5. **Hit Ctrl+Shift+up arrow to highlight all of column B (Command+up arrow on a Mac).**
6. **Hit Ctrl+D to copy the formula down the column.**

Voila! The formula gets copied without a lot of fiddling with the mouse and with menus.

READ: Reading a Database Table

The READ operator reads all the columns of data from a database table or file. In SQL, this operation is implicit when tables are included in the `FROM` clause of a query. The READ operator does not accept any input dataflows, but has an output. Generally, if a table is needed more than once in a dataflow, each occurrence has a separate READ.

OUTPUT: Outputting a Table (or Chart)

The OUTPUT operator creates desired output, such as a table in a row-column format or some sort of chart based on the data. The OUTPUT operator does not have any outputs, but accepts inputs. It also accepts parameters describing the type of output.

SELECT: Selecting Various Columns in the Table

The SELECT operator chooses one or more columns from the input and passes them to the output. It might reorder columns and/or choose a subset of them. The SELECT operator has one input and one output. It accepts parameters describing the columns to keep and their order.

FILTER: Filtering Rows Based on a Condition

The FILTER operator chooses rows based on a TRUE or FALSE condition. Only rows that satisfy the condition are passed through, so it is possible that no rows

ever make it through the node. The FILTER operator has one input and one output. It accepts parameters describing the condition used for filtering.

APPEND: Appending New Calculated Columns

The APPEND operator appends new columns, which are calculated from existing columns and functions. The APPEND operator has one input and one output. It accepts parameters describing the new columns.

UNION: Combining Multiple Datasets into One

The UNION operator takes two or more datasets as inputs and creates a single output that combines all rows from both of them. The input datasets need to have exactly the same columns. The UNION operator has two or more inputs and one output.

AGGREGATE: Aggregating Values

The AGGREGATE operator groups its input based on zero or more key columns. All the rows with the same key values are summarized into a single row, and the output contains the aggregate key columns and the summaries. The AGGREGATE operator takes one input and produces one output. It also takes parameters describing the aggregate keys and the summaries to produce.

LOOKUP: Looking Up Values in One Table in Another

The LOOKUP operator takes two inputs, a base table and a reference table, which have a key in common. The reference table should have at most one row for each key value. The LOOKUP operator appends one or more columns in the reference table to the base table, based on matching key values. When there is no match, LOOKUP just outputs `NULL` for the corresponding output columns.

It takes two parameters. The first describes the key and the second describes which columns to append. Although this can also be accomplished with a JOIN, the LOOKUP is intended to be simpler and more readable for this common operation where no new rows are generated and no rows are filtered.

CROSSJOIN: Generating the Cartesian Product of Two Tables

The CROSSJOIN operator takes two inputs and combines them in a very specific way. It produces a wider table that contains all the columns in the two inputs, the Cartesian product of the two tables. Every row in the output corresponds to a pair of rows, one from each input. For instance, if the first table has four rows, A, B, C, and D, and the second has three rows, X, Y, and Z, then the output

consists of all twelve combinations of these: AX, AY, AZ, BX, BY, BZ, CX, CY, CZ, DX, DY, and DZ. The CROSSJOIN is the most general join operation.

JOIN: Combining Two Tables Using a Key Column

The JOIN operator takes two inputs and a join condition as a parameter, and produces an output that has all the columns in the two tables. The join condition typically specifies that at least one column in one table is related to one column in the other, usually by having the same value. This type of join, called an equijoin, is the most common type of join.

With an equijoin, it is possible to "lose" rows in one or both of the inputs. This occurs when there is no matching row in the other table. A variation of the join ensures that all rows in one or the other table are represented in the output. Specifically, the LEFT OUTER JOIN keeps all rows in the first input table and the RIGHT OUTER JOIN keeps all rows in the second. FULL OUTER JOIN keeps all rows in both tables.

SORT: Ordering the Results of a Dataset

The SORT operator orders its input dataset based on one or more sort keys. It takes a parameter describing the sort keys and the sort order (ascending or descending).

Dataflows, SQL, and Relational Algebra

Beneath the skin of most relational databases is an engine that is essentially a dataflow engine. Dataflows focus on data and SQL focuses on data, so they are natural allies.

Historically, though, SQL has a somewhat different theoretical foundation based on mathematical set theory. This foundation is called *relational algebra*, an area in mathematics that defines operations on unordered sets of *tuples*. A tuple is a lot like a row, consisting of attribute-value pairs. The "attribute" is the column and the "value" is the value of the column in the row. Relational algebra then includes a bunch of operations on sets of tuples, operations such as union and intersection, joins and projections, which are similar to the dataflow constructs just described.

The notion of using relational algebra to access data is credited to E. F. Codd who, while a researcher at IBM in 1970, wrote a paper called *A Relational Model of Data for Large Shared Data Banks*. This paper became the basis of using relational algebra for accessing data, eventually leading to the development of SQL and modern relational databases.

A set of tuples is a lot like a table, but not quite. One difference between the two is that a table can contain duplicate rows but a set of tuples cannot have duplicates. A very important property of sets is that they have no ordering. Sets have no

concept of the first, second, and third elements—unless another attribute defines the ordering. To most people (or at least most people who are not immersed in set theory), tables have a natural order, defined perhaps by a primary key or perhaps by the sequence that rows were originally loaded into the table.

As a legacy of the history of relational algebra, SQL tables have no natural ordering. The order of the results of a query are defined only when there is an `ORDER BY` clause.

SQL Queries

This section provides the third perspective on SQL, an introduction to the SQL querying language. The querying part of SQL is the visible portion of an iceberg whose bulky mass is hidden from view. The hidden portion is the data management side of the language—the definitions of tables and views, inserting rows, updating rows, defining triggers, stored procedures, and so on. As data miners and analysts, our goal is to exploit the visible part of the iceberg, by extracting useful information from the database.

SQL queries answer specific questions. Whether the question being asked is actually the question being answered is a big issue for database users. The examples throughout this book include both the question and the SQL that answers it. Sometimes, small changes in the question or the SQL produce very different results.

What to Do, Not How to Do It

A SQL query describes the result set, but does not specify how this is accomplished. This approach has several advantages. A query is isolated from the hardware and operating system where it is running. The same query should return equivalent results on the same data in two very different environments.

Being non-procedural means that SQL needs to be compiled into computer code on any given computer. This compilation step provides an opportunity to optimize the query to run as fast as possible in the environment. Database engines contain many different algorithms, ready to be used under just the right circumstances. The specific optimizations, though, might be quite different in different environments.

Another advantage of being non-procedural is that SQL can take advantage of parallel processing. The language itself was devised in a world where computers were very expensive, had a single processor, limited memory, and one disk. The fact that SQL has adapted to modern system architectures where CPUs, memory, and disks are plentiful is a testament to the power and scalability of the ideas underlying the relational database paradigm. When Codd wrote his paper suggesting relational algebra for "large data banks," he was probably thinking of a few megabytes of data, an amount of data that now easily fits in

an Excel spreadsheet and pales in comparison to gigabytes of information on a mobile device or the terabytes of data found in corporate repositories.

The SELECT Statement

This chapter has already included several examples of simple SQL queries. More formally, the `SELECT` statement consists of clauses, the most important of which are:

- `WITH`
- `SELECT`
- `FROM`
- `WHERE`
- `GROUP BY`
- `HAVING`
- `ORDER BY`

These clauses are always in this order. There is a close relationship between the dataflow operations discussed in the previous section and these clauses.

Note that a `SELECT` statement can contain subqueries within it. Supporting subqueries provide much of the power of SQL.

A Basic SQL Query

A good place to start with SQL is with the simplest type of query, one that selects a column from a table. Consider, once again, the query that returns zip codes along with the SCF:

```
SELECT zc.zcta5, LEFT(zc.zcta5, 3) as scf
FROM ZipCensus zc
```

This query returns a table with two columns, the zip code and the SCF. The rows might be returned in any order. If you want the rows in a particular order, include an explicit `ORDER BY` clause:

```
SELECT zc.zcta5, LEFT(zc.zcta5, 3) as scf
FROM ZipCensus zc
ORDER BY zc.zcta5
```

Without an `ORDER BY`, never assume that the result of a query will be in a particular order.

WARNING **The results of a query are unordered, unless you use an `ORDER BY` clause at the outermost level. Never depend on a "default ordering," because there isn't one.**

This simple query already shows some of the structure of the SQL language. All queries begin with the SELECT clause that lists the columns being returned. The tables being accessed are in the FROM clause, which follows the SELECT statement. And, the ORDER BY is the last clause in the query.

This example uses only one table, ZipCensus. In the query, this table has a *table alias*, or abbreviation, called zc. The first part of the SELECT statement is taking the zcta5 column from zc. Although table aliases are optional in SQL, as a rule this book uses them extensively because aliases clarify where columns come from and make queries easier to write and to read.

TIP **Use table aliases in your queries that are abbreviations for the table names. These make the queries easier to write and to read.**

The second column returned by the query is calculated from the zip code itself, using the LEFT() function. LEFT() is just one of dozens of functions provided by SQL, and specific databases generally support user-defined functions as well. The second column has a *column alias*. That is, the column is named SCF, which is the header of the column in the output.

A simple modification that returns the zip codes and SCFs only in Minnesota:

```
SELECT zc.zcta5, LEFT(zc.zcta5, 3) as scf
FROM ZipCensus zc
WHERE stab = 'MN'
ORDER BY 1
```

The query has an additional clause, the WHERE clause, which, if present, always follows the FROM clause. The WHERE clause specifies a condition; in this case, that only rows where stab is equal to "MN" are included in the result set. The ORDER BY clause then sorts the rows by the first column; the "1" is a reference to the first column being selected, in this case, zc.zcta5. The preferred method, however, is to use the column name (or alias) in the ORDER BY clause.

The dataflow corresponding to this modified query is in Figure 1-4. In this dataflow, the WHERE clause has turned into a filter after the data source, and the ORDER BY clause has turned into a SORT operator just before the output. Also notice that the dataflow contains several operators, even for a simple SQL query. SQL is a parsimonious language; complex operations can often be specified quite simply.

WARNING **When a column value is NULL, any comparison in a WHERE clause—with the important exception of IS NULL— always returns unknown, which is treated as FALSE. So, the clause WHERE stab <> 'MN' really means WHERE stab IS NOT NULL AND stab <> 'MN'.**

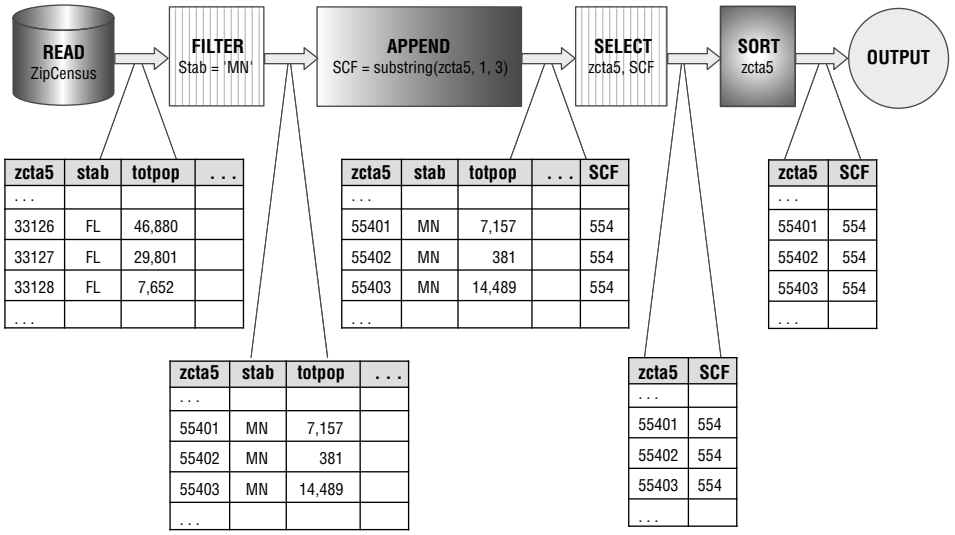


Figure 1-4: A **`WHERE`** clause in a query adds a filter node to the dataflow.

A Basic Summary SQL Query

A very powerful component of SQL is the ability to summarize data in a table. The following SQL counts the number of zip codes in `ZipCensus`:

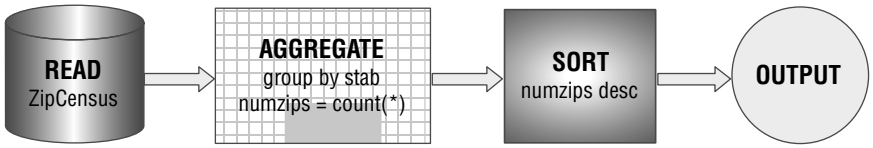
```
SELECT COUNT(*) as numzip
FROM ZipCensus zc
```

The form of this query is very similar to the basic select query. The function `COUNT(*)`, not surprisingly, counts the number of rows. The "`*`" means that all rows are being counted. It is also possible to count a column, such as `COUNT(zcta5)`. This counts the number of rows that have a valid (i.e., non-`NULL`) value in `zcta5`.

The preceding query is an aggregation query that treats the entire table as a single group. Within this group, the query counts the number of rows, which calculates the number of rows in the table. A very similar query returns the number of zip codes in each state:

```
SELECT stab, COUNT(*) as numzip
FROM ZipCensus zc
GROUP BY stab
ORDER BY numzip DESC
```

The `GROUP BY` clause says to treat the table as consisting of several groups defined by the different values in the column `stab`. The result is then sorted in reverse order of the count (`DESC` stands for "descending"), so the state with



zcta5	stab	totpop	. . .
. . .			
33126	FL	46,880	
33127	FL	29,801	
33128	FL	7,652	
. . .			

stab	numzips
PA	1,798
FL	983
AZ	405
MT	361
. . .	

stab	numzips
TX	1,935
PA	1,798
NY	1,794
CA	1,763
. . .	

Figure 1-5: This dataflow diagram describes a basic aggregation query.

the most zip codes (Texas) is first. Figure 1-5 shows the corresponding dataflow diagram.

In addition to `COUNT()`, standard SQL offers other useful aggregation functions. The `SUM()`, `AVG()`, `MIN()`, and `MAX()` functions compute, respectively, the sum, average, minimum, and maximum values. In general, the first two operate only on numeric values and the `MIN()` and `MAX()` can work on any data type. Note that all these functions ignore `NULL` values in their calculations.

`COUNT(DISTINCT)` returns the number of distinct values. An example of using it is to answer the following question: *How many SCFs are in each state?* The following query answers this question:

```
SELECT zc.stab, COUNT(DISTINCT LEFT(zc.zcta5, 3)) as numscf
FROM ZipCensus zc
GROUP BY zc.stab
ORDER BY zc.stab
```

This query also shows that functions, such as `LEFT()`, can be nested in the aggregation functions. SQL allows arbitrarily complicated expressions. Chapter 2 shows another way to answer this question using subqueries.

What It Means to Join Tables

Because they bring together information from two tables, joins are perhaps the most powerful feature of SQL. Database engines can have dozens of algorithms just for this one key word. A lot of programming and algorithms are hidden beneath this simple construct.

As with anything powerful, joins need to be used carefully—not sparingly, but carefully. It is very easy to make mistakes using joins, especially the following two:

- "Mistakenly" losing rows in the result set, and
- "Mistakenly" adding unexpected additional rows.

Whenever joining tables, it is worth asking whether either of these could be happening. These are subtle questions because the answer depends on the data being processed, not on the syntax of the expression itself. There are examples of both problems throughout the book.

This discussion is about what joins do rather than about the multitude of algorithms for implementing them (although the algorithms are quite interesting—to some people—they don't help us understand customers and data). The most general type of join is the cross-join. The discussion then explains the more common variants: look up joins, equijoins, nonequijoins, and outer joins.

WARNING **Whenever joining two tables, ask yourself the following two questions:**

1. **Could one of the tables accidentally be losing rows because there are no matches in the other table?**
2. **Could the result set unexpectedly have duplicate rows due to multiple matches between the tables?**

The answers require understanding the underlying data.

Cross-Joins: The Most General Joins

The most general form of joining two tables is called the *cross-join* or, for the more mathematically inclined, the *Cartesian product* of the two tables. As discussed earlier in the section on dataflows, a cross-join results in an output consisting of all columns from both tables and every combination of rows from one table with rows from the other. The number of rows in the output grows quickly as the two tables become bigger. If the first table has four rows and two columns, and the second has three rows and two columns, then the resulting output has twelve rows and four columns. This is easy enough to visualize in Figure 1-6.

Because the number of rows in the output is the number of rows in each table multiplied together, the output size grows quickly. If one table has 3,000 rows and the other 4,000 rows, the result has 12,000,000 rows—which is a bit big for an illustration. The number of potential columns is the sum of the number of columns in each input table.

In the business world, tables often have thousands, or millions, or even more rows, so a cross-join quickly gets out of hand, with even the fastest computers. If this is the case, why are joins so useful, important, and practical?

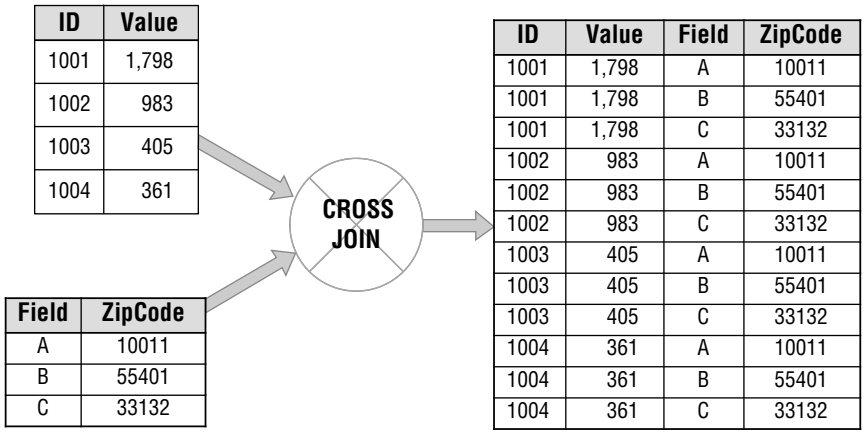


Figure 1-6: A cross-join on two tables, one with four rows and one with three rows, results in a new table that has twelve rows and all columns from both tables.

The reason is that the general form of the join is not the form that gets used very often, unless one of the tables is known to have only one row or a handful of rows. By imposing some restrictions—say by imposing a relationship between columns in the two tables—the result becomes more tractable. Even though more specialized joins are more commonly used, the cross-join is still the foundation that explains what they are doing.

Lookup: A Useful Join

`ZipCensus` is an example of a reference table summarized at the zip code level. Each row describes a zip code and any given zip code appears exactly once in the table. As a consequence, the `zcta5` column makes it possible to look up census information for zip codes stored in another table. Intuitively, this is one of the most natural join operations, using a *foreign key* in one table to look up values in a reference table.

A lookup join makes the following two assumptions about the base and reference tables:

- All values of the key in the base table are in the reference table (missing join keys lose rows unexpectedly).
- The lookup key is the primary key in the reference table (duplicate join keys cause unexpected rows).

Unfortunately, SQL does not provide direct support for lookups because there is no simple check in the query ensuring these two conditions are true. However, the join mechanism makes it possible to do lookups, and this works smoothly when the two preceding conditions are true.

Consider the SQL query that appends the zip code population to each row of `Orders`:

```
SELECT o.OrderId, o.ZipCode, zc.totpop
FROM Orders o JOIN
     ZipCensus zc
     ON o.ZipCode = zc.zcta5
```

This example uses the `ON` clause to establish the condition between the tables. There is no requirement that the condition be equality in general, but for a lookup it is.

From the dataflow perspective, the lookup could be implemented with CROSSJOIN. The output from the CROSSJOIN is first filtered to the correct rows (those where the two zip codes are equal) and the desired columns (all columns from `Orders` plus `totpop`) are selected. Figure 1-7 shows a dataflow that appends a population column to `Orders` using this approach.

Unlike the dataflow diagram, the SQL query describes that a join needs to take place, but does not explain how this is done. The cross-join is one method, although it would be quite inefficient in practice. Databases are practical, so database writers have invented many different ways to speed this up. The details of such performance enhancements are touched upon in Chapter 14, "Performance Is the Issue: Using SQL Effectively." It is worth remembering that databases are practical, not theoretical, and the database engine is usually trying to optimize the run-time performance of queries.

Although the preceding query does implement the lookup, it does not guarantee the two conditions mentioned earlier. If there were multiple rows in `ZipCensus` for a given zip code, there would be extra rows in the output (because any matching row would appear more than once). You can define a *constraint* or unique index on the table to ensure that it has no duplicates, but in the query itself there is no evidence of whether or not such a constraint is present. On the other hand, if zip code values in `Orders` were missing in `ZipCensus`, rows would unexpectedly disappear. In fact, this happens and the output has fewer rows than the original `Orders` table. The condition that all the zip codes in `Orders` match a row in `ZipCensus` could be enforced (if it were true) with another type of constraint, a *foreign key constraint*.

Having multiple rows in `ZipCensus` for a given zip code is not an outlandish idea. For instance, it could include information for both the 2000 and 2010 censuses, which would make it possible to see changes over time. One way to implement this would be to have another column, say, `CensusYear` to specify the year of the census. Now the primary key would be a compound key composed of `zcta5` and `CensusYear` together. A join on the table using just zip code would result in multiple rows, one for each census year.

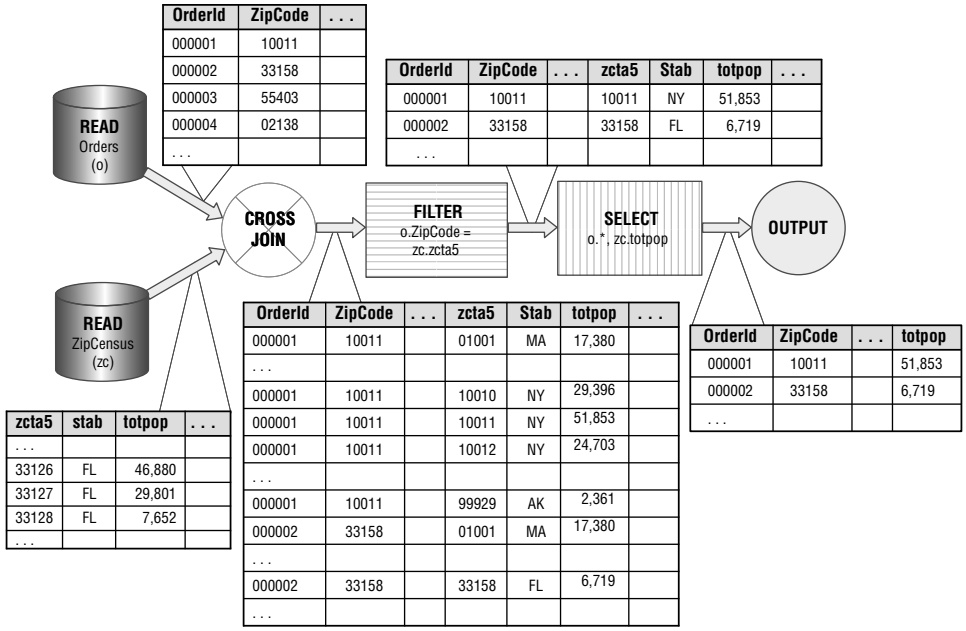


Figure 1-7: In SQL, looking up a value in one table is theoretically equivalent to creating the cross-join of the two tables and then restricting the values.

Equijoins

An equijoin is a join that has at least one condition asserting that two columns in the tables have equal values, and all the conditions are connected by `AND` (which is normally the case). In SQL, the conditions are the `ON` clause following the `JOIN` statement.

An equijoin can return extra rows the same way that a cross-join can. If a column value in the first table is repeated three times, and the same value occurs in the second table four times, the equijoin between the two tables produces twelve rows of output for that column. This is similar to the situation depicted in Figure 1-6 (page 27) that illustrates the cross-join. Using an equijoin, it is possible to add many rows to output that are not intended, especially when the equijoin is on non-key columns.

Equijoins can also filter out rows, when there are no matching key values in the second table. This filtering can be a useful feature. For instance, one table might have a small list of ids that are special in some way. The join would then apply this filter to the bigger table.

Although joins on primary keys are more common, there are cases where such a many-to-many equijoin is desired. Consider this question: *For each zip code, how many zip codes in the same state have a larger population?*

The following query uses a *self-join* (followed by an aggregation) to answer this question. A self-join simply means that two copies of the `ZipCensus` table are joined together. The equijoin uses the state column as a key, rather than the zip code column.

```
SELECT zc1.zcta5,
       SUM(CASE WHEN zc1.totpop < zc2.totpop THEN 1
                ELSE 0 END) as numzip
FROM ZipCensus zc1 JOIN
     ZipCensus zc2
     ON zc1.stab = zc2.stab
GROUP BY zc1.zcta5
```

Notice that `ZipCensus` is mentioned twice in the `FROM` clause. Each occurrence is given a different table alias to distinguish them in the query.

The dataflow for this query, in Figure 1-8, reads the `ZipCensus` table twice, feeding both into the JOIN operator. The JOIN in the dataflow is an equijoin because the condition is on the `stab` column. The results from the join are then aggregated.

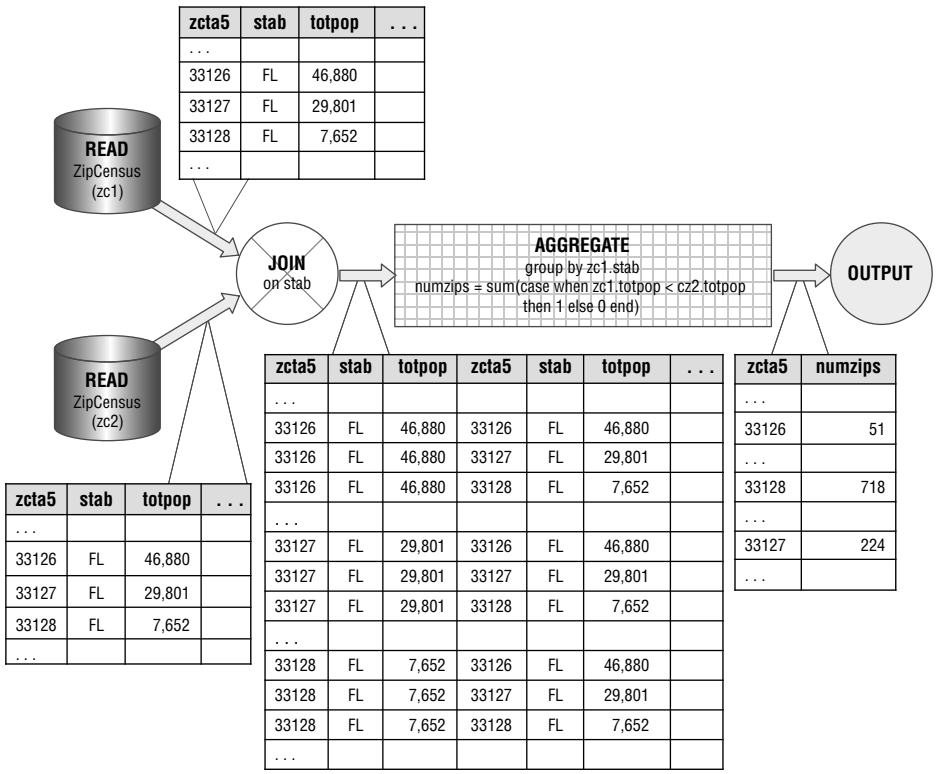


Figure 1-8: This dataflow illustrates a self-join and an equijoin on a non-key column.

Nonequijoins

A *nonequijoin* is a join where none of the conditions is equality between two columns. Nonequijoins are unusual. This is fortunate because there are many fewer performance tricks available to make them run quickly. Often, a nonequijoin is actually a mistake and indicates an error.

Note that when any of the conditions are equality, and the conditions are connected by AND, the join is an equijoin. Consider the following question about Orders: *How many orders are greater than the median rent where the customer resides*? The following query answers this question:

```
SELECT zc.stab, COUNT(*) as numrows
FROM Orders o JOIN
     ZipCensus zc
     ON o.zipcode = zc.zcta5 AND
        o.totalprice > zc.mediangrossrent
GROUP BY zc.stab
```

The JOIN in this query has two conditions, one specifies that the zip codes are equal and the other specifies that the total amount of the order is greater than the median rent in the zip code. This is still an example of an equijoin because of the condition on zip code.

Outer Joins

The final type of join is the outer join, which guarantees that all rows from one or both of the tables remain in the result set, even when there are no matching rows in the other table. All the previous joins have been *inner joins*, meaning that only rows that match are included. For a cross-join, this does not make a difference because there are many copies of rows from both tables in the result. However, for other types of joins, losing rows in one or the other table may not be desirable; hence the need for the outer join.

Lookups are a good example of an outer (equijoin), because the join asserts that a foreign key in one table equals a primary key in a reference table. Lookups return all the rows in the first table, even when there is no matching row.

Outer joins comes in three flavors:

- The LEFT OUTER JOIN ensures that all rows from the first table remain in the result set.
- The RIGHT OUTER JOIN ensures that all rows from the second table remain.
- The FULL OUTER JOIN ensures that all rows from both tables are kept. When there is no match, then the columns from the "missing" table are all set to NULL in the result set.

What does this mean? Consider the Orders table, which has some zip codes that are not in ZipCensus. This could occur for several reasons. ZipCensus contains a snapshot of zip codes as of the census, and new zip codes might have appeared since then. Also, the Census Bureau is not interested in all zip codes, so they exclude some zip codes where no one lives. Or, perhaps the problem might lie in Orders. There could be mistakes in the ZipCode column. Or, as is the case, the Orders table might include orders from outside the United States.

Whatever the reason, any query using the inner join eliminates all rows where the zip code in Orders does not appear in ZipCensus. Losing such rows could be a problem, which the outer join fixes. The only change to the query is replacing the word JOIN with the phrase LEFT OUTER JOIN (or equivalently LEFT JOIN):

```
SELECT zc.stab, COUNT(*) as numrows
FROM Orders o LEFT OUTER JOIN
     ZipCensus zc
     ON o.ZipCode = zc.zcta5 AND
        o.TotalPrice > zc.mediangrossrent
GROUP BY zc.stab
```

The results from this query are not particularly interesting. The results are the same as the previous query with one additional large group for NULL. When there is no matching row in ZipCensus, zc.stab is NULL.

TIP **In general, you can write queries using just LEFT OUTER JOIN and INNER JOIN. There is usually no reason to mix LEFT OUTER JOIN and RIGHT OUTER JOIN in the same query.**

Left outer joins are very practical. When they are chained together, they essentially say "keep all rows in the first table." As a general rule, don't mix outer join types if you can avoid it, because just having LEFT OUTER JOINs and INNER JOINs is sufficient for most purposes. As an example, if one table contains information about customers, then subsequent joins could bring in other columns from other tables, and the LEFT OUTER JOIN ensures that no customers are accidently lost. Chapter 13, "Building Custom Signatures for Further Analysis," uses outer joins extensively.

Other Important Capabilities in SQL

SQL has other features that are used throughout the book. The goal here is not to explain every nuance of the language, because reference manuals and database documentation do a good job there. The goal here is to give a feel for the important capabilities of SQL needed for data analysis.

UNION ALL

UNION ALL is a set operation that combines all rows in two tables, by just creating a result set with all the rows from each input table. The columns must be the same in each of the input tables. In practice, this means that UNION ALL almost always operates on subqueries, because it is unusual for two tables to have exactly the same columns.

SQL has other set operations, such as UNION, INTERSECTION, and MINUS (also called EXCEPT). The UNION operation combines the rows in two tables together, and then removes duplicates. This means that UNION is much less efficient than UNION ALL, so it is worth avoiding. INTERSECTION takes the overlap of two tables—rows that are in both. However, it is often more interesting to understand the relationship between two tables—how many items are in both and how many are in each one but not the other. Solving this problem is discussed in Chapter 2.

CASE

The CASE expression adds conditional logic into the SQL language. Its most general form is:

```
CASE WHEN <condition-1> THEN <value-1>
     . . .
     WHEN <condition-n THEN <value-n>
     ELSE <default-value> END
```

The <condition> clauses look like conditions in a WHERE clause; they can be arbitrarily complicated. The <value> clauses are values returned by the statement, and these should all be the same type. The <condition> clauses are evaluated in the order they are written. When no <else> condition is present, the CASE statement returns NULL if none of the previous clauses match.

One common use of CASE is to create indicator variables. Consider the following question: *How many zip codes in each state have a population of more than 10,000 and what is the total population of these?* The following SQL query is, perhaps, the most natural way of answering this question:

```
SELECT zc.stab, COUNT(*) as numbigzip, SUM(totpop) as popbigzip
FROM ZipCensus zc
WHERE totpop > 10000
GROUP BY zc.stab
```

This query uses a WHERE clause to choose the appropriate set of zip codes.

Now consider the related question: *How many zip codes in each state have a population of more than 10,000, how many have more than 1,000, and what is the total population of each of these sets?*

Unfortunately, the WHERE clause solution no longer works, because two overlapping sets of zip codes are needed. One solution is to run two queries, which is messy. Combining the results into a single query is easy using conditional aggregation:

```
SELECT zc.stab,
       SUM(CASE WHEN totpop > 10000 THEN 1 ELSE 0 END) as num_10000,
       SUM(CASE WHEN totpop > 1000  THEN 1 ELSE 0 END) as num_1000,
       SUM(CASE WHEN totpop > 10000 THEN totpop ELSE 0 END
          ) as pop_10000,
       SUM(CASE WHEN totpop > 1000  THEN totpop ELSE 0 END
          ) as pop_1000
FROM ZipCensus zc
GROUP BY zc.stab
```

Notice that in this version, the SUM() function is used to count zip codes that meet the appropriate condition; it does so by adding 1 for each matching row. COUNT() is not the right function, because it would count the number of non-NULL values.

TIP When a CASE statement is nested in an aggregation function, the appropriate function is usually SUM(), or MAX() sometimes AVG(), and on rare occasions COUNT(DISTINCT). Check to be sure that you are using SUM() even when "counting" things up.

The following two statements are very close to being the same, but the second lacks the ELSE clause:

```
SUM(CASE WHEN totpop > 10000 THEN 1 ELSE 0 END) as num_10000,
SUM(CASE WHEN totpop > 10000 THEN 1 END) as num_10000,
```

Each counts the number of zip codes where population is greater than 10,000. The difference is what happens when no zip codes have such a large population. The first returns the number 0. The second returns NULL. Usually when counting things, it is preferable to have the value be a number rather than NULL, so the first form is generally preferred.

The CASE statement can be much more readable than the WHERE clause because the CASE statement has the condition in the SELECT, rather than much further down in the query. On the other hand, the WHERE clause provides more opportunities for optimization.

IN

The IN statement is used in a WHERE clause to choose items from a set. The following WHERE clause chooses zip codes in New England states:

```
WHERE stab IN ('VT', 'NH', 'ME', 'MA', 'CT', 'RI')
```

This use is equivalent to the following:

```
WHERE (stab = 'VT' OR
       stab = 'NH' OR
       stab = 'ME' OR
       stab = 'MA' OR
       stab = 'CT' OR
       stab = 'RI')
```

The `IN` statement is easier to read and easier to modify.

Similarly, the following `NOT IN` statement would choose zip codes that are not in New England:

```
WHERE stab NOT IN ('VT', 'NH', 'ME', 'MA', 'CT', 'RI')
```

This use of the `IN` statement is simply a convenient shorthand for what would otherwise be complicated `WHERE` clauses. The section on subqueries explores another use of `IN`.

Window Functions

Window functions are a class of functions that use the `OVER` clause. These functions return a value on a single row, but the value is based on a group of rows. A simple example is `SUM()`. Say we wanted to return each zip code with the sum of the population in the state. With window functions, this is easy:

```
SELECT zc.zcta5,
       SUM(totpop) OVER (PARTITION BY zc.stab) as stpop
FROM ZipCensus zc;
```

The `PARTITION BY` clause says "do the sum for all rows with the same value of `stab`." The result is that all zip codes in a given state have the same value for `stpop`.

A particularly interesting window function is `ROW_NUMBER()`. This assigns a sequential value, starting with 1, to rows within each group.

```
SELECT zc.zcta5,
       SUM(totpop) OVER (PARTITION BY zc.stab) as stpop,
       ROW_NUMBER() OVER (PARTITION BY zc.stab
                          ORDER BY totpop DESC
                         ) as ZipPopRank
FROM ZipCensus zc
```

This query adds an additional ranking column to each row in the result set. The value is 1 for the zip code with the highest population in each state, 2 for the second highest, and so on.

Table 1-2: Example of ROW_NUMBER(), RANK(), and DENSE_RANK()

VALUE	ROW_NUMBER()	RANK()	DENSE_RANK()
10	1	1	1
20	2	2	2
20	3	2	2
30	4	4	3
50	5	5	4
50	6	5	4

SQL offers two other similar functions for ranking: `RANK()` and `DENSE_RANK()`. They differ in their handling of ties, as shown by the example in Table 1-2.

All three functions assign the first row a number of "1". `ROW_NUMBER()` ignores duplicates, just giving each row a different number. `RANK()` assigns duplicate numbers when rows have the same value, but then skips the next numbers, so the results have gaps. `DENSE_RANK()` is like rank except the resulting numbers have no gaps.

Subqueries and Common Table Expressions Are Our Friends

Subqueries are exactly what their name implies, queries within queries. They make it possible to do complex data manipulation within a single SQL statement, exactly the types of manipulation needed for data analysis and data mining.

In one sense, subqueries are not needed. All the manipulations could be accomplished by creating intermediate tables and combining them. The resulting SQL would be a series of `CREATE TABLE` statements and `INSERT` statements (or possibly `CREATE VIEW` or `SELECT INTO`), with simpler queries. Although such an approach is sometimes useful, especially when the intermediate tables are used multiple times, it suffers from several problems.

First, instead of thinking about solving a particular problem, you end up thinking about the data processing, the naming of intermediate tables, determining the types of columns, remembering to remove tables when they are no longer needed, deciding whether to build indexes, and so on. All the additional bookkeeping activity distracts from focusing on the data and the business problems.

Second, SQL optimizers can often find better approaches to running a complicated query than people can. So, writing multiple SQL statements can interfere with the optimizer.

Third, maintaining a complicated chain of queries connected by tables can be quite cumbersome. For instance, adding a new column might require adding new columns in all sorts of places. Or, you may run part of the script and not realize that one of the intermediate tables has values from a previous run.

Fourth, the read-only SQL queries that predominate in this book can be run with a minimum of permissions for the user—simply the permissions to run queries. Running complicated scripts requires create and modify permissions on at least part of the database. These permissions are dangerous, because an analyst might inadvertently damage the database. Without these permissions, it is impossible to cause such damage.

Subqueries can appear in many different parts of the query, in the `SELECT` clause, in the `FROM` clause, and in the `WHERE` and `HAVING` clauses. However, this section approaches subqueries by why they are used rather than where they appear syntactically.

Common table expressions (often referred to as *CTE*s) are another way of writing queries that appear in the `FROM` clause. They are more powerful than subqueries for two reasons. First, they can be used multiple times throughout the query. And, they can refer to themselves—something called *recursive CTE*s. The following sections have examples of both CTEs and subqueries.

Subqueries for Naming Variables

When it comes to naming variables, SQL has a shortcoming. The following is not syntactically correct in most SQL dialects:

```
SELECT totpop as pop, pop + 1
```

The `SELECT` statement names columns, but these names cannot be used again in the same clause. Because queries should be at least somewhat understandable to humans, as well as database engines, this is a real shortcoming. Complicated expressions should have names.

Fortunately, subqueries provide a solution. The earlier query that summarized zip codes by population greater than 10,000 and greater than 1,000 could instead use a subquery that is clearer about what is happening:

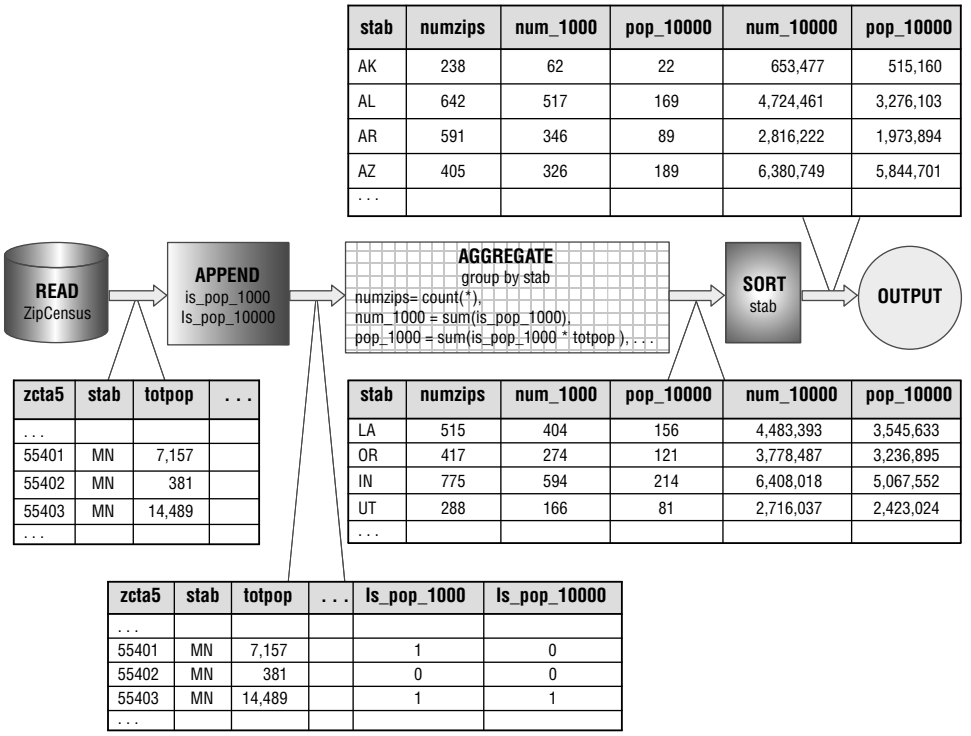
```
SELECT zc.stab,
       SUM(is_pop_10000) as num_10000,
       SUM(is_pop_1000) as num_1000,
       SUM(is_pop_10000 * totpop) as pop_10000,
       SUM(is_pop_1000 * totpop) as pop_1000
FROM (SELECT zc.*,
             (CASE WHEN totpop > 10000 THEN 1 ELSE 0
              END) as is_pop_10000,
             (CASE WHEN totpop > 1000 THEN 1 ELSE 0
              END) as is_pop_1000
```

```
        FROM ZipCensus zc
      ) zc
GROUP BY zc.stab
```

This version of the query uses two indicator variables, `IS_POP_10000` and `IS_POP_1000`. These take on the value of 0 or 1, depending on whether or not the population is greater than 10,000 or 1,000. The query then sums the indicators to get the counts, and sums the product of the indicator and the population to get the population count. Figure 1-9 illustrates this process as a dataflow. Notice that the dataflow does not include a "subquery."

TIP **Subqueries with indicator variables, such as `IS_POP_1000`, are a powerful and flexible way to build queries.**

Indicator variables are only one example of using subqueries to name variables. Throughout the book, there are many other examples. The purpose is to make the queries understandable to humans, relatively easy to modify, and might, with luck, help us remember what a query written six months ago is really doing.



stab	numzips	num_1000	pop_10000	num_10000	pop_10000
AK	238	62	22	653,477	515,160
AL	642	517	169	4,724,461	3,276,103
AR	591	346	89	2,816,222	1,973,894
AZ	405	326	189	6,380,749	5,844,701
. . .					

zcta5	stab	totpop	. . .
. . .			
55401	MN	7,157	
55402	MN	381	
55403	MN	14,489	
. . .			

stab	numzips	num_1000	pop_10000	num_10000	pop_10000
LA	515	404	156	4,483,393	3,545,633
OR	417	274	121	3,778,487	3,236,895
IN	775	594	214	6,408,018	5,067,552
UT	288	166	81	2,716,037	2,423,024
. . .					

zcta5	stab	totpop	. . .	ls_pop_1000	ls_pop_10000
. . .					
55401	MN	7,157		1	0
55402	MN	381		0	0
55403	MN	14,489		1	1
. . .					

Figure 1-9: This dataflow illustrates the process of using indicator variables to obtain information about zip codes.

FORMATTING SQL QUERIES

There is no agreed-upon standard for formatting SQL queries. There are a few good practices, such as:

- Use table aliases that are abbreviations for the table name.
- Use `as` to define column aliases.
- Be consistent in capitalization, in usage of underscores, and in indentation.
- Write the code to be understandable, so you and someone else can read it.

Writing readable code is always a good idea.

Any guidelines for writing code necessarily have a subjective element. The goal should be to communicate what the query is doing. Formatting is important: Just imagine how difficult it would be to read text without punctuation, capitalization, and paragraphs.

The code in this book (and on the companion website) follows additional rules to make the queries easier to follow.

- Most keywords are capitalized and most table and column names use CamelBack casing (except for `ZipCensus`).
- The high-level clauses defined by the SQL language are all aligned on the left. These are `WITH`, `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`.
- Within a clause, subsequent lines are aligned after the keyword, so the scope of each clause is visually obvious.
- Subqueries follow similar rules, so all the main clauses of a subquery are indented, but still aligned on the left.
- Within the `FROM` clause, table names and subqueries start on a new line (the tables are then aligned and easier to see). The `ON` predicate starts on its own line, and the join keywords are at the end of the line.
- Columns are generally qualified, meaning that they use table aliases.
- Operators generally have spaces around them.
- Commas are at the end of a line, just as a human would place them.
- Closing parenthesis—when on a subsequent line—is aligned under the opening parenthesis.
- `CASE` statements are always surrounded by parentheses.

The goal should be to write queries so other people can readily understand them. After all, you may be returning to your queries one day and you would like to be able to quickly figure out what they are doing.

The above subquery can also be written as a CTE:

```
WITH zc as (
      SELECT zc.*,
             (CASE WHEN totpop > 10000 THEN 1 ELSE 0
              END) as is_pop_10000,
             (CASE WHEN totpop > 1000 THEN 1 ELSE 0
              END) as is_pop_1000
      FROM ZipCensus zc
     )
SELECT zc.stab,
       SUM(is_pop_10000) as num_10000,
       SUM(is_pop_1000) as num_1000,
       SUM(is_pop_10000 * totpop) as pop_10000,
       SUM(is_pop_1000 * totpop) as pop_1000
FROM zc
GROUP BY zc.stab
```

Here, the subquery is introduced using the `WITH` clause; otherwise, it is very similar to the version with a subquery in the `FROM` clause. A query can have only one `WITH` clause, although it can define multiple CTEs. These can refer to CTEs defined earlier in the same clause.

Subqueries for Handling Summaries

The most typical place for a subquery is as a replacement for a table in the `FROM` clause. After all, the source is a table and a query essentially returns a table, so it makes a lot of sense to combine queries in this way. From the dataflow perspective, this use of subqueries is simply to replace one of the sources with a series of dataflow nodes.

Consider the question: *How many zip codes in each state have a population density greater than the average zip code population density in the state?* The population density is the population divided by the land area, which is in the column `landsqmi`.

Let's think about the different data elements needed to answer the question. The comparison is to the average zip code population density within a state, which is easily calculated:

```
SELECT zc.stab, AVG(totpop / landsqmi) as avgpopdensity
FROM ZipCensus zc
WHERE zc.landsqmi > 0
GROUP BY zc.stab
```

Next, the idea is to combine this information with the original zip code information in the `FROM` clause:

```
SELECT zc.stab, COUNT(*) as numzips,
       SUM(CASE WHEN zc.popdensity > zcsum.avgpopdensity
                THEN 1 ELSE 0 END) as numdenser
```

```
FROM (SELECT zc.*, totpop / landsqmi as popdensity
      FROM ZipCensus zc
      WHERE zc.landsqmi > 0
     ) zc JOIN
     (SELECT zc.stab, AVG(totpop / landsqmi) as avgpopdensity
      FROM ZipCensus zc
      WHERE zc.landsqmi > 0
      GROUP BY zc.stab) zcsum
    ON zc.stab = zcsum.stab
GROUP BY zc.stab
```

The dataflow diagram for this query follows the same logic and is shown in Figure 1-10. Later in this chapter we will see another way to answer this question using window functions.

An interesting observation is that the population density of each state is not the same as the average of the population densities for all the zip codes in the state. That is, the preceding question is different from: *How many zip codes in each state have a population density greater than the state's population density?* The state's population density would be calculated in `zcsum` as:

```
SUM(totpop) / SUM(landsqmi) as statepopdensity
```

There is a relationship between these two densities. The zip code average gives each zip code a weight of 1, no matter how big in area or population. The state average is the weighted average of the zip codes by the land area of the zip codes.

The proportion of zip codes that are denser than the average zip code varies from about 4% of the zip codes in North Dakota to about 35% in Florida. Never are half the zip codes denser than the average, although this is theoretically possible. The density where half the zip codes are denser and half less dense is the *median* density rather than the average or average of averages. Averages, average of averages, and medians are different from each other and discussed in Chapter 2.

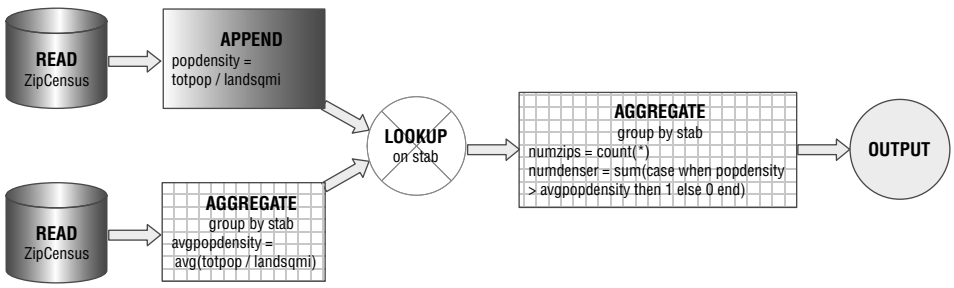


Figure 1-10: This dataflow diagram compares the zip code population density to the average zip code population density in a state.

Subqueries and IN

IN and NOT IN were introduced earlier as convenient shorthand for complicated WHERE clauses. There is another version where the "in" set is specified by a subquery, rather than by a fixed list. For example, the following query gets the list of all zip codes in states with fewer than 100 zip codes:

```
SELECT zc.zcta5, zc.stab
FROM ZipCensus zc
WHERE zc.stab IN (SELECT stab
                  FROM ZipCensus
                  GROUP BY stab
                  HAVING COUNT(*) < 100
                 )
```

The subquery creates a set of all states in ZipCensus where the number of zip codes in the state is less than 100 (that is, DC, DE, HI, and RI). The HAVING clause sets this limit. HAVING is very similar to WHERE, except it filters rows *after* aggregating, rather than *before*. The outer SELECT then chooses zip codes whose state matches one of the states in the IN set. This process takes place as a join operation, as shown in Figure 1-11.

Rewriting the "IN" as a JOIN

Strictly speaking, the IN operator is not necessary, because queries with INs and subqueries can be rewritten as joins. For example, this is equivalent to the previous query:

```
SELECT zc.*
FROM ZipCensus zc JOIN
     (SELECT stab, COUNT(*) as numstates
      FROM ZipCensus
      GROUP BY stab
     ) zipstates
     ON zc.stab = zipstates.stab AND
        zipstates.numstates < 100
```

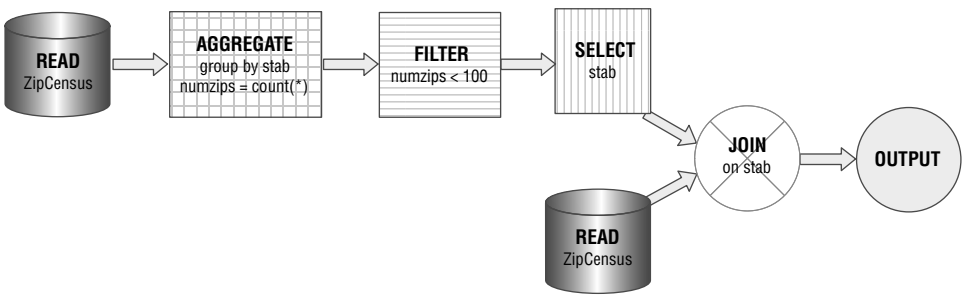


Figure 1-11: The processing for an **IN** with a subquery really uses a join operation.

Note that in the rewritten query, the `zipstates` subquery has two columns instead of one. The second column contains the count of zip codes in each state. Using the `IN` statement with a subquery makes it impossible to get this information.

On the other hand, the `IN` does have a small advantage, because it guarantees that there are no duplicate rows in the output, even when the "in" set has duplicates. To guarantee this using the `JOIN`, aggregate the subquery by the key used to join the tables. In this case, the subquery is doing aggregation anyway to find the states that have fewer than one hundred zip codes. This aggregation has the additional effect of guaranteeing that the subquery has no duplicates.

The general way of rewriting an `IN` subquery using join requires eliminating the duplicates. So, the query:

```
SELECT x.*
FROM x
WHERE x.col_a IN (SELECT y.col_b FROM y)
```

would be rewritten as:

```
SELECT DISTINCT x.*
FROM x JOIN
     y
     ON x.col_a = y.col_b;
```

or:

```
SELECT x.*
FROM x JOIN
     (SELECT DISTINCT y.col_b FROM y) y
     ON x.col_a = y.col_b;
```

The `DISTINCT` keyword removes duplicates from the output. However, this requires additional processing so it is best to avoid `DISTINCT` unless it is really necessary.

Correlated Subqueries

A correlated subquery occurs when the subquery includes a reference to the outer query. An example shows this best. Consider the following question: *Which zip code in each state has the maximum population and what is the population?* One way to approach this problem uses a correlated subquery:

```
SELECT zc.stab, zc.zcta5, zc.totpop
FROM ZipCensus zc
WHERE zc.totpop = (SELECT MAX(zcinner.totpop)
                   FROM ZipCensus zcinner
                   WHERE zcinner.stab = zc.stab
                  )
ORDER BY zc.stab
```

The "correlated" part of the subquery is the inner WHERE clause, which specifies that the state in a record processed by the subquery must match the state in the outer table.

Conceptually, the database engine reads one row from zc (the table referenced in the outer query). Then, the engine finds all rows in zcinner that match this state. From these rows, it calculates the maximum population. If the original row matches this maximum, it is selected. The engine then moves on to the next row in the outer query.

Correlated subqueries are sometimes cumbersome to understand. Although complicated, correlated subqueries are not a new way of processing the data; they are another example of joins. The following query produces the same results:

```
SELECT zc.stab, zc.zcta5, zc.totpop
FROM ZipCensus zc JOIN
     (SELECT zc.stab, MAX(zc.totpop) as maxpop
      FROM ZipCensus zc
      GROUP BY zc.stab) zcsum
     ON zc.stab = zcsum.stab AND
        zc.totpop = zcsum.maxpop
ORDER BY zc.stab
```

This query makes it clear that ZipCensus is summarized by stab to calculate the maximum population. The JOIN then finds the zip code (or possibly zip codes) that matches the maximum population, returning information about them. In addition, this method makes it possible to include other information, such as the number of zip codes where the maximum population is achieved. This can be calculated using COUNT(*) in zcsum.

The examples throughout this book tend not to use correlated subqueries for SELECT queries, preferring explicit JOINs instead. Joins provide more flexibility for processing and analyzing data and, in general, SQL engines do a good job of optimizing JOINs. There are some situations where the correlated subquery may offer better performance than the corresponding JOIN query or may even be simpler to understand.

NOT IN Operator

The NOT IN operator can also use subqueries and correlated subqueries. Consider the following question: *Which zip codes in the Orders table are not in the ZipCensus table*? Once again, there are different ways to answer this question. The first uses the NOT IN operator:

```
SELECT o.ZipCode, COUNT(*) as NumOrders
FROM Orders o
WHERE ZipCode NOT IN (SELECT zcta5
                      FROM ZipCensus zc
                     )
GROUP BY o.ZipCode
```

This query is straightforward as written, choosing the zip codes in `Orders` with no matching zip code in `ZipCensus`, then grouping them and returning the number of purchases in each.

An alternative uses the `LEFT OUTER JOIN` operator. Because the `LEFT OUTER JOIN` keeps all zip codes in the `Orders` table—even those that don't match—a filter afterwards can choose the non-matching set:

```
SELECT o.ZipCode, COUNT(*) as NumOrders
FROM Orders o LEFT OUTER JOIN
     ZipCensus zc
     ON o.ZipCode = zc.zcta5
WHERE zc.zcta5 IS NULL
GROUP BY o.ZipCode
ORDER BY NumOrders DESC
```

This query joins the two tables using a `LEFT OUTER JOIN` and only keeps the results rows do not match (because of the `WHERE` clause). This is essentially equivalent to using `NOT IN`; whether one works better than the other depends on the underlying optimization engine. Figure 1-12 shows the dataflow associated with this query.

EXISTS and NOT EXISTS Operators

`EXISTS` and `NOT EXISTS` are similar to `IN` and `NOT IN` with subqueries. The operators return true when any row exists (or no row exists) in a subquery. They are often used with correlated subqueries.

The query to return all the orders whose zip code is not in `ZipCensus` could be written as:

```
SELECT o.ZipCode, COUNT(*)
FROM Orders o
WHERE NOT EXISTS (SELECT 1
                  FROM ZipCensus zc
                  WHERE zc.zcta5 = o.ZipCode)
GROUP BY o.ZipCode
```

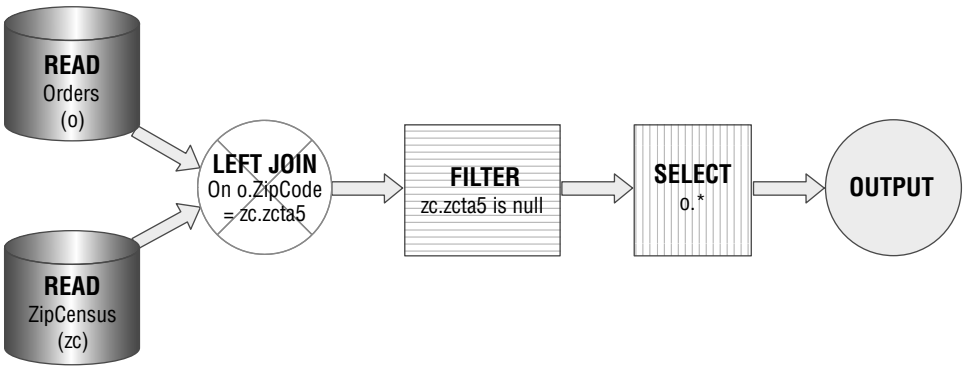


Figure 1-12: This dataflow shows the **`LEFT OUTER JOIN`** version of a query using **`NOT IN`**.

The "1" in the subquery has no importance, because NOT EXISTS is really determining if any *rows* are returned. It doesn't care about the particular value in any of the *columns*. In fact, some databases accept a nonsensical value, such as 1 / 0 (although this is not recommended).

EXISTS has several advantages over IN. First, EXISTS is more expressive—the comparison could be made on more than one column. IN only works for comparing one column to a list (although some databases extend this functionality to multiple columns). If, for instance, the query were comparing both state name and country name, then it would be easier to write using NOT EXISTS.

A second advantage is more subtle and applies only to NOT EXISTS. If the list of values returned by NOT IN contains a NULL value, then all rows fail the test. Why? SQL treats a comparison to NULL as unknown. So if the comparison were 'X' NOT IN ('A', 'B', 'X', NULL) then the result is false, because 'X' is, in fact, in the list. If the comparison were 'X' NOT IN ('A', 'B', NULL), then the result is unknown, because it is unknown whether or not X matches the NULL. The important point: neither version returns true. The equivalent NOT EXISTS query behaves more intuitively. The second example—using NOT EXISTS—would return true.

The final advantage is practical. In many databases, EXISTS and NOT EXISTS are optimized to be more efficient than the equivalent IN and NOT IN. One reason is that IN essentially creates the entire underlying list and then does the comparison, whereas EXISTS can simply stop at the first matching value.

Subqueries for UNION ALL

The UNION ALL operator almost always demands subqueries, because it requires that the columns be the same for all tables involved in the union. Consider extracting the location names from ZipCensus into a single column along with the type:

```
SELECT u.location, u.locationtype
FROM ((SELECT DISTINCT stab as location, 'state' as locationtype
       FROM ZipCensus zc
      ) UNION ALL
      (SELECT DISTINCT county, 'county' FROM ZipCensus zc
      ) UNION ALL
      (SELECT DISTINCT zipname, 'zipname' FROM ZipCensus zc
      )
     ) u
```

This example uses subqueries to ensure that each part of the UNION ALL has the same columns. Also, note that the column names are taken from the first subquery, so they are not needed in the subsequent subqueries.

Lessons Learned

This chapter introduces SQL and relational databases from several different perspectives that are important for data mining and data analysis. The focus is exclusively on using databases to extract information from data, rather than on the mechanics of building databases, the myriad options available in designing them, or the sophisticated algorithms implemented by database engines.

One very important perspective is the data perspective—the tables themselves and the relationships between them. Entity-relationship diagrams are a good way of visualizing the structure of data in the database and the relationships among tables. Along with introducing entity-relationship diagrams, the chapter also explains the various datasets used throughout this book.

Of course, tables and databases store data, but they don't actually do anything. Queries extract information, transforming data into information. For some people, thinking in terms of data flow diagrams is simpler than understanding complex SQL statements. These diagrams show how various operators transform data. About one dozen operators suffice for the rich set of processing available in SQL. Dataflows are not only useful for explaining how SQL processes data; database engines generally use a form of dataflows for running SQL queries.

In the end, though, transforming data into information requires SQL queries, whether simple or complex. The focus in this chapter, and throughout the book, is on SQL for querying. This chapter introduces the important functionality of SQL and how it is expressed, with particular emphasis on `JOIN`s, `GROUP BY`s, and subqueries, because these play an important role in data analysis.

The next chapter starts the path toward using SQL for data analysis by exploring data in a single table.