CHAPTER 3

SQL

3.1 Getting started with SQL

In the previous chapter, we discussed relational theory as the motivation and defining theoretical framework behind SQL. Now, we will consider the SQL language itself and the database products that implement it. In doing this, we will walk through several example queries and discuss their respective outputs as a method of understanding the mechanics of the language. This chapter assumes you are new to SQL and so we really focus on unpacking a series of these examples. In the next chapter, we will progress to an end-to-end project, which you can read and even follow along running commands yourself. Once you have a working understanding of SQL, it is beneficial to set up your own database environment and connect to it using one of many user-friendly graphical database exploration tools. A popular example of such tools is known as DBViz and it will allow you to construct and run SQL code as well as visually explore tables in a relational database. We leave this as a more advanced exercise for you to pursue.

3.2 Structure of SQL databases

Before going into the syntax and format of SQL queries, it is beneficial to quickly review the general architecture of a relational database as applications. In the last chapter, we discussed the important difference between the relational model as a logical system and SQL as a language designed to query and manipulate data stored according to a relational model. We also discussed the fact that SQL is a declarative language whose statements describe the rules by which the database should fetch and return data but that the details regarding how to optimize and implement SQL queries themselves are largely governed by the "query planner," which is an internal component within database applications. In fact database applications contain many additional functionalities beyond just storing tabular information, including the creation of users and rules governing which tables user can access and what actions they can perform, the creation of indexes to make tables easier to search (which we will discuss later in this chapter), and the creation of schemas which determine how tables are organizes (which we will also discuss later in this chapter). For these reasons, relational database applications are commonly called "Relational Database Management Systems" (RDBMS) as they support several data manipulation and management functions. Fig. 3.1 depicts the general architecture of such an RDBMS system.

An Introduction to Healthcare Informatics. https://doi.org/10.1016/B978-0-12-814915-7.00003-X

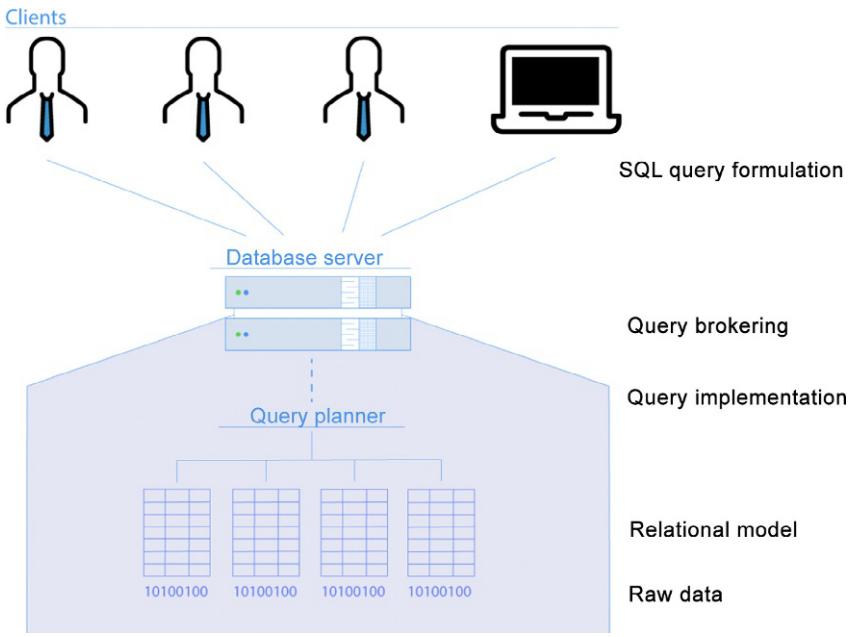

Fig. 3.1
RDBMS architecture consisting of a database server that receives SQL queries from client users and applications and implements those queries on its constituent tables. These tables themselves are abstractions on underlying raw data.

An additional point to discuss is the distinction in vocabulary between the relational model and SQL as a language which, although similar, are not quite the same. In a general sense, we can assume equivalences between relations and tables, attributes and columns, and tuples and rows and it is common practice to do so. However, recall from the previous chapter that, strictly speaking, this separation in vocabulary is appropriate because, although a row and a tuple are very similar, a row has no strict limitation that all values must be unique. This is a subtle but important example of the difference between the relational model as a theoretical construct and an RDBMS as an actual product that relaxes the clean and strict rules of the relational model to provide flexibility and convenience to users. Going forward, we will keep the discussion appropriately focused on SQL and, therefore, we will be talking in terms of tables, columns, and rows. We will also frequently describe "standard" SQL as those commands, keywords, and operators, which are part of the core definition of the SQL language as maintained by the American National Standards Institute, known as "ANSI SQL." Specific database products often extend ANSI SQL to over additional powerful but vendor-specific functionality, giving rise to several different "dialects" of SQL specific to different RDBMS systems.

3.3 Basic SQL: SELECT, FROM, WHERE, and ORDER BY statements

As a simple starting point, we will begin with the SELECT statement, used to retrieve data from a target table. We will start with an example statement first and then dissect the statement and its output. Consider the following SQL query:

```
SELECT TOP 2 * FROM Patients WHERE FirstName = 'Peter',
```

which returns the output:

```
_____________________________________________
| PatientID | FirstName | LastName   | Age | Sex  |
_____________________________________________
| 1         | Peter     | McCaffrey  | 30  | Male |
| 2         | Peter     | Parker     | 22  | Male |
_____________________________________________
```

What can we assume about this query just by reading as though it was a plain English commands? The statement we just ran roughly means "Select all of the columns of the top two rows in the table called Patients for which the value of the FirstName column is Peter." The relation is the Patients table, the predicate variable is the FirstName attribute, and the predicate is the test for whether the FirstName attribute of a row in the Patients relation is equal to "Peter" and the predicate is true for the two rows returned to us. Thus, the SQL query declares certain conditions for which only rows which are TRUE under those conditions are returned.

This is the most basic and essential example of an SQL query and it uses the three most common key words in SQL, so it's a worthwhile exercise to step through the statement above a bit more carefully. This is what's generally referred to as a "SELECT statement" and such statements are expected to contain—at a minimum—the SELECT key word followed by a range of columns (or "attributes" in the relational parlance) being selected, followed by the FROM keyword followed by the table (or relation) where those columns are to be found. In this command, we used an asterisk, which is a common SQL notation indicating that we want all columns of the target table. Thus, a minimal SELECT statement could read "SELECT * FROM Patients," which would return all the information in the Patients table. However, selecting all the information from a table without any conditions is rarely useful and, thus, the most minimal SQL statement you will likely see in practice also contains an additional WHERE keyword followed by one or more conditions specified with regard to some set of columns, such as WHERE FirstName = "Peter." Suffice it to say, all filters need not be strict equivalences. Instead, values can be filtered using a range of comparison operators as listed in Table 3.1.

Standard SQL also contains several logical operators, which can be invoked with special keywords and which can be used in conjunction with the earlier comparison operators to create rich filtering conditions. For example, both the AND and OR operators are used to combine

Table 3.1 Example valid SQL comparison operators.

Operator description	Operator symbol
Not equal to	!=
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=

multiple comparison clauses either additively or optionally. For example, if we wanted to be very specific with regard to which Peter we wanted to retrieve from the database, we could add an additional condition on the age column and require that both be true using the AND key word like so:

```
SELECT * FROM Patients WHERE FirstName = 'Peter' AND Age = 30
```

Let's take a moment to understand the relational mechanics going on here. When a SELECT statement such as this is issued, it targets a relation, here being the Patients relation, then it establishes a range of attributes along which to "project" meaning that it only considers these elements of the relation's tuples when applying filtering logic. In the case of SELECT * as we have earlier, such projection is unnecessary because we want all attributes of the Patients relation. However, we commonly want only some attributes such as only the FirstName and LastName columns. After projecting along a Relation's attributes, the filtering logic is applied to "restrict" the collection of projected tuples according to their predicate evaluation (e.g., only those for which the FirstName is "Peter"). Finally we "select" the tuples that evaluate as TRUE and return them in a new Relation containing the query results. Fig. 3.2 depicts this process graphically, color coding each step.

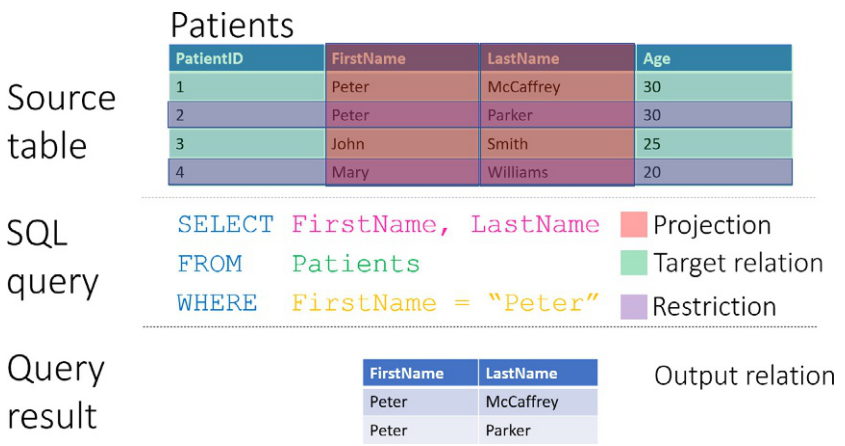

Fig. 3.2
Subcomponents of a conventional SELECT operation, including PROJECT and RESTRICT relational operations.

More specialized operators you are likely to encounter in practice include DISTINCT and COUNT, which are useful for exploring information in database tables. As the name implies, the DISTINCT key word is used to restrict a select statement to only unique values since, as we have seen, database columns can often contain many redundant values. If we were interested in knowing all of the unique first names in the FirstName column, we would employ the DISTINCT key word like so:

```
SELECT DISTINCT FirstName FROM Patients
```

which would return Peter and John because, though there are multiple rows in the table and multiple Peters and Johns, there are only really two unique first names in the FirstName column.

COUNT serves a different but often complementary purpose as it can be optionally applied to other statements to return a count of the values in a target column. If we wanted, for example, to know how many entries are in our patients table, we could use the COUNT key word like so:

```
SELECT COUNT(*) FROM Patients
```

And if we wanted to know how many unique last names were in our Patients table, we would combine COUNT and DISTINCT like so:

```
SELECT COUNT (DISTINCT LastName) FROM Patients
```

The last basic SQL key word we will introduce in this section is ORDER BY, which is used to organize the results returned from a SELECT statement. This ends up being quite a powerful convenience especially when exploring large tables. For example, if you were interested in discovering the upper and lower boundaries for the values in a column or if you were interested in ordering by a time stamp to see the most recent or the oldest entries. In our Patients table, we can sort our results in descending order of age like so:

```
SELECT * FROM Patients ORDER BY Age DESC
```

It's also possible to sort in ascending order using the ASC keyword and also to sort by multiple columns, which will nest the sorting operation by the order in which the columns were listed. Thus, if we wrote something like this:

```
SELECT * FROM Patients ORDER BY FirstName, Age DESC
```

We would have the results alphabetized by first name but, any entries that shared the same first name and were thus at the same alphabetical level would be further sorted by age.

The simple filter and select operations are universal in SQL and are the building blocks for subsequent functionality. Next, we will discuss a widely popular and essential key word in SQL: GROUP BY.

3.4 Basic SQL: GROUP BY and general aggregate functions

As we have seen, SELECT statements are powerful tools to fetch one or many rows in a table based upon conditions applied to the columns in that table. Using SELECT statements alone, we can craft many powerful queries using an assortment of evaluation operators. That being said, there is much more to the SQL language even when it comes to querying data within a single table. The GROUP BY statement is widely used and a critically important concept to understand, illustrating how SQL goes far beyond a simple retrieval language to become the rich query language its name implies.

We'll introduce the GROUP BY statement by example and then walk through its results. To begin, let's examine the structure of our Encounters table by peeking at the top two rows:

```
SELECT TOP 2 * FROM Encounters
```

And we can see that each row contains the date of a patient encounter along with the ID of each patient for whom that encounter took place:

```
_____________________________________________________
| EncounterID | Clinic    | Date        | PatientID|
_____________________________________________________
| 1           | Uptown    | 10/06/2019  | 1        |
| 2           | Uptown    | 12/07/2019  | 1        |
_____________________________________________________
```

There are many rows in our Encounters table and we certainly don't want to sift through them manually but we would like to know how many encounters occur per day and so we run the following query on our Encounters table:

```
SELECT COUNT(EncounterID), MAX(Date) FROM Encounters GROUP BY Date
```

And we get this in return:

```
_____________________
| Count   | Date      |
_____________________
| 10      | 10/06/2019 |
| 8       | 10/07/2019 |
| 9       | 10/08/2019 |
_____________________
```

To walk through what just happened, we need to understand the workflow of a GROUP BY operation. First, when this key word is used, there must be a column (or a group of columns) following it as those columns are used to create the groups. In this example, we used Date as the GROUP BY column, which splits the table up into groups of rows that share the same value for

the Date column. These don't need to be evenly sized groups, but they must share the same Date value. Second, COUNT and MAX (termed "aggregate functions" as they are special operators intended to summarize data in a group of rows) are applied to their respective column in each group of rows. Third and last, the results of the Count and Max aggregate functions calculated for each group are put together to form the results table that gets returned as depicted in Fig. 3.3. Thus, the statement can be read as "Create a table that contains the count of EncounterIDs and the maximum Date value for each group of rows in the Encounters table that share the same Date value." This workflow is very similar to the "split-apply-combine" workflow popularized in R and Python and similar to the MapReduce workflow popularized by Hadoop, which we will discuss in subsequent chapters.

If you are a bit confused by the use of MAX(Date), this is understandable. This is a common convention used to format the table returned from the query. When we create groups of rows using the date column, it's certainly true that each group will have identical values for that column. However, it will still be the case that there are multiple identical date values in that group and so, if we wish to present a single date value for each group in our results, we have to "flatten" that list to a single value. Since all values are the same, the MAX() aggregate function will pick a single value from the each group's set of identical date values. Alternatively, the MIN() aggregate function would accomplish this same purpose when referring to the column used to partition the groups.

The SQL standard contains many powerful aggregate functions such as percentile and correlation calculations, averages, and even regression slopes. It is increasingly common in practice to separate such computations—especially more advanced ones—from data storage and instead to do them in a different environment and with different tools such as Python. Toward the end of this chapter, however, we will cover a special advanced aggregate function known as WINDOW, which is fairly common to use in SQL queries.

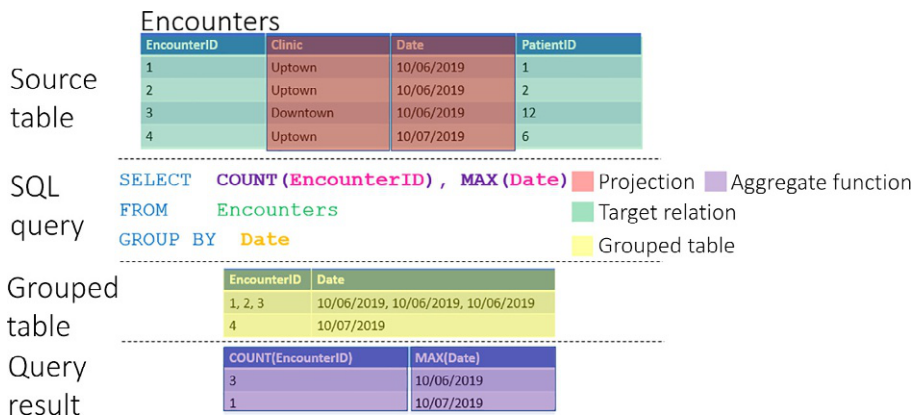

Fig. 3.3
Workflow for a SQL GROUP BY operation with aggregate functions.

3.5 Intermediate SQL: Joins

Now that we have discussed basic SELECT statements and, we can cover one more fundamental concept in SQL: Joins. Joins describe the combination of two or more tables and are used very commonly in practice to write effective queries. Joins are essential because many healthcare enterprise warehouses contain data spread across hundreds to thousands of different tables, thereby making it all but inevitable that any meaningful query will have to deal with data in several tables. Joins are the solution to that logistical challenge by allowing users to describe a query that fetches columns from separate tables while maintaining a logical integrity that keeps results from those separate tables organized.

You, as a reader, may reasonably find yourself wondering why joins are made necessary and why all databases do not consist of one large table. If so, consider the following vignette. Imagine that you are designing a medical record system and you wish to track vital signs, medications, inputs and outputs, telemetry, and several other attributes. If your database consists of one large table, then you will likely be forced into a table design wherein each row corresponds to an atomic observation and likely the most atomic observation. Thus, in this example, each cycle of the blood pressure cuff in an ICU bed would generate a new row for that patient in this table. However, being as there is only one table, all those patient's other attributes such as age, weight, room number, and insurance would also have to be duplicated on each of these new rows effectively copying the entire patient chart each time any new atomic piece of data is added to the table. Not only is this extremely inefficient from a storage perspective but, what's worse, imagine that you needed to update the patient's FirstName or LastName. Depending upon the acuity and activity involved in their chart, that patient's first and last names may exist in the table thousands of repeated times. This creates a real risk for database integrity where items of data are wantonly replicated and increases the risk of an incomplete update of a field such as this. Fig. 3.4 compares this kind of repetitive data model to a more normalized version spit across two tables, showing the reduction in redundant values this achieves.

In SQL syntax, a Join is declared by use of the JOIN key word and requires the provision of a shared column between joined tables, a purpose served by the primary and foreign keys discussed in the previous Chapter. Let's imagine that we wanted to return the first and last names of every patient who had an encounter on October 6, 2019. While ostensibly simple, this query does require data that spans both our Patients and our Encounters tables. In order to use the Date column from our Encounters table as a filter for the FirstName and Last Name values of our Patients Table, we need to JOIN these two tables on the shared key, the PatientID column, which links the two tables together like so:

```
SELECT Patients.FirstName, Patients.LastName
FROM Patients
JOIN Encounters
```

Patients

PatientID	FirstName	LastName	BloodPressure
1	John	Smith	128/78
1	John	Smith	120/60
1	John	Smith	130/82

Patients

PatientID	FirstName	LastName
1	John	Smith

BloodPressures

BloodPressureID	PatientID	BloodPressure
1	1	128/78
2	1	120/60
3	1	130/82

Fig. 3.4
Comparison of denormalized and normalized versions representing blood pressure data for patients before (top) and after (bottom) moving blood pressure recordings to their own table.

```
ON Patients.PatientID = Encounters.PatientID
WHERE Encounters.Date=10/8/2018
```

To step through the verbiage of this query, we first state which columns we want returned as well as which table we want them returned from and, at the end, the condition we want applied to filter those results. As a convention, since we are referring to more than one table, we explicitly state both the table name and the column name together with a dot separating them. This keeps it clear and organized as to which table we are referring to when we mention a column since different tables can have the same column names. In the middle of our query, we use the JOIN key word, stating the table we want to join to our Patients table (i.e., Encounters) and, using the ON key word, what columns in those joined tables we want to use to link them (i.e., PatientID). In plain English, this query would read "Select the first and last name columns from the patients table but do this only for those patients for whom their encounters in the Encounters table had a Date of 10/8/2018 and use the PatientID column in both tables to determine which Encounters apply to which Patients." This JOIN operation is depicted in Fig. 3.5. Hopefully, it is becoming clear how expressive and powerful SQL can be.

There are multiple forms of JOIN statements, which differ according to how they treat situations where the joined columns do not match cleanly. By default, when you use the JOIN key word, you are invoking an INNER JOIN, which will return only those rows where there was a matching PatientID value in the Patients and Encounters tables. Alternatively, you could specify an OUTER/FULL JOIN, which will return all rows in both tables, a LEFT JOIN, which will return all records in the first specified table (in this case Patients) along with only those records from the second table (in this case Encounters) for which the PatientID matches one in the Patients table, or a RIGHT JOIN, which is the opposite of a left join in that it will return all records in the second specified table (Encounters) while retuning only those from

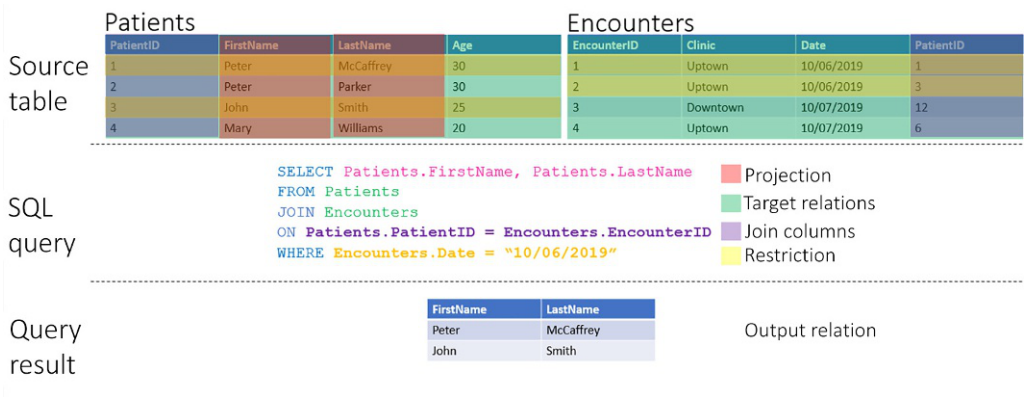

Fig. 3.5
Example SQL JOIN statement between two tables.

the first specified table (Patients) for which the PatientID matches one in the Encounters table. You are more likely to see INNER joins in practice but, for illustrative purposes, a motivation for using a LEFT JOIN, for example, would be if you wanted to return ALL patients, including those who have not yet had an encounter as those patients with no encounter would have no row in the Encounters table containing their PatientID.

3.6 Advanced SQL: Window functions

Finally, we will discuss an advanced SQL function, which you may encounter and which will prove quite useful. WINDOW functions are very similar to aggregate functions in that they apply some analytical logic to a group of rows. However, WINDOW functions do not operate within the groups established by a GROUP BY clause. Instead, WINDOW functions accept a partitioning clause, which is used to divide the table into groups of rows to which aggregate operations are applied. As an additional quirk, WINDOW functions are executed using the OVER and PARTITION BY key words. As an example, let's take a deeper look at the Encounters table:

```
_____________________
| PatientID | Date    |
_____________________
| 1         | 10/6/2018 |
| 2         | 10/6/2018 |
| 3         | 10/8/2018 |
| 2         | 10/7/2018 |
| 1         | 11/6/2018 |
| 2         | 12/1/2018 |
_____________________
```

Notice that PatientID 3 has one encounter, while PatientID 2 has three encounters and PatientID 1 has two encounters. Suppose we wanted to organize this table by PatientID, presenting the number of encounters and the first and last dates for each PatientID. The following WINDOW function would produce that for us:

```
SELECT PatientID, COUNT(Date), MIN(Date), MAX(Date)
OVER (PARTITION BY PatientID) CountDate
FROM Encounters
```

This would produce the following result:

PatientID	Count(Date)	MIN(Date)	MAX(Date)
1	2	10/6/2018	11/6/2018
2	3	10/6/2018	12/1/2018
3	1	10/8/2018	10/8/2018

3.7 SQL concept: Indexes

Heretofore, we've focused on the query linguistics around the SQL language partly because this is by far the most used part of a SQL knowledgebase for any informaticist but also because this is the most generalizable across specific RDBMS products. That being said, there are additional important concepts, which are very likely to come up in conversation with IT colleagues if not directly in analysis. Among these are indexes and schemas the first of which we address here.

As mentioned previously, composed SQL queries are ultimately reduced to a query plan that is often designed and optimized by internal components within a given RDBMS product. While this true, these query plans face a common bottleneck in being able to retrieve rows described in queries. For example, if we were to write a query asking for patient encounters between 7/4/2016 and 8/14/2016 at most medical centers, we're asking the database to retrieve a relative handful of rows from millions of pieces of information. A naïve algorithm may be for the database to simply iterate over every row in the Encounters table and test for whether that row's dates fall within the specified range. This is known as a "full table scan" and is prohibitively inefficient in almost every case. Instead, a more elegant approach is to make use of an index, which, as the name suggests, is an ancillary data structure that expedites the identification of relevant rows by storing precomputed mappings between rows and some relevant or "index" variable. This is very analogous to a book's index wherein the precomputed mapping is between an alphabetized list of important key words and their page number. Typically, when querying a large database, you will want to perform selection and filtration using indexed columns in order to benefit from a significant gain in efficiency and speed. Typically, indexed

columns are those which are most used and useful for formulating queries relevant to business needs. If querying for encounters using a date range, for example, it is very likely that the date column will be indexed as this is a very common attributes by which to filter rows. By contrast, if performing a research query for a more esoteric purpose such as all patients who had a particular insurer, it may very well rely on an unindexed column.

3.8 SQL concept: Schemas

Schemas are an important but relatively simple concept for users of any SQL database. Schemas can be thought of as a container of database objects, which include tables but also other things such as stored procedures and triggers (these latter two which are of less relevance here). Schemas serve a very important role of partitioning what would otherwise be a massive sea of tables into separate buckets, which may, therefore, have separate access policies, table structures, and import/export schedules. This is relevant to an informaticist in two ways. The first is that, when granted access to a SQL database, you will almost always needs to identify the schema that actually contains the tables of interest and either refer to it in queries or have an administrator provide access to that particular schema if such access hasn't already been provisioned. The second is that you may be granted the ability to create tables within an individualized schema provisioned for your user, allowing you to store certain summarizations of data within the database itself. Typically, this might be seen if one were wanting to report in repeated aggregate measures such as the daily average number of samples received in a clinical lab. In this case, the raw data necessary for the averages could be queried and the averages computed daily storing only the average itself in a separate table and thereby allowing for rapid summarization of the averages themselves (such as plotting moving averages over time).

3.9 Advanced SQL: SubQueries

As a final advanced SQL concept, subqueries offer a powerful additional layer of expression. In summary, a subquery can be thought of as a query within a query. Daunting at first, this is quite straightforward and best illustrated using an example. Imagine that we wanted to know how many patients who had an encounter on a certain day also had an echocardiogram performed by the Cardiology Fellow. Writing a single join statement, while workable, leaves much up to the RDBMS' query planner and runs the risk of the query planner executing a broad search across source tables in order to map out joined columns prior to filtering by selected dates. Instead, a more efficient and often more readable approach is to think of the final query as a query among the results of more basic queries. In this example, we have on the one hand a list of medical record numbers for patients who had an encounter within our target dates and on the other hand we have a list of medical record numbers for patients who had an echocardiogram performed by the cardiology fellow. Therefore, the final results are a matter of joining between these two tables on intersecting MRNs.

The subquery allows this because it lets us define a select statement on the fly within a larger select statement like so:

```
SELECT MRN FROM
       (SELECT MRN FROM Patients
        WHERE Encounters.Date > StartDate AND Encounters.Date >
         EndDate
        JOIN Encounters
        ON MRN) AS encounters
INNER JOIN
        (SELECT MRN FROM Patients
            WHERE Procedures.Physician == "Cardiology Fellow"
        JOIN Procedures
        ON MRN) as fellowProcedures
ON encounters.MRN = fellowProcedures.MRN
```

Take a minute to let the query sink in. The parentheses allow us to describe an entire SELECT statement where we would otherwise include a reference to a table forcing the query planner to explicitly break the query up into subparts and run them in order of dependency. More specifically, as the query planner maps out the query and encounters the SELECT statements in lieu of table references, it just executes the parenthetical SELECT statements and stores the results as an intermediate variable before dropping back into the original query and finishing It as a SELECT upon the intermediate tables.

3.10 Conclusion

Throughout this chapter, we have focused mainly in the ANSI standard definition of SQL. This can be thought of as a "core" SQL definition. However, many SQL products have their own extension of this core definition referred to as "dialects" of SQL often containing unique key words and powerful functions. For Microsoft, their dialect is called "Transact SQL" or T-SQL, for Oracle it's called PL/SQL, and for PostgreSQL it's called PL/pgSQL and each contain their own special enhancements to the language. Each of these dialects still enforces the ANSI SQL standard, thus allowing you to implement the key words and queries discussed in this chapter. It's also worth noting that there is a broad spectrum of analytical practice ranging from working entirely within SQL to performing simpler SELECTs and JOINs in SQL and subsequent analyses in another environment like Python. This book certainly advocates for the latter, but this is not universal practice nor is it required in order to produce meaningful analyses. It is most important to embrace each of these concepts and find a combination of tools that minimizes the time and effort required to get answers to analytical questions.

This page intentionally left blank