Chapter 1

Rows and Columns

Contents

1 What is a Relational Database

- A database is a collection of any data:
  - Could be text files, a list of names, phone numbers, audio files or images.
- This is quite broad – what makes a database *Relational*?
  - Cheap Answer: A database is relational if it follows the *Relational Data Model*
  - Longer Answer: The Relational Model is a system of storing and organizing data proposed by Edgar Codd in the 1960's and 70's. Codd proposed a set of 12 rules.[1] Any database system which follows these rules has a number of important features:[2]
- Codd's 12/13 Rules:

   0. **The foundation rule:** For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.
   1. **The information rule:** All information in a relational data base is represented explicitly at the logical level and in exactly one way – by values in tables.
   2. **The guaranteed access rule:** Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.
   3. **Systematic treatment of NULL values:** NULL values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.
   4. **Dynamic online catalog based on the relational model:** The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.
   5. **The comprehensive data sublanguage rule:** A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all of the following items:
      - Data definition.
      - View definition.
      - Data manipulation (interactive and by program).
      - Integrity constraints.
      - Authorization.
      - Transaction boundaries (begin, commit and rollback).
   6. **The view updating rule:** All views that are theoretically updatable are also updatable by the system.

[1]Like a good computer scientist, his rule list starts at zero, so there are actually 13 rules.
[2]These descriptions below were copied from Wikipedia, which I assume copied from the original paper.

7. **Possible for high-level insert, update, and delete:** The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data.

8. **Physical data independence:** Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods.[3]

9. **Logical data independence:** Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables.[4]

10. **Integrity independence:** Integrity constraints specific to a particular relational data base must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.

11. **Distribution independence:** The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only.

12. **The nonsubversion rule:** If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time).

- While there are many ways that these rules could be executed, the modern "Relational Database" is quite standard. On the language side, SQL is used as the primary programming language and a set of objects, discussed below, are used to conform to the rest of the rules.[5]

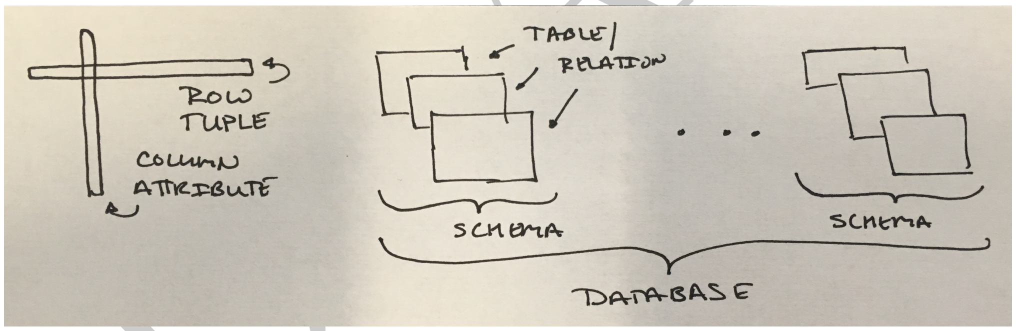

Figure 1.1: Database Objects

- Relational databases consist of the following objects:
  - A **tuple**, or **row**, is a single observation, like you would find in Excel.
  - An **attribute** of a tuple is a **column** and is defined by both a **name** and **type**.
  - A **relation** or **table** is a collection of rows and their attributes. It is comparable to a worksheet in Excel. Importantly, **tuples are not ordered in a relation** or, to state it alternatively, row order is arbitrary.

[3]The physical level consists of the physical storage of the data. Things like compression type, adding or removing a new hard drive or changing the directory that the database is stored in are considered "Physical Level".

[4]The logical level consists of the concept level logic sitting on-top of the physical layer. This includes adding and subtracting columns, changing an index or otherwise modifying how the information is stored without making underlying changes to the data in the database.

[5]While there is chatter about the "exact" specification of what an RDMS is and whether modern database systems actually follow the rules – that conversation is not of interest to practitioners.

  - A collection of relations is a **schema** and can be compared to an Excel workbook with multiple worksheets. Taking the Excel analogy further, just as it is straightforward to access data on different sheets, but within the same Excel file, it is straightforward to access data in a database that is within the same schema.
  - A collection of schemas is a **database**. A database can be thought of a directory full of Excel files, each with their own set of worksheets. Continuing the Excel analogy: while data *can* be accessed between different Excel files, it can be difficult; that same principle applies to databases: accessing data in two different schemas is not trivial.

- Modern relational databases use "dot" notation when referring to items in a database:

  database.schema.table.column

  Since rows are unnamed in relational databases there is no method to reference a row.

- The data stored in a relational database is strongly *typed.* Each column of data is defined as a certain *type* (think integer or string) which is defined by the operations that can act on it. For example, you can multiply numbers, but you cannot do the same to strings.

- Relational Databases work well when you have *structured* data or data with a high-degree or organization. For example, information such as "age", "first name", "last name" and "occupation" are easily stored in relational databases. Data without consistent structure, such as those collected from multiple sources or those with complex structures, such as the information contained within a personal history, tend to best stored in other structures.

- Relational databases are accessed using "Structured Query Language" ("SQL") and pronounced "Sequel".

- Importantly, relational databases are *boring.* They are designed to be warehouses of data, but NOT to do data analysis. The most common use case for relational databases is that they store information that is too large to put on a single computer. When a user or process wishes to do analysis, they connect to the database and extract only the information necessary to do their work. In other words, the database is useful for storing and accessing data, but not for doing any data science or analytics. In SQL there are no functions for "logistic regression", "random forrest", "anova" or any other common standard statistical techniques.

- Relational databases are generally *client-server* systems. In order to access data within a database the server must be up and running and a client needs to be configured to interact with it. This can happen in many different ways:
  - Both the client and server can be running on the same physical computer.
  - The server can be in the cloud and a user may be anywhere in the world.
  - The server could be in a storage room or under a desk and the client could be somewhere in the office.

- The parameters required to connect to most SQL systems include a (1) a host, (2) a username, (3) a password (4) a port to connect on and (5) the database name. The combination of these four items is referred to as a "connection string".

- When we refer to different SQL variants, such as MS-SQL, Vertica or MySQL, we are talking about the program that is run on the server, not on the client. There are clients that can connect to multiple server variants and there are clients which are specific to ones.

- There are many clients for SQL. One you may consider is DataGrip, by the makers of PyCharm, so if you like using that interface, use it. Others that I tend to recommend include PopSQL and Postico.[6]
- SQL is a *declarative* programming language. In a declarative language, the programmer describes the output while not explicitly detailing each step to create that output. The opposite of declarative programming is *imperative* programming which focuses on how a program should operate in a step-by-step manner.[7]

2 Selecting Columns

- **Iowa Cars:** The first dataset that we will consider is a database of car registrations from Iowa. Details about this dataset, including how to load it and its data dictionary can be found in Appendix A, Section 2.
- Let's begin with the following query template:

```
SELECT ______________  <- Comma separated List of Columns
FROM ______________ <- Table From which columns are selected from
; <- Sometimes necessary
```

- The query begins with the SELECT clause which states which columns of data we are interested in having our database return.
- The FROM clause tells the database which table to look for the data .
- Why is the ";" only "sometimes" necessary? It depends on your SQL client. The semi-colon represents the end of a statement and some SQL clients will automatically place one. The use of a semi-colon is highly recommended!
- Let's say that we have the cls.cars table structure and we want to select the "CountyName" column. We would write the following query:

```
select CountyName from cls.cars;
```

  which would return a single column and all 41,202 from the table.
- SQL ignores hard returns, additional white space and is generally case insensitive[8] so we could write the above query like this:

```
SeLeCT
        CountyNAMe                  FRom
CLS.cARS;
```

  and it would return the same data as above.
- Selecting multiple columns is straightforward:

```
select countyname, annualfee
from cls.cars;
```

[6]This site `https://wiki.postgresql.org/wiki/PostgreSQL_Clients` contains an up to date list of clients which are usable in PostgreSQL.
[7]Languages such as Python and R are generally considered imperative.
[8]There are a few exceptions to case insensitivity, but they are outside the scope of this class.

which will return two columns and all rows.

- To return **all** columns, we use an "*":

```
select * from cls.cars;
```

which will return all rows and all columns from the database.

- Be careful with the above command, SQL servers are powerful and assume you know what you are doing. If you ask for all the data from a large table... it will give you all of the data.

3 WHERE: Filtering rows

The select operator chooses which columns to return to the client while the WHERE filters out rows based on their contents.

- The WHERE syntax uses standard Boolean style logic to evaluate row-inclusion on a row-by-row basis.

- The WHERE clause occurs after the FROM clause, such as in this example:

```
select countyname, vehicletype
from cls.cars
WHERE vehicletype = 'Semi Trailer';
```

which which returns 1,683 rows.

- We can select all columns which fulfill a certain criteria:

```
select * from cls.cars where vehicletype = 'Semi Trailer';
```

which returns all columns with this vehicle type.

- Strings themselves, such as 'Semi Trailer', are case-sensitive, so the query

```
select countyname
from cls.cars
where vehicletype = 'Semi TRAiler';
```

will return zero rows.[9]

- When selecting strings we use single quotes. Double-quotes should be reserved for referencing database objects (such as tables, columns and schemas). Most databases allow users to define objects with special characters, such as spaces. We use double-quotes in these situations to refer to these objects.[10] For example, if a database had a column with a space in it, such as "user name" then to select that column would require using double quotes:

```
select "user name" from table;
```

[9]Surprisingly, this is not true for all variants of SQL. MySQL, which may be the most popular variant of SQL, is case-insensitive by default.

[10]Personally, I strongly believe that special characters should be avoided when defining database objects, but it sometimes occurs.

- With numbers, we can use any standard comparison operator (=, >, <, <=, >=) to select rows, such as in the below,

  ```
  select * from cls.cars where registrations > 10000;
  ```

  returns 1,248 rows.

- There are two ways to state inequality: “! =” or “<>”.

- To combine multiple criteria we use “AND” and “OR” clauses:

  ```
  select
      *
  from
      cls.cars
  where
      registrations > 1200
      and countyname = 'Wright';
  ```

  returns 70 rows, while

  ```
  select * from cls.cars
  where
      registrations > 1200
      and registrations <= 3000
      and countyname = 'Wright';
  ```

  returns 21 rows, or all information from Wright county where the registrations are between 1,200 and 3,000 (inclusive). A more complicated example,

  ```
  select * from cls.cars where
      (
      (registrations > 1200 and registrations <= 3000)
      or
      (registrations > 4000 and registrations <= 4200)
      )
      and countyname = 'Wright';
  ```

  returns 24 rows.

- Parenthesis dictate the order of operations, so this query returns all rows from Wright County with between 1,200 and 3,000 registrations or 4,000 and 4,200 registrations. To understand this query, consider the following potential values, with how they would evaluate in the above condition:

Countyname	Registrations	Yes or No
Wright	2000	Yes
Wright	4100	Yes
Right	3500	No
Wright	5000	No

4 Null

- Relational databases use *3-value* logic. Unlike some logic systems which solely consist of True and False, 3-value logic systems include a third value: Null. Null is sometimes sometimes called “unknown” or “missing”, but be careful about this phrasing as different data sources may represent those concepts separately.

- The truth tables describes how the three value logic behaves. Tables 1.2, 1.3 and 1.4 demonstrate how Null is handled in a range of situations – the key things to keep in mind is that Null does not evaluate Null, it is it’s own, separate value.

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Figure 1.2: AND 3-value logic

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Figure 1.3: OR 3-value logic

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Figure 1.4: NOT 3-value logic

- Most conditional expressions are predicated on “Success” being “True”, which means that Null *tends* to behave like “False”. For example, in a WHERE clause, if the resulting expression evaluates Null it is *not* returned. So this query, which returns Null for every row, will yield zero rows:

```
select * from cls.cars where null;
```

- Null thus represents a special class of data in the database. Consider the following query, which returns 3 rows of data:

```
SELECT
    year, registrations, annualfee
FROM
    cls.cars
WHERE
    registrations > 217000
    and year < 2010;


  year    registrations    annualfee
------  ---------------  -----------
  2006           218883
  2005           218235
  2008           217073   24160802.0
```

- The first two rows in “annualfee” are null. Let’s put a further restriction on this columns and see what happens:

```
SELECT
    year, countyname, registrations, annualfee
FROM
    cls.cars
WHERE
    registrations > 217000
    and year < 2010
    and annualfee < 25000000;


  year  countyname      registrations    annualfee
------  ------------  ---------------  -----------
  2008  Polk                   217073   24160802.0
```

  This query will return 1 row (the third row from the previous query). In particular, Null annual fees are not evaluated to be true. Just to be clear, the only row that will be returned by this query, of the three that were returned will be the third row which did *not* have a Null annualfee.

- The query above only returned the non-Null annualfee rows when we put a restriction that annualfee had to be *less* than 25,000,000. Let’s put a restriction in the opposite direction, this time only returning rows *greater* than 25,000,000:

```
select
    year, countyname, registrations, annualfee
from
    cls.cars
where
    registrations > 217000
    and annualfee >= 25000000
    and year < 2010;


year    countyname    registrations    annualfee
------  ------------  ---------------  -----------
```

  This query returns zero rows!

- So, if we have no restriction on annualfee it returns 3 rows. If we restrict annualfee to above 25,000,000, it returns 1 row and if we restrict annualfee to below 25,000,000 then it returns 0 rows. What happened? A key relational databases feature is the following:

  NULL behaves like FALSE in WHERE CLAUSES!

- In the previous example, for the first and second row the database evaluated NULL $\geq$ 25,000,000 and said that this is not true. The database also evaluated Null $<$ 25,000,000 and found it to be false. In other words, Null evaluates False in a boolean expression.

- Since Null behaves like False we need to use a different operator to compare it; this operator is `IS`:

```
select
    year, countyname, registrations, annualfee
from
    cls.cars
where
    registrations > 217000
    and annualfee IS NULL
    and year < 2010;


  year  countyname      registrations  annualfee
------  ------------  ---------------  -----------
  2006  Polk                   218883
  2005  Polk                   218235
```

which will return the two rows with a Null annualfee.

- To remove NULL values we use the `IS NOT` expression:

```
select
    year, countyname, registrations, annualfee
from
    cls.cars
where
    registrations > 217000
    and annualfee is not NULL
    and year < 2010;


  year  countyname      registrations    annualfee
------  ------------  ---------------  -----------
  2008  Polk                   217073   24160802.0
```

which will return only the row with a non-Null annualfee value.

- What about this query?

```
select
    year, countyname, registrations, annualfee
from
    cls.cars
where
    registrations > 217000
    and annualfee = NULL;
```

Ha! Trick question: this will return zero rows because Null always evaluates False!

5 ORDER BY and LIMIT

- To sort the data that is returned by a query we use an ORDER BY clause:

```
select
    registrations, *
from
    cls.cars
order by registrations ASC;


  registrations    year  countyname    motorvehicle    vehiclecat    vehicletype  [...]
---------------  ------  ------------  --------------  ------------  ------------ [...]
              1    2015  Poweshiek     Yes             Truck         Tractor/Truc [...]
              1    2014  Dallas        Yes             Truck         Tractor/Truc [...]
              1    2014  Jefferson     Yes             Truck         Tractor/Truc [...]
              1    2013  Tama          Yes             Truck         Truck Tracto [...]
              1    2011  Wapello       Yes             Truck         Truck Tracto [...]
[...]
```

will return every column of data in the database ordered by registrations from low-to-high (Ascending).

- Data can also be sorted "Descending" – from high-to-low:

```
select
    registrations, countyname, year
from
    cls.cars
order by registrations DESC;


  registrations  countyname      year
---------------  ------------  ------
         218975  Polk            2015
         218883  Polk            2006
         218235  Polk            2005
         218211  Polk            2014
         217540  Polk            2016
[...]
```

Running this query will return the Null annualfee rows first! Because annualfee Nulls are sorted as if they are larger than non-Nulls, this implies that descending order returns them before non-Null values.

- ORDER BY also works with character columns:

```
select
    countyname
from
    cls.cars
order by countyname desc;


countyname
------------
Wright
Wright
Wright
Wright
Wright
[...]
```

will return the data, sorted by countyname ascending. For characters, DESC is reverse alphabetical order, while ASC is alphabetical order. PostgreSQL sorts character data in a case insensitive manner.

- Under the default sort order (Ascending), Null values are *last*, which means that they are treated as if (a) they are larger than non-Nulls, when numbers and (b) alphabetically last.
- The default sort order, if neither ASC or DESC is specified is ASC.
- Multiple sort orders can be combined:

```
select
    countyname, year
from
    cls.cars
order by countyname desc, year asc;


countyname      year
------------  ------
Wright          2005
Wright          2005
Wright          2005
Wright          2005
Wright          2005
[...]
```

will return data sorted by countyname first, and then within common countyname's sorted by year (ascending).

- In order to limit the number of rows returned to the client we use the LIMIT command. For example, the following query:

```
select
    *
from
    cls.cars
limit 10;


  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2008  Ida           Yes             Bus           Bus                           [...]
  2011  Jasper        Yes             Moped         Moped                         [...]
  2012  Harrison      Yes             Truck         Truck           3 Tons        [...]
  2015  Palo Alto     No              Trailer       Travel Trailer                [...]
  2016  Adair         Yes             Truck         Truck           3 Tons        [...]
[...]
```

returns every column in the cars table, but only 10 rows. Note that this is **not** the first 10 rows, it is an *arbitrary* 10 rows. If you run the query again you may get a different set of rows!

- We can combine LIMIT with the rest of our commands:

```
select
    registrations, annualfee
from
    cls.cars
where
    countyname = 'Scott'
limit 100;


  registrations    annualfee
---------------  -----------
          31626    1700000.0
              3        376.0
           1779     553072.0
            172      40480.0
           2972     135942.0
[...]
```

will return 100 rows of data. Note that limit is evaluated after the where command, so as long as there are 100 rows with countyname = ‘Scott’, this will return 100 rows.

The LIMIT clause is not present in all versions of SQL. Other SQL variants use alternative syntax to limit the rows returned. The following table highlights those differences and presents a query using each version:

Variant	Syntax	Example
MySql	LIMIT	select * from cls.cars order by annualfee asc limit 10;
MS-SQL	TOP	select top 10 * from cls.cars order by annualfee asc;
Older Oracle	rownum	select * from (select * from cls.cars order by annualfee asc) where rownum <= 10;

- A common use case for ORDER BY and LIMIT is to return the top X of something, like the query below, which returns the five largest registration values:

```
select
    registrations
from
    cls.cars
order by registrations desc
limit 5;


  registrations
---------------
         218975
         218883
         218235
         218211
         217540
```

- Some other examples: Top 15 rows ordered by registrations:

```
select
    *
from
    cls.cars
order by registrations desc
limit 15;


  year  countyname    motorvehicle    vehiclecat    vehicletype    tonnage      r [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
  2015  Polk          Yes             Automobile    Automobile                    [...]
  2006  Polk          Yes             Automobile    Automobile                    [...]
  2005  Polk          Yes             Automobile    Automobile                    [...]
  2014  Polk          Yes             Automobile    Automobile                    [...]
  2016  Polk          Yes             Automobile    Automobile                    [...]
[...]
```

- Top-15 rows, countyname only, where annualfee is not equal to zero.

```
select countyname
from cls.cars
where annualfee <> 0
order by registrations desc limit 15;


countyname
------------
Polk
Polk
Polk
Polk
Polk
[...]
```

This query demonstrates that (1) we can order by columns that are not selected and (2) we can remove both NULL and zero annualfees.[11]

6 Column Numbering

- In this section we introduce an SQL feature that make your life easier: using column number order in an ORDER BY clause.
- Instead of putting the column name, we can use the column number to specify how we should order the rows:

```
select
    registrations
    , countyname
from
    cls.cars
where
    annualfee <> 0
order by 1 desc
limit 15;


  registrations  countyname
---------------  ------------
         218975  Polk
         218211  Polk
         217540  Polk
         217073  Polk
         216792  Polk
[...]
```

In the query above the "1" in the ORDER BY refers not to the number 1, but to the first column position, which in this case is "registrations". We can also use ASC, DESC and multiple sort orders using this notation:

[11]Since NULL <> is always false this removes all the null annualfee values.

```
select
    registrations
    , countyname
from
    cls.cars
where
    annualfee <> 0
order by 2, 1 desc;


  registrations  countyname
---------------  ------------
           4247  Adair
           4137  Adair
           4109  Adair
           3916  Adair
           3768  Adair
[...]
```

The data from this example is sorted by countyname first and then, within each county sorted by the number of registrations. There are only a few operators that allow for column numbering. The ORDER BY operator is one of them.

- The downside of using column numbering syntax is that if you change the select statement, such as adding or subtracting a column the query will not raise an error, but will no longer return the same data. For this reason a number of organizations avoid using this syntax and require full column name expressions at all times.

7 Where are we: A Note on Scope

- A common issue that new SQL users have is how much detail to include when writing a query. For example, if we assume that the database name is "sql_class", then the following two queries will return the same data, though they look very different:

```
select sql_class.cls.cars.year from cls.cars;
```

and

```
select year from cls.cars;
```

- We call this issue *scope*. Scope is the region of a query where a particular name is valid. The standard way that we write queries is to include the schema and table in the FROM clause, but only include the column names in the rest of the query, unless those column names are not fully specified.
- Note that the table was written as "cars" and not "cls.cars" or something even more specific. This may return an error for you depending on how your "search path" is set. The search path represents where the database looks, by default, for tables. If you are working with data that is not in your search path, you will need to include the schema name, otherwise the database will return an error. In the notes for this class we will ignore issues surrounding this.[12]

[12]More information here: `https://www.postgresql.org/docs/current/static/ddl-schemas.html`