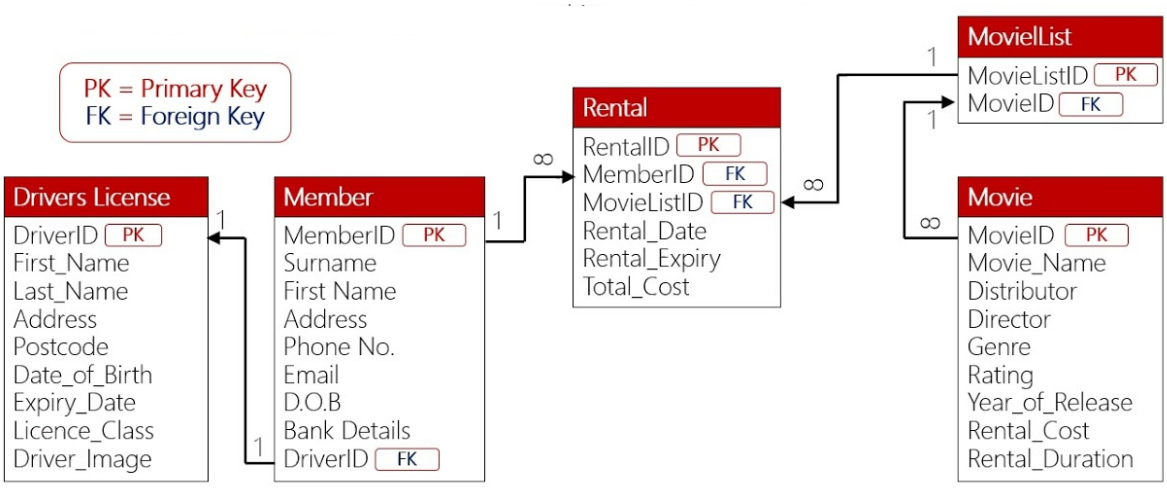
SQL Statistics 506

SQL and Relational Databases

A **relational database** is a set of rectangular data frames called **tables** linked by **keys** relating one table to another. Software implementations of such data structures are known as *relational database management systems* (RDBMS). Most RDBMS use **structured query language** or **SQL** ("sequel" or "S-Q-L") to modify or search the relational database.

Here is an example of a relational database. "Primary" keys are ones which uniquely identify rows of a particular table; "foreign" keys simply refer to "primary" keys in other tables. A key can contain multiple variables.



SQL provides a syntax for interfacing with relational data. It is largely a declarative language in that we use SQL to specify *what* we wish to accomplish, leaving the *how* to the RDBMS. While there are standards for SQL implementations put out by the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI), there are several open source and commercial implementations that each have unique features.

I will try to focus on the commonalities, but will be using an SQLite engine in R for providing examples. One unique feature of SQLite is that it does not follow the client-server model. In

this model, a physical computer storing the data and executing queries within the RDMBS, the *server*, is separate from the machine requesting the queries known as the *client*.

The client-server model is popular in business, health care, and other domains as it allows security and monitoring of how the data is queried. It is also popular for many large open data projects (i.e ensemble) where it is beneficial for data to be centrally maintained and frequently accessed on the fly.

For the examples in class, we will use SQL to access smaller data-sets for which there are more efficient approaches. In real world scenarios, databases can be extremely large (multiple gigabytes or larger) that would be difficult to access directly in R.

There are a wild number of SQL-related packages in R - some for connecting to databases, some for sending SQL queries. We will primarily use the "DBI" package. There may be others which are better for interfacing with particular SQL databases, but generally the SQL syntax should not differ across packages.

Lahman Example

For our examples, we will use the "Lahman" dataset which contains historical baseball data from 1871-2022. It can be downloaded from https://github.com/jknecht/baseball-archive-sqlite.

(Image from https://relational.fit.cvut.cz/dataset/Lahman.)

The *RSQLite* package contains the backend required to load up an SQLite database. The *DBI* package interfaces with any database, in this case the SQLite data.

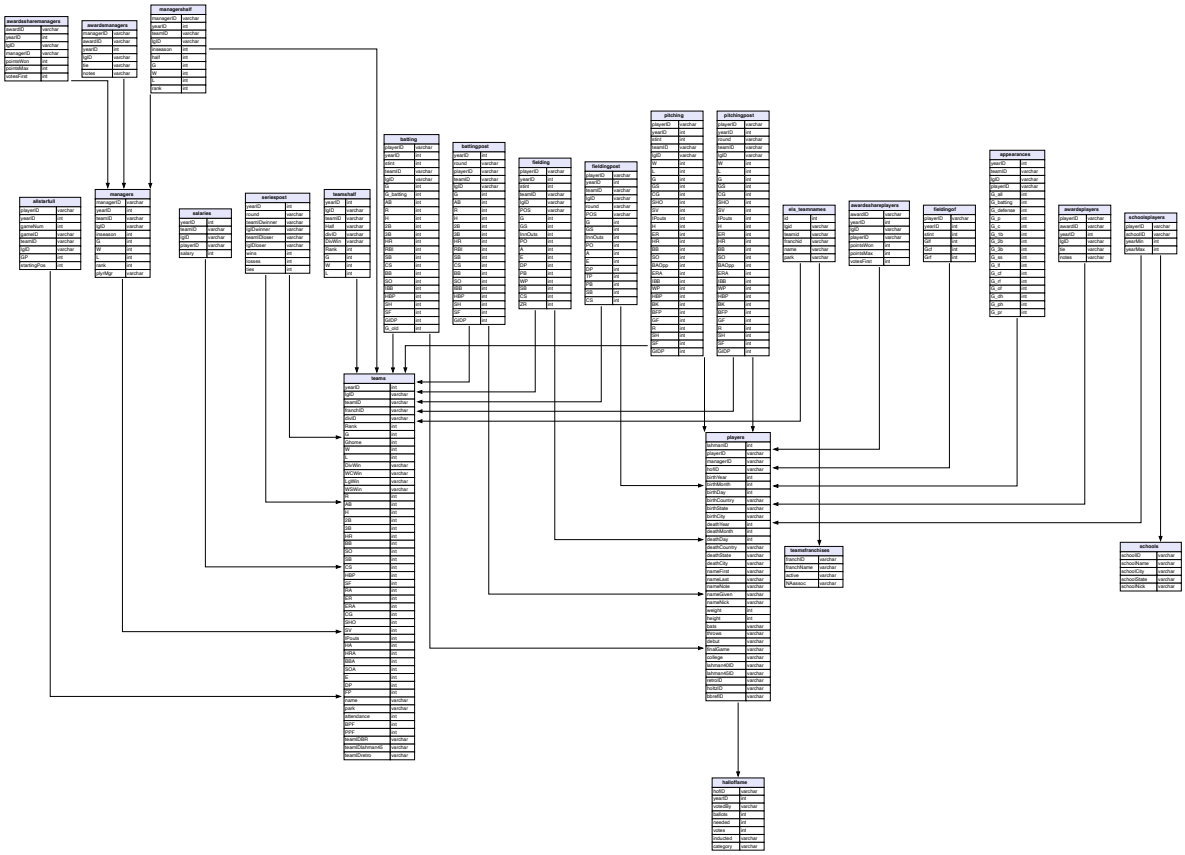
```
# Packages
library(DBI)     # For interfacing with a database

# Import the SQLite database of the Lahman data
lahman <- dbConnect(RSQLite::SQLite(), "data/lahman_1871-2022.sqlite")
lahman
```

```
<SQLiteConnection>
  Path: /Users/josh/repositories/_teaching/506-f23/data/lahman_1871-2022.sqlite
  Extensions: TRUE
```

We now have an SQLite database we can work with. Let's start by getting a list of all tables contained within the database.

```
dbListTables(lahman)
```


awardssharemanagers
awardsmanagers
managerawards
allstarfull
managers
salaries
seriespost
teamshalf
batting
battingpost
fielding
fieldingpost
pitching
pitchingpost
teams
players
ats_teamnames
awardsshareplayers
fieldingpost
appearances
awardsplayers
schoolsplayers
teamsfranchises
schools
halloffame

```
 [1] "AllstarFull"         "Appearances"         "AwardsManagers"
 [4] "AwardsPlayers"       "AwardsShareManagers" "AwardsSharePlayers"
 [7] "Batting"             "BattingPost"         "CollegePlaying"
[10] "Fielding"            "FieldingOF"          "FieldingOFsplit"
[13] "FieldingPost"        "HallOfFame"          "HomeGames"
[16] "Managers"            "ManagersHalf"        "Parks"
[19] "People"              "Pitching"            "PitchingPost"
[22] "Salaries"            "Schools"             "SeriesPost"
[25] "Teams"               "TeamsFranchises"     "TeamsHalf"
```

We can also dive into a particular table and get a list of all columns.

```
dbListFields(lahman, "Batting")
```

```
 [1] "playerID"  "yearID"    "stint"     "teamID"    "lgID"      "G"
 [7] "G_batting" "AB"        "R"         "H"         "2B"        "3B"
[13] "HR"        "RBI"       "SB"        "CS"        "BB"        "SO"
[19] "IBB"       "HBP"       "SH"        "SF"        "GIDP"      "G_old"
```

Working with tables

Search, subset, and limiting clauses

The basic structure of a SQL query contains a `SELECT` statement indicating which columns are desired and a `FROM` clause explaining where to find them (as we saw above).

```
dbGetQuery(lahman, "SELECT playerID FROM Batting LIMIT 5")
```

```
   playerID
1 aardsda01
2 aardsda01
3 aardsda01
4 aardsda01
5 aardsda01
```

The string, `"SELECT playerID FROM Batting LIMIT 5"` is an **SQL query**. `SELECT` is the **statement**, and it has the basic syntax of

```
SELECT var1, var2 FROM table
```

We add an additional **clause**, `LIMIT 5` to reduce the amount of output. This is good practice when developing new queries as it prevents large wait times only to discover a bug.

You can use a wild card `*` to select all columns in a table:

```
dbGetQuery(lahman, "SELECT * FROM Batting LIMIT 5")
```

```
   playerID yearID stint teamID lgID  G G_batting AB R H 2B 3B HR RBI SB CS BB
1 aardsda01   2004     1    SFN   NL 11        NA  0 0 0  0  0  0   0  0  0  0
2 aardsda01   2006     1    CHN   NL 45        NA  2 0 0  0  0  0   0  0  0  0
3 aardsda01   2007     1    CHA   AL 25        NA  0 0 0  0  0  0   0  0  0  0
4 aardsda01   2008     1    BOS   AL 47        NA  1 0 0  0  0  0   0  0  0  0
5 aardsda01   2009     1    SEA   AL 73        NA  0 0 0  0  0  0   0  0  0  0
  SO IBB HBP SH SF GIDP G_old
1  0   0   0  0  0    0    NA
2  0   0   0  1  0    0    NA
3  0   0   0  0  0    0    NA
4  1   0   0  0  0    0    NA
5  0   0   0  0  0    0    NA
```

Note that by convention, the keywords in SQL queries are capitalized, but SQL is itself not case sensitive, so this works as well:

```
dbGetQuery(lahman, "select * from batting limit 5")
```

```
   playerID yearID stint teamID lgID  G G_batting AB R H 2B 3B HR RBI SB CS BB
1 aardsda01   2004     1    SFN   NL 11        NA  0 0 0  0  0  0   0  0  0  0
2 aardsda01   2006     1    CHN   NL 45        NA  2 0 0  0  0  0   0  0  0  0
3 aardsda01   2007     1    CHA   AL 25        NA  0 0 0  0  0  0   0  0  0  0
4 aardsda01   2008     1    BOS   AL 47        NA  1 0 0  0  0  0   0  0  0  0
5 aardsda01   2009     1    SEA   AL 73        NA  0 0 0  0  0  0   0  0  0  0
  SO IBB HBP SH SF GIDP G_old
1  0   0   0  0  0    0    NA
2  0   0   0  1  0    0    NA
3  0   0   0  0  0    0    NA
4  1   0   0  0  0    0    NA
5  0   0   0  0  0    0    NA
```

To obtain the number of rows in a table, we can use the `COUNT()` function:

```
dbGetQuery(lahman, "SELECT COUNT(*) FROM Batting")
```

```
  COUNT(*)
1   112184
```

A quick loop can tell us the size of all tables:

```
for (t in dbListTables(lahman)) {
  rows <- dbGetQuery(lahman, paste("SELECT COUNT(*) FROM", t))
  cols <- length(dbListFields(lahman, t))
  print(paste(t, "-", rows, "x", cols))
}
```

```
[1] "AllstarFull - 5516 x 8"
[1] "Appearances - 112106 x 21"
[1] "AwardsManagers - 179 x 6"
[1] "AwardsPlayers - 6531 x 6"
[1] "AwardsShareManagers - 425 x 7"
[1] "AwardsSharePlayers - 6879 x 7"
[1] "Batting - 112184 x 24"
[1] "BattingPost - 16374 x 22"
[1] "CollegePlaying - 17350 x 3"
[1] "Fielding - 149365 x 18"
[1] "FieldingOF - 12028 x 6"
[1] "FieldingOFsplit - 35315 x 18"
[1] "FieldingPost - 15540 x 17"
[1] "HallOfFame - 4323 x 9"
[1] "HomeGames - 3200 x 9"
[1] "Managers - 3718 x 10"
[1] "ManagersHalf - 93 x 10"
[1] "Parks - 255 x 7"
[1] "People - 20676 x 25"
[1] "Pitching - 50402 x 30"
[1] "PitchingPost - 6538 x 30"
[1] "Salaries - 26428 x 5"
[1] "Schools - 1207 x 5"
[1] "SeriesPost - 378 x 9"
[1] "Teams - 3015 x 48"
[1] "TeamsFranchises - 120 x 4"
[1] "TeamsHalf - 52 x 10"
```

Limiting Clauses

Aside from `LIMIT`, there are more nuanced and powerful ways of extracting specific entries.

WHERE

We can use Boolean comparisons in a `WHERE` clause as shown in the example below. We find all player-seasons since 2000 in which the player was credited with an RBI 100 or more times.

Here is our query:

```
SELECT playerID, yearID, RBI
  FROM batting
 WHERE RBI >= 100 AND yearID >= 2000
```

And here it is in R:

```
## Get all 100+ RBI seasons since 2000
head(dbGetQuery(lahman, "
SELECT playerID, yearID, RBI
  FROM batting
 WHERE RBI >= 100 AND yearID >= 2000
"))
```

```
   playerID yearID RBI
1 abreubo01   2001 110
2 abreubo01   2003 101
3 abreubo01   2004 105
4 abreubo01   2005 102
5 abreubo01   2007 101
6 abreubo01   2008 100
```

(Note that the spacing here is stylistic - in both SQL and `dbGetQuery` splitting onto multiple lines and aligning on the first space do not affect the query, but do make it easier to read!)

We of course could use `LIMIT` instead of `head()` to keep it entirely inside the query. In addition, `LIMIT` will be much faster than `head()`:

```
library(microbenchmark)
microbenchmark(
  head  = head(dbGetQuery(lahman, "SELECT playerID FROM batting")),
  limit = dbGetQuery(lahman, "SELECT playerID FROM batting LIMIT 6"))
```

```
Warning in microbenchmark(head = head(dbGetQuery(lahman, "SELECT playerID FROM
batting")), : less accurate nanosecond times to avoid potential integer
overflows
```

```
Unit: microseconds
  expr       min        lq       mean     median         uq       max neval cld
  head 13709.990 13784.385 14000.0289 13837.1925 13917.4295 16826.523   100  a
 limit   101.106   102.664   106.5336   104.8985   107.5635   143.787   100   b
```

`IN`

To select on a column by testing against a set of fixed values use `IN`. We also rename the columns as we gather them, leaving the original data alone

```
SELECT nameGiven AS given, nameLast AS last, birthYear
  FROM master
 WHERE nameLast IN ('Alou', 'Griffey')
```

Use single quotations (') instead of double (") - in some flavors of SQL, double quotes have different meaning (if they're used at all), whereas it is consistent that single quotes start and end strings. In addition, this avoids conflict if you use double quotes to start the string in R, and single quotes in the actual query.

```
dbGetQuery(lahman, "
SELECT nameGiven AS given, nameLast AS last, birthYear
  FROM People
 WHERE nameLast IN ('Alou', 'Griffey')
")
```

```
              given    last birthYear
1      Felipe Rojas    Alou      1935
2 Jesus Maria Rojas    Alou      1942
3       Mateo Rojas    Alou      1938
4      Moises Rojas    Alou      1966
5    George Kenneth Griffey      1950
6    George Kenneth Griffey      1969
```

LIKE

Use a `LIKE` predicate with a `WHERE` clause to get partial string matching. You can use `%` to match any sub-string.

```
SELECT nameGiven AS given, nameLast AS last, birthYear
  FROM People
 WHERE nameLast LIKE '%riff%'
```

```
## Find all players with last name containing a 'riff' sub-string
dbGetQuery(lahman, "
SELECT nameGiven AS given, nameLast AS last, birthYear
  FROM People
 WHERE nameLast LIKE '%riff%'
 LIMIT 5
")
```

```
               given     last birthYear
1      Arthur Joseph  Griffin      1988
2   Alfredo Claudino  Griffin      1957
3 Bartholomew Joseph Griffith      1896
4       Clark Calvin Griffith      1869
5     Robert Derrell Griffith      1943
```

Most SQL implementations also have a `REGEXP` or `REGEXLIKE` function that works with regular expressions, but `SQLite` requires a user defined `regex()` for its use so we skip it here.

Combining limiting clauses

Limiting `WHERE` clauses can be combined using `AND` and `OR`. Clauses can be negated using `NOT`.

```
SELECT nameGiven AS given, nameLast AS last,
       birthYear, birthCountry
  FROM People
 WHERE birthCountry == 'P.R.' AND birthYear LIKE '199%'
```

```
## Find all players born in Puerto Rico during the 1990's
dbGetQuery(lahman, "
SELECT nameGiven AS given, nameLast AS last,
       birthYear, birthCountry
  FROM People
 WHERE birthCountry == 'P.R.' AND birthYear LIKE '199%'
 LIMIT 5
")
```

```
          given     last birthYear birthCountry
1  Ednel Javier     Baez      1992         P.R.
2  Jose Orlando  Berrios      1994         P.R.
```

```
3 Victor Manuel Caratini      1993         P.R.
4  Willi Rafael   Castro      1997         P.R.
5     Alexander  Claudio      1992         P.R.
```

```
## Find all players from Hawaii or Alaska
dbGetQuery(lahman, "
SELECT nameGiven AS given, nameLast AS last,
       birthYear, birthState
  FROM People
 WHERE birthState == 'HI' OR birthState == 'AK'
 LIMIT 5
")
```

```
                              given     last birthYear birthState
1                       Benny Peter Agbayani      1971         HI
2 Dustin Kamakana Mai Ku'u Makualani  Antolin      1989         HI
3                       Anthony Lee Barnette      1983         AK
4                       Chad Robert    Bentz      1980         AK
5                     Douglas Edmund  Capilla      1952         HI
```

```
## Find all players from Hawaii or Alaska that aren't born in 1971
dbGetQuery(lahman, "
SELECT nameGiven AS given, nameLast AS last,
       birthYear, birthState
  FROM People
 WHERE (birthState == 'HI' OR birthState == 'AK') AND NOT birthyear == 1971
 LIMIT 5
")
```

```
                              given     last birthYear birthState
1 Dustin Kamakana Mai Ku'u Makualani  Antolin      1989         HI
2                       Anthony Lee Barnette      1983         AK
3                       Chad Robert    Bentz      1980         AK
4                     Douglas Edmund  Capilla      1952         HI
5                      Shawn Anthony   Chacon      1977         AK
```

ORDER BY

Use an `ORDER BY` clause with a comma separated list of columns to arrange the table.

```
SELECT playerID, yearID, RBI
  FROM Batting
 WHERE RBI >= 100 AND yearID >= 2010
 ORDER BY yearID, RBI
```

```
## Get all 100+ RBI seasons since 2010, ordered
rbi100 <- dbGetQuery(lahman, "
SELECT playerID, yearID, RBI
  FROM batting
 WHERE RBI >= 100 AND yearID >= 2010
 ORDER BY yearID, -RBI
")
head(rbi100)
```

```
   playerID yearID RBI
1 cabremi01   2010 126
2 rodrial01   2010 125
3 bautijo02   2010 124
4 pujolal01   2010 118
5 gonzaca01   2010 117
6 guerrvl01   2010 115
```

```
tail(rbi100)
```

```
     playerID yearID RBI
201 olsonma02   2022 103
202  croncj01   2022 102
203 machama01   2022 102
204 garciad02   2022 101
205 freemfr01   2022 100
206 turnetr01   2022 100
```

Aggregations, Group By

We can perform aggregations such as sums, means, and counts by using a `GROUP BY` clause.

Here we find the players with the most total RBI since 2010.

```
SELECT playerID, SUM(RBI) AS rbi_total
  FROM Batting
 WHERE yearID >= 2010
 GROUP BY playerID
 ORDER BY -rbi_total
```

```
## Count total RBIs since 2010 by player
dbGetQuery(lahman, "
SELECT playerID, SUM(RBI) AS rbi_total
  FROM Batting
 WHERE yearID >= 2010
 GROUP BY playerID
 ORDER BY -rbi_total
 LIMIT 10
")
```

```
    playerID rbi_total
1   cruzne02      1144
2  pujolal01      1106
3  cabremi01      1094
4  goldspa01      1042
5  freemfr01      1041
6  encared01       975
7  stantmi03       971
8  arenano01       968
9  mccutan01       948
10 longoev01       933
```

Having

The operator defining a limiting clause on an aggregate variable is `HAVING`. It is essentially like `WHERE` except for operating on summary statistics rather than individual rows. In other words, `HAVING` refers to the output table specified in `SELECT` rather than the input table(s) specified using `FROM`.

In the query below, observe that the `HAVING` clause comes after the `GROUP BY` but before the `ORDER BY`.

```
SELECT playerID, SUM(RBI) AS rbi_total
  FROM Batting
 WHERE yearID >= 2010
```

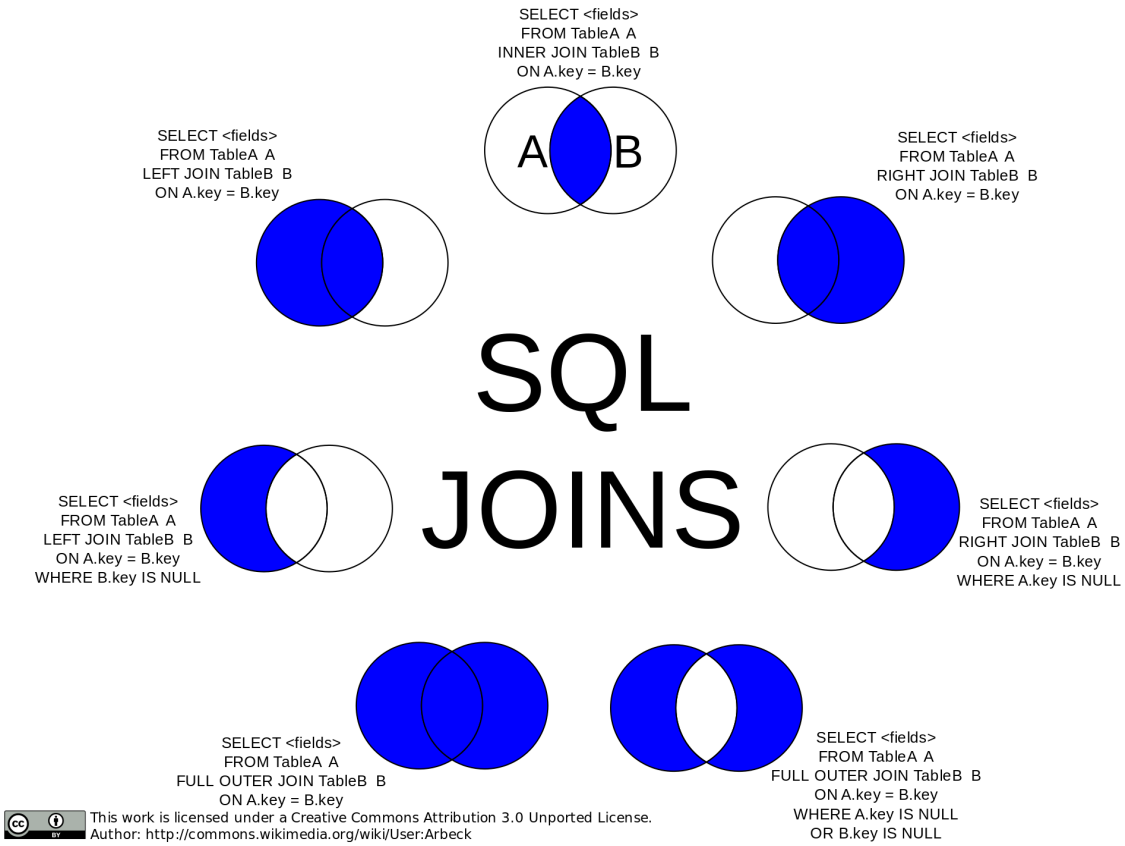
```
 GROUP BY playerID
HAVING rbi_total >= 1000
 ORDER BY -rbi_total
```

```
## Players with 1000+ RBIs since 2010
dbGetQuery(lahman, "
SELECT playerID, SUM(RBI) AS rbi_total
  FROM Batting
 WHERE yearID >= 2010
 GROUP BY playerID
HAVING rbi_total >= 1000
 ORDER BY -rbi_total
")
```

```
   playerID rbi_total
1  cruzne02      1144
2 pujolal01      1106
3 cabremi01      1094
4 goldspa01      1042
5 freemfr01      1041
```

Joins

So far we have discussed working with single tables only. The SQL term for merging data from two or more tables is a 'join'. All joins are based on the idea of equating rows that match on one or more variables. Here's a nice visualization of the different types of joins. In this image, the A table is the primary table (e.g. `FROM`), and the B table is the table you are joining in.



We'll demonstrate a few of these.

Inner Join

What if we wanted to supplement our earlier table showing players with 1000+ RBI since 2010 with information about those players? We could use an inner join of our RBI table with the "People" table to accomplish this.

`JOIN`s are clauses (similar to `FROM`, `GROUP BY`, etc) but typically considered as "part of" the `FROM`. So in indentation here, I choose to indicate that relationship, though others may not.

```
SELECT p.nameFirst AS first, p.nameLast AS last, p.birthState AS state,
       p.birthCountry AS country, SUM(b.RBI) AS rbi_total
  FROM Batting AS b
       INNER JOIN People AS p ON b.playerID = p.playerID
 WHERE b.yearID >= 2010
 GROUP BY b.playerID
HAVING rbi_total >= 1000
```

```
 ORDER BY -rbi_total
```

```
dbGetQuery(lahman, "
SELECT p.nameFirst AS first, p.nameLast AS last, p.birthState AS state,
       p.birthCountry AS country, SUM(b.RBI) AS rbi_total
  FROM Batting AS b
       INNER JOIN People AS p ON b.playerID = p.playerID
 WHERE b.yearID >= 2010
 GROUP BY b.playerID
HAVING rbi_total >= 1000
 ORDER BY -rbi_total
")
```

```
    first        last             state   country rbi_total
1  Nelson        Cruz      Monte Cristi      D.R.      1144
2  Albert      Pujols Distrito Nacional      D.R.      1106
3  Miguel     Cabrera            Aragua Venezuela      1094
4    Paul Goldschmidt                DE       USA      1042
5 Freddie     Freeman                CA       USA      1041
```

Note the renaming of the tables - `FROM batting AS b` and `INNER JOIN People AS p`. This allows us to preface variables names, e.g. `p.nameFirst`, to indicate which table to look for the variable. We could of course use `People.nameFirst` but it's shorter to rename.

The `ON` clause determines the connecting variables between the two tables.

Left & Right (Outer) Joins

In a left join – sometimes called a left outer join – we add columns from the right table to the left table when matching rows are found. Rows from the left table with no matches from the right table are retained with columns from the right table filled in as `NULL` (i.e. `NA`). When there are multiple matches of a row from the left table to rows in the right table, these each become a row in the new table.

A right join is equivalent to a left join with the exception that the roles between right and left are reversed.

Left joins are particularly useful when the information in the right table is only applicable to a subset of the rows from the left table. As an example, suppose we would like to know which US colleges and universities have produced the most "Rookie of the Year Awards" given to the best debuting player(s) each season.

To get started, we first test a query to find the last college attended.

```
-- Last college attended
SELECT *
  FROM CollegePlaying
 GROUP BY playerID
HAVING yearID == max(YearID)
```

```
# Query to find last college attended
dbGetQuery(lahman, "
SELECT *
  FROM CollegePlaying
 GROUP BY playerID
HAVING yearID == max(YearID)
 LIMIT 5
")
```

```
   playerID schoolID yearID
1 aardsda01     rice   2003
2  abadan01  gamiddl   1993
3 abbeybe01  vermont   1892
4 abbotje01 kentucky   1994
5 abbotji01 michigan   1988
```

Now, we find all distinct awards in the `AwardPlayers` table.

```
# Distinct Player Awards
dbGetQuery(lahman, "
SELECT DISTINCT(awardID)
  FROM AwardsPlayers
 LIMIT 5
")
```

```
                    awardID
1 Baseball Magazine All-Star
2               Triple Crown
3      Pitching Triple Crown
4       Most Valuable Player
5               TSN All-Star
```

Next we test a query for finding all Rookie of the Year Awards.

```
SELECT *
  FROM AwardsPlayers
 WHERE awardID LIKE 'Rookie%'
```

```
# Query to find Rookie of the Year Awards
dbGetQuery(lahman, "
SELECT *
  FROM AwardsPlayers
 WHERE awardID LIKE 'Rookie%'
 LIMIT 5
")
```

```
   playerID            awardID yearID lgID  tie notes
1 robinja02 Rookie of the Year   1947   ML <NA>  <NA>
2  darkal01 Rookie of the Year   1948   ML <NA>  <NA>
3 sievero01 Rookie of the Year   1949   AL <NA>  <NA>
4 newcodo01 Rookie of the Year   1949   NL <NA>  <NA>
5 dropowa01 Rookie of the Year   1950   AL <NA>  <NA>
```

Finally, we use a *left join* of the tables for Rookie of the Year awards and last college attended to match winners to their schools. We need a left join as many of the winners may never have played collegiate baseball, and we want to keep them, but we don't want to keep colleges which never produced any winners.

```
SELECT roy.playerID AS playerID, roy.yearID AS year, lgID AS league, schoolID
  FROM AwardsPlayers AS roy
       LEFT JOIN
       (SELECT *  --Final College Attended
          FROM CollegePlaying
         GROUP BY playerID
        HAVING yearID == MAX(YearID)
       ) AS c ON c.playerID = roy.playerID
 WHERE awardID LIKE 'Rookie%'
```

Note the nested structure here - Inside the `LEFT JOIN`, we write a separate `SELECT` statement. We could have done this in two steps: generate a new table (using `CREATE TABLE finalcollege AS` followed by the `SELECT` statement) and then directly use it in the clause, `LEFT JOIN finalcollege AS c`.

```
# Query to find last college for ROY
dbGetQuery(lahman, "
```

```
SELECT roy.playerID AS playerID, roy.yearID AS year, lgID AS league, schoolID
  FROM AwardsPlayers AS roy
       LEFT JOIN
       (SELECT *  --Final College Attended
          FROM CollegePlaying
         GROUP BY playerID
        HAVING yearID == MAX(YearID)
       ) AS c ON c.playerID = roy.playerID
 WHERE awardID LIKE 'Rookie%'
 LIMIT 5
")
```

```
   playerID year league schoolID
1 robinja02 1947     ML     ucla
2  darkal01 1948     ML    ulala
3 sievero01 1949     AL     <NA>
4 newcodo01 1949     NL     <NA>
5 dropowa01 1950     AL    uconn
```

To complete the example, we modify the query to display which schools have produced the most ROY awards in total.

```
SELECT schoolID, COUNT(c.playerID) AS ROY_awards
  FROM AwardsPlayers roy
       LEFT JOIN
       (SELECT *  --Last College Attended
          FROM CollegePlaying
         GROUP BY playerID
        HAVING yearID == MAX(YearID)
       ) c ON c.playerID = roy.playerID
 WHERE awardID LIKE 'Rookie%'
       AND schoolID IS NOT NULL
 GROUP BY schoolID
HAVING ROY_awards > 1
 ORDER BY -ROY_awards
```

```
# Which schools have produced the most ROY?
dbGetQuery(lahman, "
SELECT schoolID, COUNT(c.playerID) AS ROY_awards
  FROM AwardsPlayers roy
       LEFT JOIN
```

```
        (SELECT *  --Last College Attended
           FROM CollegePlaying
          GROUP BY playerID
         HAVING yearID == MAX(YearID)
        ) c ON c.playerID = roy.playerID
  WHERE awardID LIKE 'Rookie%'
        AND schoolID IS NOT NULL
  GROUP BY schoolID
 HAVING ROY_awards > 1
  ORDER BY -ROY_awards
 ")
```

```
   schoolID ROY_awards
1 arizonast          4
2  michigan          3
3      ucla          3
4       usc          3
5 floridaam          2
6 longbeach          2
7  oklahoma          2
8   texasam          2
9    wagner          2
```

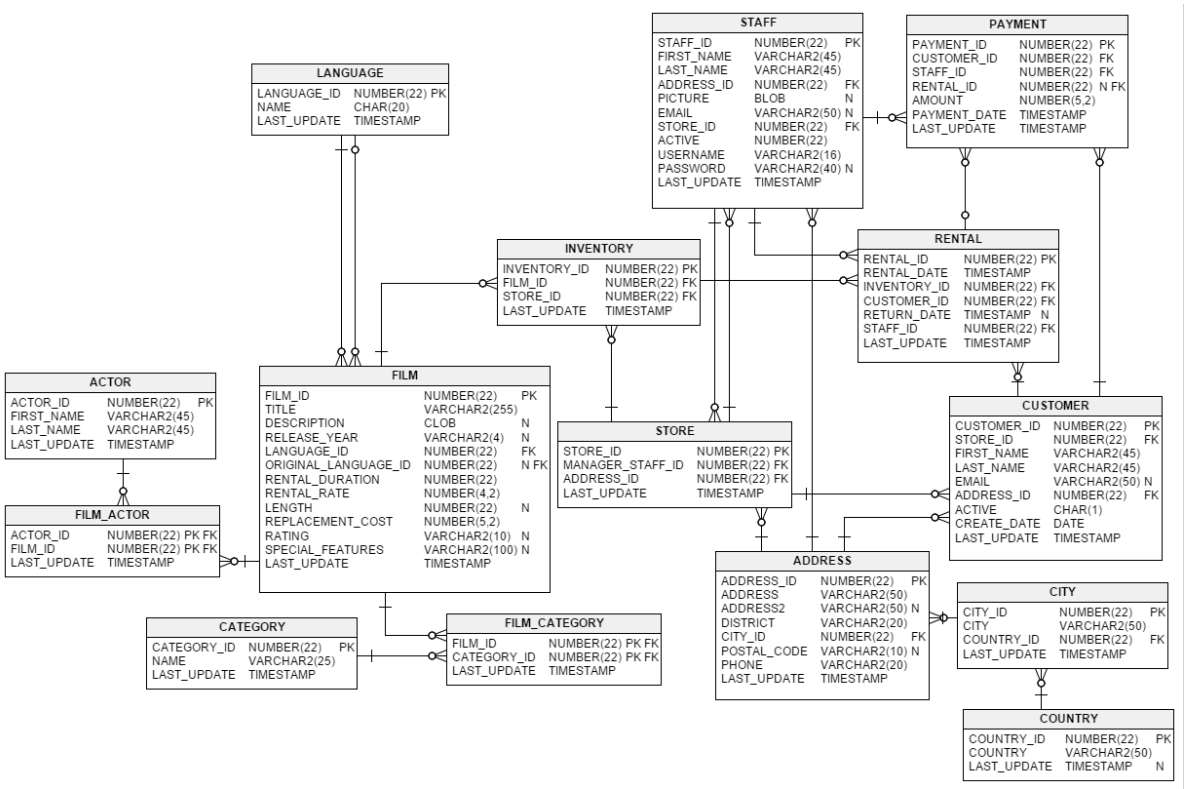
Order of clauses

The previous example demonstrated the order of almost all clauses:

```
SELECT
FROM
JOIN
WHERE
GROUP BY
HAVING
ORDER BY
LIMIT
```

Another Example

The "sakila" database is a fake data set created by the MySQL team which simulates a very rich database of many tables. A map of its contents is:



It simulates the database of a Blockbuster-style rental store. It includes customer data, movie data, and rental data linking the two. An SQLite database containing the data can be downloaded from https://github.com/bradleygrant/sakila-sqlite3.

```
sakila <- dbConnect(RSQLite::SQLite(), "~/Downloads/sakila_master.db")
dbListTables(sakila)
```

```
 [1] "actor"                "address"              "category"
 [4] "city"                 "country"              "customer"
 [7] "customer_list"        "film"                 "film_actor"
[10] "film_category"        "film_list"            "film_text"
[13] "inventory"            "language"             "payment"
[16] "rental"               "sales_by_film_category" "sales_by_store"
[19] "staff"                "staff_list"           "store"
```

Let's use this data to examine which actor or actress it the most "rented". Looking at the tables, we have the "rental" table which contains records of every movie rented. From here, we are connected to the "inventory" table, which records which movie was actually rented. So first, let's obtain a list of every movie ever rented.

```
SELECT i.film_id
  FROM rental AS r
       LEFT JOIN inventory AS i ON i.inventory_id = r.inventory_id
```

We use a left join here because there may be movies in the inventory that were never rented.

```
dbGetQuery(sakila, "
SELECT i.film_id
  FROM rental AS r
       LEFT JOIN inventory AS i ON i.inventory_id = r.inventory_id
 LIMIT 5
")
```

```
  film_id
1       1
2       1
3       1
4       1
5       1
```

Now, we could join next to the "films" table, however, you may notice that both the "films" and "film_actor" have a `film_id` key, so we can completely bypass "films".

```
SELECT fa.actor_id
  FROM film_actor AS fa
       RIGHT JOIN
       (SELECT i.film_id
          FROM rental AS r
               LEFT JOIN inventory AS i ON i.inventory_id = r.inventory_id
       ) AS rr ON fa.film_id = rr.film_id
```

This time we use a right join as we don't want to list any actors which were in movies that weren't rented.

```
dbGetQuery(sakila, "
SELECT fa.actor_id
  FROM film_actor AS fa
       RIGHT JOIN
       (SELECT i.film_id
          FROM rental AS r
          LEFT JOIN inventory AS i ON i.inventory_id = r.inventory_id
```

```
      ) AS rr ON fa.film_id = rr.film_id
 LIMIT 5
")
```

```
  actor_id
1        1
2        1
3        1
4        1
5        1
```

We now have a list of actor ID's, next we just need to connect it to the actor names.

```
SELECT COUNT(a.actor_id) AS count, a.first_name, a.last_name
  FROM actor AS a
       RIGHT JOIN
       (SELECT fa.actor_id
          FROM film_actor AS fa
               RIGHT JOIN
               (SELECT i.film_id
                  FROM rental AS r
                       LEFT JOIN inventory AS i
                       ON i.inventory_id = r.inventory_id
               ) AS rr ON fa.film_id = rr.film_id
       ) AS ff ON ff.actor_id = a.actor_id
 GROUP BY a.actor_id
 ORDER by -count
```

```
dbGetQuery(sakila, "
SELECT COUNT(a.actor_id) AS count, a.first_name, a.last_name
  FROM actor AS a
       RIGHT JOIN
       (SELECT fa.actor_id
          FROM film_actor AS fa
               RIGHT JOIN
               (SELECT i.film_id
                  FROM rental AS r
                       LEFT JOIN inventory AS i
                       ON i.inventory_id = r.inventory_id
               ) AS rr ON fa.film_id = rr.film_id
       ) AS ff ON ff.actor_id = a.actor_id
```

```
  GROUP BY a.actor_id
  ORDER by -count
  LIMIT 5
 ")
```

```
  count first_name   last_name
1   753       GINA   DEGENERES
2   678    MATTHEW      CARREY
3   674       MARY      KEITEL
4   654     ANGELA WITHERSPOON
5   640     WALTER        TORN
```