Databases

Yue Xiong, LMU, yue.xiong@stat.uni-muenchen.de

Chapter 01 Databases

Basic Introduction to SQL and Relational Databases

SQL is a language designed for a very specific purpose: to interact with relational databases.

- **Database**: A database is a structured collection of data. There are various different ways of structuring the database, and there may or may not be information about the relationship between entities in the database.
- **Query**: A query is a request for data from the database.
- **Database Management System (DBMS)**: A DBMS is a system of storing and managing databases, including querying the database.
- **Relational Database Management System (RDBMS)**: In an RDBMS, data records are stored in *tables*, each of which has a predefined set of *columns*, the pieces of information captured for each record in a table, and *rows* in the table, where each row has a place to store a value for every column in the table.

Tables, including their columns, column types and relationships with other tables, are defined in a database **schema**. Many times, tables will contain a **primary key**, one or more columns that uniquely define a row. You can think of the primary key as a kind of ID, in which each row is given a unique ID. Tables can also contain **foreign keys**, which are column(s) that comprise the primary key in another table and, thus, provides a way of matching between multiple tables.

In this notebook, we will use SQL to: - Select data subsets - Sum over groups - Create new tables - Count distinct values of desired variables - Order data by chosen variables - Join tables together

Step 1: Estabilish a Connection to the Database using `R`

In order to get access to the corresponding SQL database, we need to firstly install the required R package developed for accessing databases: `RSQlite`. Furthermore, by installing the `dbplyr` package, it helps provide an interface focusing on retrieving and analyzing datasets by generating `SELECT` SQL statements.

```
# install.packages(c("dbplyr", "RSQLite", "DBI"))  # install the required packages
library('dbplyr')  # load the corresponding libraries
```

```
## Warning: package 'dbplyr' was built under R version 4.0.5
```

```
library('dplyr')
```

```
## Warning: package 'dplyr' was built under R version 4.0.5
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:dbplyr':
##
##     ident, sql

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library('RSQLite')
```

```
## Warning: package 'RSQLite' was built under R version 4.0.5
```

```r
library('DBI')
```

```
## Warning: package 'DBI' was built under R version 4.0.5
```

```r
# connect to the corresponding database
database_path = "F:/hiwi_work_notebook/bdss-notebooks/ncdoc.db"
ncdoc <- DBI::dbConnect(RSQLite::SQLite(), database_path)
```

```r
# src_dbi(ncdoc)  # checking tables in this database
```

Formulate Data Query To connect to tables within a database, we can insert the sql chunk and specify the connection. Let's demonstrate this functionality by looking at the **inmates** data.

```sql
SELECT *
FROM inmate
LIMIT 20
```

Table 1: Displaying records 1 - 10

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	MALE	WHITE 1961-10-15	[illegible] MED-01	[illegible] POPU-LATION RPOP	[illegible] 01-01	REGULAR 01-01	1983-07-13	1984-07-11	[illegible] PAROLE	[illegible]	UNKNOWN AT CONVERSION	0001-01-01	0.00 FELON	0000000000

INMATE												
0000ACKGERALD	MALWHITE 1951-07-17	UNKNOWN 1 MN	01-01 POPULATION RPOP	REGULAR 01-01	1973-04-15 CI DIAG CENTER	1975-08-18 PACC ROCI CLAHERN	UNKNOWN AT CONVERSION	0001-01-01	00 MISD.	0000000000	0001-01-01 01 01	
0000ADAMS JAMES	MALWHITE 1963-12-29	UNKNOWN 1 MN	01-01 POPULATION RPOP	REGULAR 01-01	1995-01-13 CI DIAG CENTER	1995-09-14 PACC ROCI GOLD	ESCAPED/UNKNOWN FROM INM. MASTER	1994-12-13	HABITUAL IMPAIRED DRIVING	0000000000	0001-01-09 01 14	
0000AIKKENNETH	MALBLACK 1953-05-18	UNKNOWN MEDIUM	01-01 POPULATION RPOP	REGULAR 01-01	1977-03-17 CENTRAL PRISON D. C.	1984-04-27 ROCI CE HANOVER NHAN	UNKNOWN AT CONVERSION	0001-01-01	00 FELON	0000000000	0001-01-01 01 01	
0000ALROY ROYNER	MALWHITE 1951-08-26	UNKNOWN 1 MN	01-01 POPULATION RPOP	REGULAR 01-01	1977-06-21 PIEDMONT AREA DC	1978-01-23 PACC ROCI DAVI	UNKNOWN AT CONVERSION	0001-01-01	00 MISD.	0000000000	0001-01-01 01 01	
0000ASTONSTEVEN	MALWHITE 1963-05-16	UNKNOWN 1 MN	01-01 POPULATION RPOP	REGULAR 01-01	1992-08-26	1992-09-08 PACI ROCI TERMINATED TLEI	UNKNOWN AT CONVERSION	0001-01-01	00 MISD.	0000000000	0001-01-01 01 01	
0000AIDKINSONJOHN	FEMWHITE 1955-06-15	UNKNOWN MEDIUM	01-01 POPULATION RPOP	REGULAR 01-01	1988-07-15 INST. WOMEN DC	1990-06-22 PARFW ROCI WOMEN NCRFW	UNKNOWN AT CONVERSION	0001-01-01	00 FELON	0000000000	0001-01-01 01 01	
0000ALBELIDIAS	MALOTHER 1955-05-31	UNKNOWN 1 MN	01-01 POPULATION RPOP	REGULAR 01-01	1977-07-19 CI DIAG CENTER	1978-02-28 PACC ROCI RICHMOND RICH	UNKNOWN AT CONVERSION	0001-01-01	00 MISD.	0000000000	0001-01-01 01 01	
0000ALBEJEFFERY	MALWHITE 1949-11-17	UNKNOWN MEDIUM	01-01 POPULATION RPOP	REGULAR 01-01	1983-07-21 CI DIAG CTR	1984-09-09 PACC ROCI ROCKINGHAM ROCK	UNKNOWN AT CONVERSION	0001-01-01	00 FELON	0000000000	0001-01-01 01 01	

[illegible]
0000... MICHAEL... 02-20 ... 1 MN ... 05-09 POPULATION RPOP ... 08-04 ... 10-13 CORR INSTIT ... HANOVER CC NHAN ... HANOVER COUNTY ... UNCAPPED COMPUTED ... 10-22 ... 000 ... 000 ... BB ... 01-14 ... 01-13 ... [illegible]

You should see 20 rows of the `inmate` dataset. Let's go over the basics of this SQL command.

- **SELECT:** We start out with the `SELECT` statement. The `SELECT` statement specifies which variables (columns) you want.
  - Here, we used `SELECT *`. The "`*`" just says that we want all the variables.
  - If we wanted a few columns, we would use the column names separated by commas instead of "`*`".
- **FROM:** Now, let's look at the next part of the query, `FROM awards`. This part of the query specifies the table, `awards`, from which we want to retrieve the data. Most of your queries will begin in this fashion, describing which columns you want and from which table.
- **LIMIT:** We typically include a `LIMIT` statement at the end of our query so that we don't get overloaded with rows being output. Here, `LIMIT 20` means that we just want the first ten rows. Many times, the `LIMIT` that you want will be higher than 20 – you might generally prefer to use 1000 or so. Having a `LIMIT` for all queries is highly recommended even if you know only a few rows will be shown, since it acts as a safety precaution against (for example) displaying millions of rows of data.

In this case, we've put everything in one line, but that's not necessary. We could have split the code up into multiple lines, like so:

```
SELECT *
FROM inmate
LIMIT 20;
```

This gives the same output as our original query. Generally, once queries start getting longer, breaking up the code into multiple lines can be very helpful in organizing your code and making it easier to read.

Along those lines, note that we used a semi-colon at the end of the query to mark the end of the query. That isn't absolutely necessary here, but it does help mark the end of a query and is required in other applications of SQL, so it's good practice to use it.

Side note about capitalization

If you notice, we've been using all caps for SQL commands and all lowercase for data table and schema names. This is simply a convention, as SQL is not case sensitive. For example, we could have run `select * from inmate limit 20` and it would have given us the exact same output as the first query.

This does mean you need to be careful when using column names. If your column name has capital letters in it, you need use double quotes (e.g. `"INMATE_DOC_NUMBER"`) to preserve the capitalization. For this reason, you might find that using all lowercase letters in column names is preferable, which is what we've done here.

Note that the `LIMIT` provides one simple way to get a "sample" of data; however, using `LIMIT` does **not provide a *random*** sample. You may get different samples of data than others using just the `LIMIT` clause, but it is just based on what is fastest for the database to return.

Basic Summaries of the Data One of the basic things you might be interested in doing is finding out how many rows there are in the dataset. You can do this using the `COUNT` statement.

Let's find the total number of employees that we have in our `sentences` table.

```
SELECT COUNT(*)
FROM sentences
LIMIT 20;
```

Table 2: 1 records

COUNT(*)
1721015

We can also count the number of non-NULL values there are in a given variable by including a column name instead of `*`.

```
SELECT COUNT(inmate_doc_number)
FROM sentences
LIMIT 20;
```

Table 3: 1 records

COUNT(inmate_doc_number)
1721015

To count the number of unique cases, you can use the `DISTINCT` statement. This checks how many unique values of that variable there are.

Let's find the total number of unique individuals who were in those sentences.

```
SELECT COUNT(DISTINCT inmate_doc_number)
FROM sentences
LIMIT 20;
```

Table 4: 1 records

COUNT(DISTINCT inmate_doc_number)
465123

You can also do other basic summaries, such as finding sum using `SUM` or average using `AVG`.

Conditional Statements Suppose we want to look at a subset of the data. We can use conditional statements to do this.

```
SELECT *
FROM inmate
WHERE inmate_gender_code = 'MALE'
LIMIT 10;
```

Table 5: Displaying records 1 - 10

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0000ADKOND	MAWHITE 1961 10- 15	MED- 01	NONE POP- U- LA- TION RPOP 01- 01	REGULAR 01- 01	1983- 07- 13	1984 07- 11 PACC ROLE ALESO	UNKNOWN AT CON- VER- SION	0001- 01- 01	00 FELON	000000000	0001-01-01 01-01-01 01 01
0000AGERALD	MAWHITE 1951 07- 17	1 MN 01	NONE POP- U- LA- TION RPOP 01- 01	REGULAR 01- 01	1973- 04- 15 CI DIAG CEN- TER	1975 08- 18 PACI ROLE HARN	UNKNOWN AT CON- VER- SION	0001- 01- 01	00 MISD.	000000000	0001-01-01 01-01-01 01 01
0000AJAMES	MAWHITE 1963 12- 29	1 MN 01	NONE POP- U- LA- TION RPOP 01- 01	REGULAR 01- 01	1995- 01- 13 CI DIAG CEN- TER	1995 09- 14 PACC ROLE GOLD	GOLDSBORO UN- KNOWN CAME FROM INM. MAS- TER	1994- 12- 13	HABITUAL IM- PAIRED DRIV- ING	000000000	0001-09-01-09-01 14
0000AKENNETH	MALBLACK 1953 05- 18	MED- 01	NONE POP- U- LA- TION RPOP 01- 01	REGULAR 01- 01	1977- 03- 17 PRISON D. C.	1983 06- 27 ROLE NHAN	HANOVER UNKNOWN AT CON- VER- SION	0001- 01- 01	00 FELON	000000000	0001-01-01 01-01-01 01 01
0000AROYNER	MAWHITE 1952 08- 26	1 MN 01	NONE POP- U- LA- TION RPOP 01- 01	REGULAR 01- 01	1977- 06- 21 PIED- MONT AREA DC	1978 12- 23 PACC ROLE DAVI	UNKNOWN AT CON- VER- SION	0001- 01- 01	00 MISD.	000000000	0001-01-01 01-01-01 01 01
0000ASTEVEN	MAWHITE 1963 05- 16	1 MN 01	NONE POP- U- LA- TION RPOP 01- 01	REGULAR 01- 01	1992- 08- 26	1992 09- 08 PACI ROLE ILLI	UNKNOWN AT CON- VER- SION	0001- 01- 01	00 MISD.	000000000	0001-01-01 01-01-01 01 01
0000ABDIAS	MAOTHER 1955 05- 31	1 MN 01	NONE POP- U- LA- TION RPOP 01- 01	REGULAR 01- 01	1977- 07- 19 CI DIAG CEN- TER	1978 02- 28 PACC ROLE RICH	UNKNOWN AT CON- VER- SION	0001- 01- 01	00 MISD.	000000000	0001-01-01 01-01-01 01 01
0000ABJEFFERY	MAWHITE 1949 11- 17	MED- 01	NONE POP- U- LA- TION RPOP 01- 01	REGULAR 01- 01	1983- 07- 21 CI DIAG CTR	1984 09- 09 PACC ROLE ROCK	ROCKINGHAM UNKNOWN AT CON- VER- SION	0001- 01- 01	00 FELON	000000000	0001-01-01 01-01-01 01 01

[illegible]												
[illegible]	02-20	1 MN	05-09 POP-U-LATION RPOP	01-01	08-04	10-13	COR-R IN-STIT	HANOVER CC NHAN	HANOVER COUNTY UNAPPED COMPUTED	10-22	000 BB 001	01-14 01-13 [illegible]
[illegible]	01-08	MED-01	POP-U-LATION RPOP	01-01	01-01	09-19	PRISON D.C.	02 ROW WAKE CC	UNKNOWN AT CONVERSION	0001-01-01	FELON 0000 00000	0001-01-01 01-01 [illegible]

The WHERE statement is used to return only data that meets certain conditions. Here, we used it to find the information for employee whose id is 90014540.

Common Comparison Operators Though there are some more complicated comparison operators (if you're curious, feel free to look up what `LIKE` and `IN` do), these should cover most of what you want to do. - `=`: equal to - `!=` or "`<>`": not equal to - `<`: less than - `<=`: less-than-or-equal-to - `>`: greater than - `>=`: greater-than-or-equal-to - `IS NULL` and `IS NOT NULL`: The signifier of a row in a column not having a value is a special keyword: `NULL`. To check for `NULL`, you use `IS NULL` or `IS NOT NULL`, rather than "=" or "!=". For example, to count the number of rows with `NULL` values for `employee` we might use the following:

```
SELECT *
FROM awards
WHERE employee IS NOT NULL
LIMIT 10;
```

Using Aggregation Functions What if we wanted to get summaries of the data, such as counts, aggregated by a categorical variable? We can do this using the `GROUP BY` statement.

- Using `GROUP BY` with single variable

```sql
SELECT inmate_race_code, count(*)
FROM inmate
GROUP BY inmate_race_code
LIMIT 20;
```

Table 6: 7 records

INMATE_RACE_CODE	count(*)
	1
ASIAN/ORTL	905
BLACK	229101
INDIAN	9269
OTHER	16904
UNKNOWN	1559
WHITE	203682

Here, the `GROUP BY` statement groups it into the categories of the variable. Since we've chosen to display the count, we can see the counts. We can also change the order in which the results are displayed so that it's in increasing order.

```
SELECT inmate_race_code, count(*)
FROM inmate
GROUP BY inmate_race_code
ORDER BY count(*)
LIMIT 20;
```

Table 7: 7 records

INMATE_RACE_CODE	count(*)
	1
ASIAN/ORTL	905
UNKNOWN	1559
INDIAN	9269
OTHER	16904
WHITE	203682
BLACK	229101

The `ORDER BY` statement orders the rows that it displays according to whatever you put after it. In this case, we chose the count.

- Using `GROUP BY` with Multiple Variables

Suppose we wanted to look at counts by race and by gender. We could do this by adding the gender variable to the `GROUP BY` statement.

```
SELECT inmate_race_code, inmate_gender_code, count(*)
FROM inmate
GROUP BY inmate_race_code, inmate_gender_code
ORDER BY count(*) DESC
LIMIT 20;
```

Table 8: Displaying records 1 - 10

INMATE_RACE_CODE	INMATE_GENDER_CODE	count(*)
BLACK	MALE	204271
WHITE	MALE	172562
WHITE	FEMALE	31120
BLACK	FEMALE	24830
OTHER	MALE	16184
INDIAN	MALE	7972
UNKNOWN	MALE	1501
INDIAN	FEMALE	1297
ASIAN/ORTL	MALE	825
OTHER	FEMALE	720

This first groups by `inmate_race_code`, then it groups by `inmate_gender_code`, in that order. Further, notice that we used `DESC` after `ORDER BY`. This orders in descending order instead of ascending order, so that we can see the groups with the most people at the top.

Conditional Statements After Aggregation Suppose we wanted to display only certain counts. We can use `HAVING` to do this.

```
SELECT inmate_race_code, inmate_gender_code, count(*)
FROM inmate
GROUP BY inmate_race_code, inmate_gender_code
HAVING count(*) > 1000
ORDER BY count(*) DESC
LIMIT 20;
```

Table 9: 8 records

INMATE_RACE_CODE	INMATE_GENDER_CODE	count(*)
BLACK	MALE	204271
WHITE	MALE	172562
WHITE	FEMALE	31120
BLACK	FEMALE	24830
OTHER	MALE	16184
INDIAN	MALE	7972
UNKNOWN	MALE	1501
INDIAN	FEMALE	1297

This will only display the counts for which the count is greater than 1000. Note that this is different from using `WHERE`, since the conditional statement comes after the `GROUP BY` statement. Basically, `HAVING` gives us a way of using the same types of conditional statements after we do our aggregation.

Joins One of the nice things about relational databases is organization using multiple tables that are linked together in some way. For example, suppose we have one table with 6 rows called **Table A**:

id	var1
1	5
2	10
3	2
4	6
5	22
6	9

And another table with 5 rows called **Table B**:

is	var2
2	2
5	4
6	1
7	2
8	0

Let's say we want to combine Table A and Table B so that we have one table that contains information about `id`, `var1`, and `var2`. We want to do this by matching the two tables by what they have in common, `id`. That is, we want a table that looks like this (let's call this **Table C**):

id	var1	var2
2	10	2
5	22	4
6	9	1

Table C has each `id` that was in both Table A and Table B. It also contains the appropriate values for `var1` and `var2` corresponding to each `id`. This kind of matching can be quite tricky to figure out manually, since there are different numbers of rows in each table, not all of the `id` values match for the two tables, and there are some `id` values that aren't in both. Fortunately for us, SQL is well-equipped to handle this task using the `JOIN` statement.

```sql
SELECT sentences.inmate_doc_number, inmate.inmate_race_code, sentences.actual_sentence_end_date
FROM sentences
JOIN inmate
ON sentences.inmate_doc_number = inmate.inmate_doc_number
LIMIT 1000
```

Table 13: Displaying records 1 - 10

INMATE_DOC_NUMBER	INMATE_RACE_CODE	ACTUAL_SENTENCE_END_DATE
0000004	WHITE	1984-07-11
0000004	WHITE	1984-07-11
0000006	WHITE	1973-03-28
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18

Now that we're connected and have established a plan for how we're joining two tables together, let's take a look at the SQL code that performs this join and break it down.

```
SELECT * FROM sentences
JOIN inmate
ON sentences.inmate_doc_number = inmate.inmate_doc_number
LIMIT 1000
```

Here, we want to `SELECT` each column from a data table that we get from joining the tables `inmate` and `sentences`. The second line takes the `inmate` table and joins the `sentences` table to it.

We can't just mash two tables together though – we need some way of making sure that the appropriate rows match. We do this with the third line:

```
ON sentences.inmate_doc_number = inmate.inmate_doc_number
```

This part specifies what we're joining on. That is, what is the ID variable that is in both tables that we want to match. They don't need to be named the same in both tables, though you do need to specify what they are in each table, even if they are the same, as well as which table they are from.

If you run the full code below, you should see the first 1000 rows (because of the `LIMIT 1000`) of the joined table. You should be able to scroll through all of the variables and see that we've managed to merge the `inmate` and `sentences` tables together according to their IDs.

> **Side note:** We're only going to be displaying a few of the columns instead of using `SELECT *` like we showed above. This is because we aren't able to display more than 50 columns here in this notebook format. Joining to get tables with greater than 50 columns is perfectly fine, but we'll only look at a few at a time to make it easier to follow in these exercises.

```
SELECT sentences.inmate_doc_number, inmate.inmate_race_code, sentences.actual_sentence_end_date
FROM sentences
JOIN inmate
ON sentences.inmate_doc_number = inmate.inmate_doc_number
LIMIT 1000
```

Table 14: Displaying records 1 - 10

INMATE_DOC_NUMBER	INMATE_RACE_CODE	ACTUAL_SENTENCE_END_DATE
0000004	WHITE	1984-07-11
0000004	WHITE	1984-07-11
0000006	WHITE	1973-03-28
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18

Different Types of Joins We've so far done only one type of join, an inner join. This is the default join (which is why we didn't need to specify anything more in the code). However, there are different types of joins.

- Left and Right Joins in SQL

Suppose we want to look at every single census block in one table, only filling in information from the second table if it exists. We'll illustrate this using Table A and Table B from before. Recall that our `JOIN` created Table C:

id	var1	var2
2	10	2
5	22	4
6	9	1

Instead, we want to create the following table:

id	var1	var2
1	5	*null*

id	var1	var2
2	10	2
3	2	*null*
4	6	*null*
5	22	4
6	9	1

Here, we've kept every single row in Table A, and simply filled in the information from Table B if it existed for that `id`. This is called a **LEFT JOIN**, since we're taking the table on the left (that is, Table A) and adding the information from Table B onto that. We could have also done a **RIGHT JOIN**, which does the same thing, except flipping the tables, giving us something that looks like:

id	var1	var2
2	10	2
5	22	4
6	9	1
7	*null*	2
8	*null*	0

```
SELECT sentences.inmate_doc_number, inmate.inmate_race_code, sentences.actual_sentence_end_date
FROM sentences
LEFT JOIN inmate
ON sentences.inmate_doc_number = inmate.inmate_doc_number
LIMIT 1000
```

Table 18: Displaying records 1 - 10

INMATE_DOC_NUMBER	INMATE_RACE_CODE	ACTUAL_SENTENCE_END_DATE
0000004	WHITE	1984-07-11
0000004	WHITE	1984-07-11
0000006	WHITE	1973-03-28
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18
0000006	WHITE	1975-08-18

- Outer Join

An outer join keeps all unique ids, then puts `NULL` if it isn't part of that table. This is similar to a `LEFT` or `RIGHT JOIN`, except instead of only keeping all IDs from one table, it keeps them from both tables. Consider our example with Table A and Table B. We want to join them such that we get a table that looks like:

id	var1	var2
1	5	*null*
2	10	2

id	var1	var2
3	2	*null*
4	6	*null*
5	22	4
6	9	1
7	*null*	2
8	*null*	0

In a way, it's like combining the `LEFT` and `RIGHT JOIN`s so that we have all information from both tables.

Notice that we aren't able to show the outer join here, as it isn't supported by SQLite. We've provided the code here, but it won't be executed, so just make sure to keep it in mind for the future.

```
SELECT sentences.inmate_doc_number, inmate.inmate_race_code, sentences.actual_sentence_end_date
FROM sentences
FULL OUTER JOIN inmate
ON sentences.inmate_doc_number = inmate.inmate_doc_number
LIMIT 1000
```

Creating New Tables for Future Use

- Creating Tables

So far, we've mostly just been exploring the data without making any changes to the database. However, there might be times when we might want to create new tables. We can do this using `CREATE TABLE`. Let's use a previous example to create a new table.

```
CREATE TABLE joinedtable AS
SELECT *
FROM sentences
JOIN inmate
ON sentences.inmate_doc_number = inmate.inmate_doc_number
```

This should look mostly familiar, since everything after the first line is stuff we've already done. The first line creates a new table called `joinedtable` from the output.

This is a bit of a mess, though. We usually don't need everything from the tables that we do join, so we can choose what we keep. Let's create a new table that has just the information we need.

```
CREATE TABLE joinedtable2 AS
SELECT sentences.inmate_doc_number, inmate.inmate_race_code, sentences.actual_sentence_end_date
FROM sentences
JOIN inmate
ON sentences.inmate_doc_number = inmate.inmate_doc_number
```

Firstly, notice that we use aliasing to help make referring to tables easier. That is, in the third and fourth lines, we put "`a`" and "`b`" after each table to give it that alias. We can then use "`a`" and "`b`" whenever we refer to either table, which makes the `SELECT` statement easier.

Along those lines, notice that we specify which table each variable was from. If the column name is unique between the two tables (i.e. both tables don't have a column with the same name), then you don't need to specify the table as we've done. However, if they aren't unique and both tables have a variable with that name, you need to specify which one you want.

Finally, we've made the table easier to read by changing the name of the variable in the new table, using `AS` in the `SELECT` part of the query.

- Dropping Tables

Conversely, you can also drop, or delete, tables. We created a table in the previous section that we won't need, so let's drop it.

```sql
DROP TABLE joinedtable;
```

```sql
DROP TABLE joinedtable2;
```

You might be tempted to avoid dropping tables since it seems relatively harmless to simply not use the table anymore without dropping them. However, it is important to keep databases clean and consider the amount of space each table takes up.