Introduction to NIBRS and SQL

Greg Ridgeway

2026-07-23

Table of contents

1 National Incident-Based Reporting System

The FBI established the Uniform Crime Reporting (UCR) Program in 1930 to generate reliable crime statistics for law enforcement administration, operation, and management. It has been the primary source of crime data in the United States for decades.

The UCR's Summary Reporting System (SRS) is the best-known component of the UCR. About 18,000 law enforcement agencies, including municipal police departments, sheriff's departments, campus police, transit police, park police, and many other agencies, reported monthly counts of Part 1 crimes (murder and non-negligent manslaughter, forcible rape, robbery, aggravated assault, burglary, larceny theft, motor vehicle theft, and arson), often called "index crimes." The SRS also tracked monthly counts of Part II crimes, such as simple assault, fraud, vandalism, drug offenses, and driving under the influence. Even though reporting to the SRS was voluntary, almost all law enforcement agencies reported their data, offering fairly comprehensive coverage of crimes reported to the police in the United States.

While the UCR SRS has provided valuable crime data for many years, it has several notable limitations.

1. Limited number of crimes. The SRS focused on a limited number of crimes, potentially overlooking important details and emerging trends. With SRS data researchers cannot separate out trends in shootings, identity theft, and cybercrimes. Shootings, for example, are lumped together in an aggravated assault category that includes a range of serious assaults like striking someone with a beer bottle.
2. Only most serious crime reported. The SRS also operated under the "Hierarchy Rule," reporting only the most severe offense in a multi-offense incident.
3. No crime details. The SRS only collected aggregate counts, meaning that detailed information about the context of the crimes, such as the characteristics of the victims and offenders, the extent of property loss or damage, the time, place, and context of the crime, and the relationships between these crime features, is lost.

The FBI introduced the National Incident-Based Reporting System (NIBRS) in the 1980s. NIBRS aimed to address the shortcomings of the SRS by capturing incident-level data and a comprehensive description of what happened in each incident. Unlike the SRS, NIBRS collects data on each individual crime incident, capturing detailed information about the offenses, victims, offenders, property, and arrestees. NIBRS records data on 52 "Group A" offenses and 10 "Group B" offenses, covering a broader spectrum of criminal activity. It did away with the Hierarchy Rule and collects data on all offenses within a single incident, providing a fuller picture of criminal activity. Because NIBRS is incident-based, we have full access to the multivariate relationships between features of crime incidents.

On January 1, 2021, the FBI officially retired the SRS marking a significant transition towards the exclusive use of the National Incident-Based Reporting System for crime data collection and reporting. Although the transition has been planned for almost a decade, many law enforcement agencies are yet to transition their information systems to report to NIBRS. In 2024, four of the nation's largest states, California, New York, Pennsylvania, and Florida, had just begun to genuinely participate in NIBRS. You can find a map describing NIBRS coverage at the Bureau of Justice Statistics. The largest law enforcement agencies in those states regularly post crime data to their local open data portals, but you may not find those data in NIBRS yet. The transition to NIBRS also makes the study of long-term national crime trends challenging. Any study that spans the SRS-NIBRS transition will have to grapple with inconsistencies and gaps in data as agencies adapt to the new system.

The FBI accepts data from law enforcement agencies through March of the following year. That is, the FBI accepted NIBRS data for 2025 through March 2026. Some crimes committed in December 2025, for example, may be solved or result in an arrest in April 2026. Such a case would not be marked as "cleared" in the NIBRS data because the clearance came after the NIBRS 2025 submission end date. Crimes in January 2025 may seem to be solved at a higher rate than crimes in December 2025 because of the March 2026 censoring. This is a feature of the data that simply requires care. As our earlier work with the NCVS emphasized, NIBRS also contains only crimes reported to the police, and reporting rates can vary greatly by type of crime, victim features, geography, and other characteristics.

2 Structured Query Language (SQL) and SQLite

The NIBRS 2025 master file contains 70 million records, far too large to load into R's memory all at once, and too complex to manage as a collection of flat files. This is where a relational database management system becomes essential. Rather than storing all data in one monolithic data frame, a relational database organizes information into multiple tables that are linked together by shared keys. In the case of NIBRS, that means separate tables for offenses, victims, offenders, property, and arrests, all linked by the ORI and incident number that uniquely identify each incident. This structure eliminates redundancy, keeps related data together, and makes it possible to answer complex multi-table questions efficiently. Perhaps most importantly for our purposes, a relational database lets you query a specific slice of the data without loading everything into memory first. A question like "how many completed burglaries occurred in Texas in 2025?" can be answered in a fraction of a second against 70 million records without R ever seeing any row outside of Texas.

Structured Query Language, or SQL (pronounced "sequel"), is the standard language for communicating with a relational database. SQL has been standardized by the American National Standards Institute since 1986 and is supported by virtually every database system in existence. One of SQL's most important characteristics is that it is a *declarative* language rather than an *imperative* one. In R you write code that tells the computer *how* to do something:

loop over these rows, filter this column, group by that variable. SQL instead lets you describe *what* you want, and the database engine figures out the most efficient way to retrieve it. You write `SELECT crime_type, COUNT(*) FROM offense GROUP BY crime_type` and the engine decides on its own how to scan, filter, and aggregate the data as fast as possible. This frees you to think about the question rather than the mechanics of retrieval, and it means the same SQL query will run correctly (and fast) whether the table has a thousand rows or a billion.

We will use SQLite as our database engine. SQLite differs from most database systems in one important way: it stores the entire database, all tables, indexes, and metadata, in a single ordinary file on disk. There is no server to install, no background process to manage, and no administrator credentials to configure. You simply point your code at the `.db` file and start querying. SQLite is also released into the public domain, meaning there are no licensing fees, no restrictions on use, and no vendor lock-in. Despite its simplicity, SQLite is far from a toy system. It is the most widely deployed database engine in the world by a large margin. Estimates place the number of active SQLite databases in the trillions. It runs inside every iPhone and Android phone, every Mac and Windows 10 or later machine, every Chrome and Firefox browser, every Airbus A350 flight information system, and countless other devices and applications. The skills you develop querying SQLite transfer directly to PostgreSQL, MySQL, Microsoft SQL Server, Oracle, and other enterprise database systems, since the core SQL language is largely the same across all of them.

3 Acquiring the data

The complete NIBRS data are available from the FBI's Crime Data Explorer downloads page under the Master File Downloads section heading. From the dropdown menu, select "National Incident-Based Reporting System (NIBRS)." The compressed data file, `nibrs-2025.zip`, is over 500 MB, so downloading will take some time. Leave it in the ZIP archive because R can read the compressed file directly.

NIBRS is too large to work with comfortably in R alone, so we will load it into a SQLite database. Once the data is in SQLite, we can query any subset of the 70 million records nearly instantaneously without ever loading the entire dataset into R's memory. We will use the `RSQLite` package to connect R to SQLite.

Let's start by loading the libraries we will need and peeking at a few lines of the data.

```
library(readr)
library(dplyr)
library(RSQLite)

# Set rebuildDB to TRUE to rebuild the database from the master file
#   (deletes any existing nibrs2025.db and re-reads all 70 million records,
#    which takes several minutes). Set to FALSE to reuse the existing
```

```
#   nibrs2025.db and skip the slow rebuild.
rebuildDB <- TRUE

# find out what files are packed inside the zip file
unzip("NIBRS/nibrs-2025.zip", list = TRUE)
```

```
                                Name     Length                Date
1 2025_NIBRS_NATIONAL_MASTER_FILE.txt 5801319730 2026-03-15 17:42:00
```

```
# create a connection to the compressed data file
con <- unz("NIBRS/nibrs-2025.zip", "2025_NIBRS_NATIONAL_MASTER_FILE.txt")
# read first 5 lines
scan(con, nlines=5, what="", sep="\n")
```

```
[1] "BH50AK00101000000000000000                ANCHORAGE                      AK1C941Y
[2] "BH50AK00102000000000000000        20210101FAIRBANKS                      AK4 941Y
[3] "0750AK00102000L1G0BRVSCTD2025020301    O90C01    30MINN "
[4] "0750AK00102000L1H0BRVSCTD2025011401    T90C01    15MWNRR"
[5] "0750AK00102000L1K0BRVSCTD2025011701    O90C01    20FINR "
```

As you can see, the data are not pretty. These data are in "fixed-width format." Rather than separating each field with a delimiter such as a comma or tab, fixed-width data place all fields side by side. This legacy format has some benefits. First, there is no need to store delimiters. This matters less today because storage is inexpensive, but delimiters in a dataset this size would require about 1 GB just to store commas. Instead, the data come with a separate file that describes which positions correspond to each field. The following table shows the first five fields of the NIBRS offense segment in the included `NIBRS Records Description updated.xlsx` file.

Positions	Field Length and Type	Field Name
1-2	A2	SEGMENT LEVEL
3-4	N2	NUMERIC STATE CODE
5-13	A9	ORIGINATING AGENCY IDENTIFIER (ORI)
14-25	A12	INCIDENT NUMBER
26-33	A8	INCIDENT DATE
34-36	A3	UCR OFFENSE CODE
…	…	…

The table tells us that for offense segments, the first two characters represent the segment level, characters 3 and 4 capture the state numeric code, characters 5-13 capture the ORI (a unique

identifier for a law enforcement agency), and so on. The second column describes the data type, (A)lphanumeric or (N)umeric, and the width (number of characters) of the data field.

NIBRS uses separate tables for administrative records, offenses, victims, offenders, property, and arrests. The data file interleaves records from all of these tables. The first two characters of each row indicate which table the row belongs to.

Segment code	Segment type
BH	batch header
01	administrative
02	offenses
03	property
04	victims
05	offender
06	arrestee
07	Group B arrests
W1	Incomplete admin
W3	Incomplete property
W6	Incomplete arrest

The W1, W3, and W6 codes identify "window segments" and represent partial reporting. These are relatively rare records and mostly relate to arrests or recovered property connected to offenses that do not appear in the offense segment, most likely because the agency transitioned to NIBRS between the offense and the recovery or arrest.

4 The NIBRS format lookup table

I have created a lookup table called `nibrs_format.csv` and included it with the course materials. This table tells R exactly how wide each column is and what type of data it contains for every segment.

```
nibrs_format <- read.csv("NIBRS/nibrs_format.csv")
head(nibrs_format)
```

```
                          col_name col_type col_width segment
1                    segment_level        c         2      01
2                numeric_state_code        n         2      01
3 originating_agency_identifier_ori        c         9      01
4                   incident_number        c        12      01
5                     incident_date        c         8      01
6             report_date_indicator        c         1      01
```

The `nibrs_format` table has four columns:

- `col_name`: the cleaned-up column name we will use in our database
- `col_type`: "c" for character (text) or "n" for numeric
- `col_width`: the number of characters that this column occupies in the fixed-width file
- `segment`: the two-character segment code (01, 02, 03, 04, 05, 06, 07, BH)

Here, for example, are the column specifications for the offense segment.

```
nibrs_format |> filter(segment=="02")
```

```
                                col_name col_type col_width segment
1                          segment_level        c         2      02
2                       numeric_state_code        n         2      02
3        originating_agency_identifier_ori        c         9      02
4                          incident_number        c        12      02
5                            incident_date        c         8      02
6                         ucr_offense_code        c         3      02
7              offense_attempted_completed        c         1      02
8                     suspected_of_using_1        c         1      02
9                     suspected_of_using_2        c         1      02
10                    suspected_of_using_3        c         1      02
11                           location_type        c         2      02
12             number_of_premises_entered        c         2      02
13                          method_of_entry        c         1      02
14 criminal_activity_1_gang_information_1        c         1      02
15 criminal_activity_2_gang_information_2        c         1      02
16                     criminal_activity_3        c         1      02
17                          weapon_force_1        c         2      02
18                  automatic_indicator_1        c         1      02
19                          weapon_force_2        c         3      02
20                          weapon_force_3        c         3      02
21                        bias_motivation        c         2      02
```

The code used to create `nibrs_format.csv` from the Excel documentation file is included for reference in Section 10. You do not need to run it to complete this chapter.

5 Setting up the SQLite database

We will create a SQLite database called `nibrs2025.db` to store all the NIBRS segments. Once built, you can query the database from R in any future session without rebuilding it… simply reconnect.

```
# Create a new database (or connect if it already exists)
con <- dbConnect(SQLite(), dbname="NIBRS/nibrs2025.db")
```

6 Loading the data into SQLite

The master file interleaves all the segment records, including the window-segment variants, in a single enormous file. Our goal is to separate them into eight destination tables in the SQLite database. Here is the strategy.

We will read the master file one million rows at a time. Each batch gets split by the first two characters of each line, which is the segment code. Then, for each segment in that batch, we parse the fixed-width lines directly into a data frame and append it to the appropriate SQLite table. We repeat until the file is exhausted.

The key to making this fast is `I()`. Normally `read_fwf()` expects a file path as its first argument. Wrapping a character vector in `I()`, which stands for "as-is", tells `read_fwf()` to treat the vector itself as the data rather than as a path to a file. This means we can feed each batch of raw text lines directly to `read_fwf()` without first writing them to an intermediate file. The result is that we read the master file exactly once and write each row to SQLite exactly once, rather than touching the data twice with a set of intermediate files in between.

The other performance trick here is `dbBegin()` and `dbCommit()`. By default, every `dbWriteTable()` call is its own transaction. SQLite flushes its write buffer to disk after every one. With 70 million records spread across dozens of loop iterations, that amounts to hundreds of individual disk commits and can slow the process to a crawl. `dbBegin()` opens a single transaction that spans the entire loop. SQLite accumulates all the writes in memory and only commits them to disk once when `dbCommit()` is called at the end. This can make bulk loading an order of magnitude faster.

Building the database can take many minutes and will produce a `nibrs2025.db` database file that is over 5 GB.

```
# Map segment codes to SQLite table names
seg_tables <- list(
 "01" = "admin",
 "02" = "offense",
 "03" = "property",
 "04" = "victim",
 "05" = "offender",
 "06" = "arrestee",
 "07" = "groupb_arrest",
 "BH" = "batch_header"
```

```
)

# Start fresh - remove any existing tables
for (tbl in unlist(seg_tables)) {
  if (dbExistsTable(con, tbl)) dbRemoveTable(con, tbl)
}

infile <- unz("NIBRS/nibrs-2025.zip",
              "2025_NIBRS_NATIONAL_MASTER_FILE.txt",
              open = "r") # keep open for reading
dbBegin(con)  # open one transaction for the entire load
linesRead <- 0
while ((length(a <- readLines(infile, n=1000000)) > 0))
{
   linesRead <- linesRead + length(a)
   if(linesRead <= 10000000 ||       # less than 10M
      linesRead %% 10000000 == 0 ||  # a multiple of 10M
      length(a) <  1000000)          # the last lines read
   {
      message("Lines read: ", format(linesRead, big.mark=",", scientific=FALSE))
   }

   dSplit <- split(a, substring(a, 1, 2))
   # Merge window segments with their corresponding complete segments
   dSplit[["01"]] <- c(dSplit[["01"]], dSplit[["W1"]])
   dSplit[["03"]] <- c(dSplit[["03"]], dSplit[["W3"]])
   dSplit[["06"]] <- c(dSplit[["06"]], dSplit[["W6"]])
   dSplit[c("W1","W3","W6")] <- NULL

   for (seg in names(dSplit))
   {
      fmt <- nibrs_format |> filter(segment == seg)

      df <-
         read_fwf(I(dSplit[[seg]]),            # parse directly from memory
                  col_positions  = fwf_widths(fmt$col_width),
                  col_type       = paste(fmt$col_type, collapse=""),
                  show_col_types = FALSE) |>
         rename_with(~fmt$col_name) |>
         rename(ori     = originating_agency_identifier_ori,
                segment = segment_level,
                state   = numeric_state_code)
```

```
      # Segment-specific adjustments
      if (seg == "03")
         df <- df |> mutate(value_of_property = as.numeric(value_of_property))

      # population covered is split by counties (for agencies spanning counties)
      #   compute total population
      if (seg == "BH")
      {
         df <- df |>
            mutate(total_pop = rowSums(across(starts_with("current_pop")),
                                       na.rm=TRUE))
      }

      dbWriteTable(con, seg_tables[[seg]], df, append=TRUE)
   }
}
```

```
Lines read: 1,000,000

Lines read: 2,000,000

Lines read: 3,000,000

Lines read: 4,000,000

Lines read: 5,000,000

Lines read: 6,000,000

Lines read: 7,000,000

Lines read: 8,000,000

Lines read: 9,000,000

Lines read: 10,000,000

Lines read: 20,000,000
```

```
Lines read: 30,000,000

Lines read: 40,000,000

Lines read: 50,000,000

Lines read: 60,000,000

Lines read: 69,201,709
```

```
dbCommit(con)  # flush everything to disk in one shot
close(infile)
```

Let's check what tables are now in our database.

```
dbListTables(con)
```

```
[1] "admin"         "arrestee"      "batch_header"  "groupb_arrest"
[5] "offender"      "offense"       "property"      "victim"
```

Our database now has eight data tables, one for each NIBRS segment. (We will add lookup tables later to translate the many codes in the data into readable labels.) Let's look at one record from Philadelphia in the administrative table.

```
dbGetQuery(con, "
    SELECT *
    FROM admin
    WHERE ori='PAPEP0000' AND
          incident_number='2H-0507296XO'
    LIMIT 1")
```

```
  segment state       ori incident_number incident_date report_date_indicator
1      01    37 PAPEP0000    2H-0507296XO      20250106                  <NA>
  incident_date_hour total_offense_segments total_victim_segments
1                 14                      1                     1
  total_offender_segments total_arrestee_segments city_submission
1                       1                       1           <NA>
  cleared_exceptionally exceptional_clearance_date offense_code_1
1                     N                      <NA>           <NA>
```

```
  offense_code_2 offense_code_3 offense_code_4 offense_code_5 offense_code_6
1           <NA>           <NA>           <NA>           <NA>           <NA>
  offense_code_7 offense_code_8 offense_code_9 offense_code_10
1           <NA>           <NA>           <NA>            <NA>
  cargo_theft_indicator
1                     N
```

The incident occurred in Pennsylvania (state 37), reported by the Philadelphia Police Department (ORI PAPEP0000), on January 6, 2025. The incident number 2H-0507296XO is only unique within an agency, so whenever linking to other segments always join on **both** `ori` and `incident_number`.

From here on, all exploration happens through SQL queries. Every time you want to work with your database in a new R session, just reconnect. No need to reload the data:

```
con <- dbConnect(SQLite(), dbname="NIBRS/nibrs2025.db")
```

7 Database normalization

A relational database usually stores related information in several tables rather than repeating everything in one enormous table. The process of organizing the data this way is called **database normalization**. Its basic principle is to store each fact in one place and connect related facts with keys.

NIBRS is naturally organized this way. The `admin` table stores information about an incident, while the `offense`, `victim`, `offender`, `property`, and `arrestee` tables store the things associated with that incident. One incident can involve several offenses, several victims, and several offenders. If all of this information were stored in one flat table, the incident information would have to be repeated for every possible combination of offense, victim, and offender. Besides wasting space, that repetition creates the possibility of inconsistent values for what should be the same incident.

Lookup tables are another form of normalization. For example, the `offense` table stores a compact UCR offense code, while `offense_lookup` stores the readable crime description associated with each code. If the description were repeated in every offense row, a spelling change or correction would require updating millions of records. With a lookup table, the description is stored once.

The tradeoff is that answering a question often requires a `JOIN`. In NIBRS, `incident_number` is unique only within a reporting agency, so the pair `ori` and `incident_number` is the key we use to connect incident-level tables. A key does not have to identify only one row in every table: the same incident key may correctly appear many times in `offense` or `victim`, reflecting a one-to-many relationship.

SQLite can show us the structure, or **schema**, of a table with `PRAGMA table_info()`. The result lists each column's name, storage type, missing-value restriction, default value, and whether it belongs to the primary key.

```
dbGetQuery(con, "PRAGMA table_info(offense)") |>
  head(10)
```

```
   cid                         name type notnull dflt_value pk
1    0                      segment TEXT       0         NA  0
2    1                        state REAL       0         NA  0
3    2                          ori TEXT       0         NA  0
4    3              incident_number TEXT       0         NA  0
5    4                incident_date TEXT       0         NA  0
6    5             ucr_offense_code TEXT       0         NA  0
7    6 offense_attempted_completed TEXT       0         NA  0
8    7         suspected_of_using_1 TEXT       0         NA  0
9    8         suspected_of_using_2 TEXT       0         NA  0
10   9         suspected_of_using_3 TEXT       0         NA  0
```

8 Exploring NIBRS with SQL

Now that the data are loaded, we can ask precise questions of our database without ever loading the full dataset into memory. We will cover the key SQL clauses using NIBRS examples. SQL is not case-sensitive, but the convention is to write keywords in ALL CAPS to distinguish them from table and column names.

8.1 `SELECT`, `FROM`, and `WHERE`

The three primary SQL clauses are `SELECT`, `FROM`, and `WHERE`. `SELECT` chooses which columns to return. `FROM` names the table where SQL will pull the data. `WHERE` filters rows. Look at the first five offense records.

```
dbGetQuery(con, "
  SELECT ori, incident_number, incident_date, ucr_offense_code
  FROM   offense
  LIMIT  5")
```

```
        ori incident_number incident_date ucr_offense_code
1 AK0010200    C30BRCZ9728N      20250301              13C
```

```
2 AK0010200    C30BREBQ728N      20250601              23G
3 AK0010200    C30BRERH728N      20250106              26F
4 AK0010200    C30BRETM728N      20251128              23D
5 AK0010200    C30BRFTC728N      20250618              23G
```

The `SELECT` clause includes the four columns that I wish to pull from the table listed in the `FROM` clause, offense. The SQL keyword `LIMIT` is like the R `head()` function and will just show us the first 5 rows of the result rather than the entire offense table. Because there is no `WHERE` clause, this query will pull all the rows from the offense table.

We can use `*` to select all columns.

```
dbGetQuery(con, "
  SELECT *
  FROM   offense
  LIMIT  3")
```

```
  segment state       ori incident_number incident_date ucr_offense_code
1      02    50 AK0010200    C30BRCZ9728N      20250301              13C
2      02    50 AK0010200    C30BREBQ728N      20250601              23G
3      02    50 AK0010200    C30BRERH728N      20250106              26F
  offense_attempted_completed suspected_of_using_1 suspected_of_using_2
1                           C                   N                 <NA>
2                           C                   N                 <NA>
3                           C                   N                 <NA>
  suspected_of_using_3 location_type number_of_premises_entered method_of_entry
1                 <NA>            08                      <NA>            <NA>
2                 <NA>            20                      <NA>            <NA>
3                 <NA>            09                      <NA>            <NA>
  criminal_activity_1_gang_information_1 criminal_activity_2_gang_information_2
1                                      N                                  <NA>
2                                   <NA>                                  <NA>
3                                   <NA>                                  <NA>
  criminal_activity_3 weapon_force_1 automatic_indicator_1 weapon_force_2
1                <NA>           <NA>                  <NA>           <NA>
2                <NA>           <NA>                  <NA>           <NA>
3                <NA>           <NA>                  <NA>           <NA>
  weapon_force_3 bias_motivation
1           <NA>              88
2           <NA>              88
3           <NA>              88
```

Let's look at the offense record for the Philadelphia incident we saw in the administrative segment. The `WHERE` clause can combine multiple conditions with `AND` and `OR`.

```
dbGetQuery(con, "
  SELECT *
  FROM   offense
  WHERE ori='PAPEP0000' AND
        incident_number='2H-0507296XO'")
```

```
  segment state       ori incident_number incident_date ucr_offense_code
1      02    37 PAPEP0000    2H-0507296XO      20250106              120
  offense_attempted_completed suspected_of_using_1 suspected_of_using_2
1                           C                    N                 <NA>
  suspected_of_using_3 location_type number_of_premises_entered method_of_entry
1                 <NA>            20                       <NA>            <NA>
  criminal_activity_1_gang_information_1 criminal_activity_2_gang_information_2
1                                  <NA>                                  <NA>
  criminal_activity_3 weapon_force_1 automatic_indicator_1 weapon_force_2
1                <NA>             40                  <NA>           <NA>
  weapon_force_3 bias_motivation
1           <NA>              88
```

The UCR offense code is 120 (robbery). `offense_attempted_completed` is `C`, meaning it was a completed crime rather than an attempted crime. The offender was not suspected of using drugs, alcohol, or computer equipment (`suspected_of_using_1` = `N`), and the location type is `20` (Residence/Home).

SQL uses a single `=` to test for equality, unlike R, which uses `==`. `ori = 'PAPEP0000'` tests whether `ori` equals that value. SQLite also accepts `==` as an extension, but standard SQL uses `=`, so using a single equals sign makes queries portable to other database systems. SQL has no separate assignment operator in a `WHERE` clause, so there is no ambiguity.

Text values in SQL `WHERE` clauses go inside single quotes. In standard SQL, double quotes instead identify table or column names that contain spaces, punctuation, or SQL keywords.

Column names should ideally contain only letters, numbers, and underscores and should not be SQL keywords. When an inherited database has an awkward name containing a space, punctuation, or a keyword, the identifier must be quoted. Standard SQL uses double quotes around identifiers, but SQLite also accepts square brackets, which are convenient inside an R string. Here we deliberately give a result column an awkward alias:

```
dbGetQuery(con, "
  SELECT incident_date AS [incident date]
  FROM   offense
  LIMIT  5")
```

```
  incident date
1      20250301
2      20250601
3      20250106
4      20251128
5      20250618
```

The same brackets protect an awkward name when it appears in `SELECT`, `FROM`, `WHERE`, or another clause. For example, an existing column named `incident date` would be written as `[incident date]`.

> **Exercises**
>
> 1. Using the `victim` table, show the ORI, incident number, age, and sex for the first 10 individual victims. Individual victims have `type_of_victim = 'I'`.
> 2. Using the `arrestee` table, show the incident number, arrest date, age, and arrest offense code for the first 10 arrest records from the Philadelphia Police Department (ORI PAPEP0000).

8.2 Regular expressions in SQL

SQLite can also use regular expressions in a `WHERE` clause. Once per R session, initialize RSQLite's regular-expression extension. The following pattern selects ORIs that begin with `PA`:

```
initExtension(con, "regexp")

dbGetQuery(con, "
  SELECT DISTINCT ori
  FROM   batch_header
  WHERE  ori REGEXP '^PA'
  LIMIT  10")
```

```
         ori
```

```
1  PA0010000
2  PA0010100
3  PA0010200
4  PA0010300
5  PA0010400
6  PA0010500
7  PA0010600
8  PA0010700
9  PA0010800
10 PA0011100
```

8.3 `COUNT()`, `GROUP BY`, `ORDER BY`, and `DISTINCT`

How many offenses were reported to the Philadelphia Police Department (ORI PAPEP0000)?

```
dbGetQuery(con, "
  SELECT COUNT(*) AS offense_count
  FROM   offense
  WHERE  ori='PAPEP0000'")
```

```
  offense_count
1        150292
```

`COUNT(*)` is an aggregate function that counts rows. Unlike `COUNT(column)`, it does not skip a row because a particular column contains `NULL`. You will commonly see `COUNT()` used in three ways.

- `COUNT(*)` for total number of rows
- `COUNT(column)` for nonmissing values
- `COUNT(DISTINCT column)` for unique nonmissing values

We use `AS` to give the result a readable column name.

`GROUP BY` groups rows by one or more columns and applies an aggregate function to each group. SQL aggregate functions include `COUNT()`, `SUM()`, `AVG()`, `MIN()`, and `MAX()`. `ORDER BY` then sorts the results. Add `DESC` after a column or expression to sort from largest to smallest; ascending order is the default. These clauses commonly appear together: first SQL forms the groups and calculates their counts, and then it sorts those counts.

The rule to remember is: every column in `SELECT` that is not inside an aggregate function must appear in `GROUP BY`.

Let's count offenses by UCR code to find the most commonly reported crime types.

```
dbGetQuery(con, "
  SELECT ucr_offense_code,
         COUNT(*) AS offense_count
  FROM   offense
  GROUP BY ucr_offense_code
  ORDER BY offense_count DESC
  LIMIT 10")
```

```
   ucr_offense_code offense_count
1               13B       2002482
2               290       1383058
3               23H       1323592
4               35A       1111418
5               23C       1042192
6               23F        680332
7               13C        666091
8               240        594354
9               13A        594193
10              220        571619
```

Why is `ucr_offense_code` in the `GROUP BY` clause? Because it is not being aggregated in the `SELECT` clause.

> **!** What goes in the `GROUP BY` clause?
>
> *Everything* in the `SELECT` clause not being aggregated.
> What happens if I forget? You will not get an error or warning. SQL will still give you a result, but the SQL standard does not define what that result will be. Most likely it will not be correct.

Those UCR codes are hard to interpret. This is a recurring pattern in NIBRS. Many fields are stored as short codes that require a separate lookup table to decode. We will deal with this properly in a later section by loading lookup tables into SQLite and joining them. For now, let's keep exploring.

Let's count offenses by state.

```
dbGetQuery(con, "
  SELECT state,
         COUNT(*) AS offense_count
  FROM   offense
  GROUP BY state
```

```
  ORDER BY offense_count DESC
  LIMIT 10")
```

```
   state offense_count
1     42       1488616
2      4       1434203
3     31        756418
4     12        546209
5     32        538290
6      9        498670
7     34        427236
8     21        398916
9     41        397026
10    10        390323
```

Again we have codes rather than names. I made a `nibrs_state_lookup.csv` file that maps numeric state codes to names. We can load it into R temporarily to decode the top result.

```
a <- read.csv("NIBRS/nibrs_state_lookup.csv")
head(a)
```

```
  state_code state_abbr     state_name
1         50         AK         Alaska
2          1         AL        Alabama
3          3         AR       Arkansas
4         54         AS American Samoa
5          2         AZ        Arizona
6          4         CA     California
```

```
a |> filter(state_code == 42)
```

```
  state_code state_abbr state_name
1         42         TX      Texas
```

State 42 is Texas, which tends to report more offenses than other states, not necessarily because it has more crime, but because it has very good NIBRS participation across both large and small agencies. Shortly we will load all the lookup tables into SQLite so we can decode codes directly in SQL queries rather than bouncing back to R.

Sometimes we want the unique values rather than a count for each value. `DISTINCT` removes duplicate rows from a result, and `ORDER BY` can sort the resulting unique values. Let's find the set of offense types present in the database.

```
dbGetQuery(con, "
  SELECT DISTINCT ucr_offense_code
  FROM   offense
  ORDER BY ucr_offense_code")
```

```
   ucr_offense_code
1               09A
2               09B
3               09C
4               100
5               11A
6               11B
7               11C
8               11D
9               120
10              13A
11              13B
12              13C
13              200
14              210
15              220
16              23A
17              23B
18              23C
19              23D
20              23E
21              23F
22              23G
23              23H
24              240
25              250
26              26A
27              26B
28              26C
29              26D
30              26E
31              26F
32              26G
33              26H
34              270
35              280
36              290
```

```
37              30A
38              30B
39              30C
40              30D
41              35A
42              35B
43              360
44              36A
45              36B
46              370
47              39A
48              39B
49              39C
50              39D
51              40A
52              40B
53              40C
54              49A
55              49B
56              510
57              520
58              521
59              522
60              526
61              58B
62              61A
63              61B
64              620
65              64A
66              64B
67              720
```

Every offense code in that list corresponds to a crime type in `nibrs_offense_lookup.csv`. Peek at it to get a sense of what is there.

```
read.csv("NIBRS/nibrs_offense_lookup.csv") |> head(10)
```

```
   ucr_code                              crime_cat
1       720                 Animal Cruelty Offenses
2       200                                 Arson
3       13A                       Assault Offenses
4       13B                       Assault Offenses
```

```
5       13C                          Assault Offenses
6       510                                   Bribery
7       220              Burglary/Breaking & Entering
8       250                    Counterfeiting/Forgery
9       290 Destruction/Damage/Vandalism of Property
10      35A                    Drug/Narcotic Offenses
                                      crime
1                            Animal Cruelty
2                                     Arson
3                         Aggravated Assault
4                             Simple Assault
5                               Intimidation
6                                    Bribery
7                Burglary/Breaking & Entering
8                      Counterfeiting/Forgery
9   Destruction/Damage/Vandalism of Property
10                  Drug/Narcotic Violations
```

Let's explore the property table.

```
dbGetQuery(con, "
  SELECT *
  FROM   property
  WHERE ori='PAPEP0000' AND
        incident_number='2H-0507296XO'")
```

```
  segment state       ori incident_number incident_date type_property_loss
1      03    37 PAPEP0000    2H-0507296XO      20250106                  5
2      03    37 PAPEP0000    2H-0507296XO      20250106                  7
  property_description value_of_property date_recovered
1                   16                20       20250106
2                   16                20           <NA>
  number_of_stolen_motor_vehicles number_of_recovered_motor_vehicles
1                            <NA>                                <NA>
2                            <NA>                                <NA>
  suspected_drug_type estimated_quantity estimated_quantity_1000ths
1                <NA>               <NA>                       <NA>
2                <NA>               <NA>                       <NA>
  type_measurement drug_involvement_2 drug_involvement_3
1             <NA>               <NA>               <NA>
2             <NA>               <NA>               <NA>
  window_ucr_offense_code_1 window_ucr_offense_code_2 window_ucr_offense_code_3
```

```
1                      <NA>                      <NA>                      <NA>
2                      <NA>                      <NA>                      <NA>
  window_ucr_offense_code_4 window_ucr_offense_code_5 window_ucr_offense_code_6
1                      <NA>                      <NA>                      <NA>
2                      <NA>                      <NA>                      <NA>
  window_ucr_offense_code_7 window_ucr_offense_code_8 window_ucr_offense_code_9
1                      <NA>                      <NA>                      <NA>
2                      <NA>                      <NA>                      <NA>
  window_ucr_offense_code_10
1                       <NA>
2                       <NA>
```

These show two records related to the Philadelphia robbery. With the same ORI and incident number, these relate to the same incident. I made a lookup table for the codes in `property_description`.

```
read.csv("NIBRS/nibrs_property_lookup.csv") |>
   filter(property_code == 16)
```

```
  property_code   property_type
1            16 Household Goods
```

We are getting a clearer picture. Someone broke into someone's home while they were there and had some household goods stolen. The `type_property_loss` column explains what exactly happened.

Code	Description
1	None
2	Burned (includes damage caused in fighting the fire)
3	Counterfeited/Forged
4	Destroyed/Damaged/Vandalized
5	Recovered (to impound property that was previously stolen)
6	Seized (to impound property that was not previously stolen)
7	Stolen/Etc. (includes bribed, defrauded, embezzled, extorted, ransomed, robbed, etc.)
8	Unknown

So the household goods were stolen (Code 7) and recovered (Code 5) on the same day. The data also give the estimated value of the items, here $20, but these values can look a little strange, so let's explore those a little more.

`MIN()`, `MAX()`, `AVG()`, and `SUM()` work with numeric columns. Here is the range of reported property values by type of property stolen. The `property_description` column is yet another code field. Code 75, for example, is "Portable Electronic Communications" (i.e., a mobile phone).

```
dbGetQuery(con, "
  SELECT property_description,
         COUNT(*)                          AS n,
         MIN(value_of_property)            AS min_value,
         MAX(value_of_property)            AS max_value,
         ROUND(AVG(value_of_property), 0) AS avg_value
  FROM   property
  WHERE  value_of_property > 1        -- 1 means 'unknown value'
    AND  type_property_loss = '7'     -- 7 means stolen
  GROUP BY property_description
  ORDER BY avg_value DESC
  LIMIT 10")
```

```
   property_description      n min_value max_value avg_value
1                    01    342         2 236478628    695791
2                    29    605         2   1049990     74047
3                    21  32884         2 400367874     27542
4                    37  50514         2  25000000     25708
5                    03 352881         2 729000000     25220
6                    05    584         2   1000000     24257
7                    78  28172         2 375007500     23889
8                    43  35012         2 600960674     23884
9                    15  20730         2  56028000     23772
10                   30    377         2   1000000     23538
```

SQL comments start with `--`. Anything after `--` on a line is ignored by the database engine.

The NIBRS documentation says "If the value of the property is unknown, the value will be one ("1") dollar." However, it is very common, particularly in older or official data collections to use 9, 99, -9, or other variations of lots of 9s to indicate missing values. Let's see if there are ORIs that seem to have a lot of 9s.

```
dbGetQuery(con, "
  SELECT ori,
         value_of_property,
         COUNT(*) AS property_records
  FROM property
```

```
  WHERE value_of_property IN (
    999999999, 99999999, 9999999, 999999, 99999,
    9999, 999, 99)
  GROUP BY ori,value_of_property
  ORDER BY property_records DESC
  LIMIT 10
")
```

```
         ori value_of_property property_records
1  NY0303000                99              578
2  NY0303000               999              330
3  TXHPD0000                99              211
4  TXSPD0000                99              169
5  CA0194200               999              165
6  NY0303000              9999              157
7  NY0303000             99999              140
8  NY0303000           9999999              137
9  CA0194200              9999              105
10 CA0194200                99               86
```

There are many values composed entirely of 9s, and they appear concentrated in a few departments that use this convention repeatedly.

It is hard to tell what to do with these. For now, let's replace 1 and 999999999 with `NULL`, SQL's missing value indicator.

```
dbExecute(con, "
  UPDATE property
  SET    value_of_property = NULL
  WHERE  value_of_property IN (1, 999999999)")
```

```
[1] 1495432
```

We use `dbExecute()` instead of `dbGetQuery()` since we are not retrieving data from the database. `dbExecute()` will return the number of rows that the update affects.

> **Exercises**
>
> 3. How many completed robbery offense records (UCR code 120) were reported to the Philadelphia Police Department (ORI PAPEP0000)?

4. How many distinct agencies reported at least one offense in Pennsylvania? Use `REGEXP` to identify ORIs beginning with PA.

5. How many offense records meet our definition of a shooting: aggravated assault (13A), completed (C), with firearm code 11, 12, 13, 14, or 15 in any weapon_force column?

8.4 Creating and changing tables

So far, `SELECT` queries have returned results to R. Sometimes we instead want to save a query result as a new table in the database. `CREATE TABLE ... AS SELECT ...` creates the table and fills it with the rows returned by the query. The following example makes a small table containing 100 randomly selected homicide records. Note the use of `RANDOM()` to put the results in a random order before taking the first 100.

```
# in case they already exist
dbExecute(con, "DROP TABLE IF EXISTS homicide_sample")
```

```
[1] 0
```

```
dbExecute(con, "DROP TABLE IF EXISTS homicide_review_sample")
```

```
[1] 0
```

```
dbExecute(con, "
  CREATE TABLE homicide_sample AS
  SELECT ori, incident_number, incident_date
  FROM   offense
  WHERE  ucr_offense_code = '09A'
  ORDER BY RANDOM()
  LIMIT  100")
```

```
[1] 0
```

The new table is independent of the query that created it, meaning later changes to `offense` do not affect `homicide_sample`. Also, `CREATE TABLE AS` copies the query results but not indexes or constraints from the original table.

`ALTER TABLE` changes the structure of an existing table. Common operations include renaming a table, adding a column, renaming a column, and dropping a column. Let's rename that sample of homicide offenses.

```
dbListTables(con)
```

```
[1] "admin"           "arrestee"        "batch_header"    "groupb_arrest"
[5] "homicide_sample" "offender"        "offense"         "property"
[9] "victim"
```

```
dbExecute(con, "ALTER TABLE homicide_sample RENAME TO homicide_review_sample")
```

```
[1] 0
```

```
dbListTables(con)
```

```
[1] "admin"                 "arrestee"              "batch_header"
[4] "groupb_arrest"         "homicide_review_sample" "offender"
[7] "offense"               "property"              "victim"
```

The `batch_header` table contains a column called `date_state_began_ibr_for_law_enforcement_officers_ki` which is much too long.

```
# LIKE allows partial matching
#   % means zero or more characters
#   _ means exactly one character
# pragma_table_info() gives a table of table info
dbGetQuery(con, "
  SELECT *
  FROM pragma_table_info('batch_header')
  WHERE name LIKE '%leoka%'")
```

```
  cid
1  58
                                                                          name
1 date_state_began_ibr_for_law_enforcement_officers_killed_and_assaulted_leoka_data
  type notnull dflt_value pk
1 TEXT       0         NA  0
```

```
dbExecute(con, "
  ALTER TABLE batch_header RENAME COLUMN
    date_state_began_ibr_for_law_enforcement_officers_killed_and_assaulted_leoka_data
    TO date_leoka")
```

```
dbGetQuery(con, "
  SELECT *
  FROM pragma_table_info('batch_header')
  WHERE name LIKE '%leoka%'")
```

```
  cid       name type notnull dflt_value pk
1  58 date_leoka TEXT       0         NA  0
```

Some agencies, such as Oklahoma City OK, Kansas City MO, and High Point NC, span multiple counties. NIBRS records their population separately by county. When creating `batch_header`, we computed `total_pop` by summing the individual county populations. Since we no longer need the individual county populations, we can remove those columns to simplify the table. Dropping columns makes their storage available for SQLite to reuse but does not necessarily make the database file smaller. A later section explains how `VACUUM` can shrink the file.

```
dbGetQuery(con, "
  SELECT *
  FROM pragma_table_info('batch_header')
  WHERE name REGEXP '_[1-5]$'")
```

```
   cid                 name type notnull dflt_value pk
1   18 current_population_1 REAL       0         NA  0
2   19    ucr_county_code_1 TEXT       0         NA  0
3   20            msa_code_1 TEXT       0         NA  0
4   21    last_population_1 REAL       0         NA  0
5   22 current_population_2 REAL       0         NA  0
6   23    ucr_county_code_2 TEXT       0         NA  0
7   24            msa_code_2 TEXT       0         NA  0
8   25    last_population_2 REAL       0         NA  0
9   26 current_population_3 REAL       0         NA  0
10  27    ucr_county_code_3 TEXT       0         NA  0
11  28            msa_code_3 TEXT       0         NA  0
12  29    last_population_3 REAL       0         NA  0
13  30 current_population_4 REAL       0         NA  0
14  31    ucr_county_code_4 TEXT       0         NA  0
15  32            msa_code_4 TEXT       0         NA  0
16  33    last_population_4 REAL       0         NA  0
17  34 current_population_5 REAL       0         NA  0
18  35    ucr_county_code_5 TEXT       0         NA  0
19  36            msa_code_5 TEXT       0         NA  0
20  37    last_population_5 REAL       0         NA  0
```

```
21  53        fips_county_1 TEXT       0         NA  0
22  54        fips_county_2 TEXT       0         NA  0
23  55        fips_county_3 TEXT       0         NA  0
24  56        fips_county_4 TEXT       0         NA  0
25  57        fips_county_5 TEXT       0         NA  0
```

```
dbExecute(con, "ALTER TABLE batch_header DROP COLUMN current_population_1")
dbExecute(con, "ALTER TABLE batch_header DROP COLUMN current_population_2")
dbExecute(con, "ALTER TABLE batch_header DROP COLUMN current_population_3")
dbExecute(con, "ALTER TABLE batch_header DROP COLUMN current_population_4")
dbExecute(con, "ALTER TABLE batch_header DROP COLUMN current_population_5")
```

```
dbGetQuery(con, "
  SELECT *
  FROM pragma_table_info('batch_header')
  WHERE name REGEXP '_[1-5]$'")
```

```
   cid              name type notnull dflt_value pk
1   18 ucr_county_code_1 TEXT       0         NA  0
2   19        msa_code_1 TEXT       0         NA  0
3   20 last_population_1 REAL       0         NA  0
4   21 ucr_county_code_2 TEXT       0         NA  0
5   22        msa_code_2 TEXT       0         NA  0
6   23 last_population_2 REAL       0         NA  0
7   24 ucr_county_code_3 TEXT       0         NA  0
8   25        msa_code_3 TEXT       0         NA  0
9   26 last_population_3 REAL       0         NA  0
10  27 ucr_county_code_4 TEXT       0         NA  0
11  28        msa_code_4 TEXT       0         NA  0
12  29 last_population_4 REAL       0         NA  0
13  30 ucr_county_code_5 TEXT       0         NA  0
14  31        msa_code_5 TEXT       0         NA  0
15  32 last_population_5 REAL       0         NA  0
16  48     fips_county_1 TEXT       0         NA  0
17  49     fips_county_2 TEXT       0         NA  0
18  50     fips_county_3 TEXT       0         NA  0
19  51     fips_county_4 TEXT       0         NA  0
20  52     fips_county_5 TEXT       0         NA  0
```

The `batch_header` table already has `number_of_months_reported`, but let's double-check it against the month-by-month reporting fields.

```
# drop the column if it already exists
if ("reported_all_12_months" %in% dbListFields(con, "batch_header")) {
  dbExecute(con, "
    ALTER TABLE batch_header
    DROP COLUMN reported_all_12_months")
}

# create the new column filled with 0s
dbExecute(con, "
  ALTER TABLE batch_header
  ADD COLUMN reported_all_12_months INTEGER NOT NULL DEFAULT 0
")
```

```
[1] 0
```

```
dbExecute(con, "
  UPDATE batch_header
  SET reported_all_12_months =
    CASE
      WHEN january   != 'NNN'
       AND february  != 'NNN'
       AND march     != 'NNN'
       AND april     != 'NNN'
       AND may       != 'NNN'
       AND june      != 'NNN'
       AND july      != 'NNN'
       AND august    != 'NNN'
       AND september != 'NNN'
       AND october   != 'NNN'
       AND november  != 'NNN'
       AND december  != 'NNN'
      THEN 1
      ELSE 0
    END
")
```

```
[1] 22485
```

```
dbGetQuery(con, "
  SELECT reported_all_12_months,
         number_of_months_reported,
```

```
        COUNT(*) AS n
  FROM batch_header
  GROUP BY reported_all_12_months,
           number_of_months_reported")
```

```
   reported_all_12_months number_of_months_reported     n
1                       0                         0    51
2                       0                        00  7373
3                       0                        01   188
4                       0                        02   155
5                       0                        03   140
6                       0                        04   130
7                       0                        05   145
8                       0                        06   177
9                       0                        07   183
10                      0                        08   212
11                      0                        09   325
12                      0                        10   423
13                      0                        11  1001
14                      0                        12   120
15                      0                        60     4
16                      0                        61     8
17                      1                        12 11824
18                      1                        21    26
```

Note that `number_of_months_reported` has several strange values. Our own calculation may be preferable.

Deleting rows or dropping tables does not necessarily reduce the size of a SQLite database file. SQLite normally keeps the freed pages available for later use. `VACUUM` rebuilds the database and returns that unused space to the operating system.

```
dbExecute(con, "VACUUM")
```

On a multi-gigabyte database, `VACUUM` can take a long time, requires additional free disk space while it rebuilds the file, and prevents other connections from writing to the database. It is maintenance to run when substantial space needs to be reclaimed, not part of an ordinary analysis. It also cannot run inside an active transaction.

Exercise

6. Which 10 states have the largest number of fully reporting agencies? Use the `batch_header` table and the `reported_all_12_months` column.

8.5 Joining tables

So far, when we needed to interpret a code, we loaded the corresponding lookup file into R and searched it for a matching description. That works for an occasional code, but repeatedly moving between SQL results and R lookup tables is tedious and error-prone. A better approach is to load the lookup tables into SQLite and let SQL combine the codes with readable labels such as crime, location, weapon, property, city, and state names.

Joins are also necessary because NIBRS distributes the information for an incident across separate tables for offenses, victims, offenders, property, and arrestees. We link these incident tables using `ori` and `incident_number`. We link lookup tables using the code field and its corresponding lookup key. Both tasks use a SQL `JOIN`, so joins are central to working with NIBRS.

A `JOIN` is how SQL links two tables together. You name both tables in the `FROM` clause, then use an `ON` clause to specify which columns in each table contain the matching keys. For every row in the left table, SQL finds the matching row(s) in the right table and merges the columns together.

8.5.1 Loading lookup tables into SQLite

Although `dbWriteTable()` could create and populate the lookup table in a single line, it would infer only the columns' storage types and would not designate `ucr_code` as the table's primary key. Creating the table explicitly requires more code, but it records an important rule in the database schema: every row must have a nonmissing, unique UCR code. SQLite will therefore reject missing or duplicated codes rather than silently creating an ambiguous lookup table. The primary-key declaration also documents how the table should be joined to the offense data and automatically creates an index that can make code lookups more efficient. After establishing this structure, `dbAppendTable()` adds the CSV data without replacing the constraints defined for the table.

```
dbExecute(con, "DROP TABLE IF EXISTS offense_lookup")
```

```
[1] 0
```

```r
a <- read.csv("NIBRS/nibrs_offense_lookup.csv", colClasses = "character")
head(a)
```

```
  ucr_code               crime_cat              crime
1      720 Animal Cruelty Offenses     Animal Cruelty
2      200                   Arson              Arson
3      13A        Assault Offenses Aggravated Assault
4      13B        Assault Offenses     Simple Assault
5      13C        Assault Offenses       Intimidation
6      510                 Bribery            Bribery
```

```r
# create the table structure
#   set ucr_code to be the primary key
#     NOT NULL: every row must have a ucr_code
#     PRIMARY KEY: must be unique, no duplicates
dbExecute(con, "
  CREATE TABLE offense_lookup (
    ucr_code  TEXT NOT NULL PRIMARY KEY,
    crime_cat TEXT,
    crime     TEXT)")
```

```
[1] 0
```

```r
# populate with data
dbAppendTable(con, "offense_lookup", a)
```

```
[1] 63
```

```r
# note that ucr_code is now marked as the primary key
dbGetQuery(con, "PRAGMA table_info(offense_lookup)")
```

```
  cid      name type notnull dflt_value pk
1   0  ucr_code TEXT       1         NA  1
2   1 crime_cat TEXT       0         NA  0
3   2     crime TEXT       0         NA  0
```

```r
# can also see that SQLite has built an index for this table
dbGetQuery(con, "PRAGMA index_list(offense_lookup)")
```

```
  seq                                name unique origin partial
1   0 sqlite_autoindex_offense_lookup_1      1     pk       0
```

We will talk more about the purpose of an index later. Let's create the remaining lookup tables.

```
dbExecute(con, "DROP TABLE IF EXISTS state_lookup")
```

```
[1] 0
```

```
dbExecute(con, "DROP TABLE IF EXISTS location_lookup")
```

```
[1] 0
```

```
dbExecute(con, "DROP TABLE IF EXISTS weapon_lookup")
```

```
[1] 0
```

```
dbExecute(con, "DROP TABLE IF EXISTS property_lookup")
```

```
[1] 0
```

```
dbExecute(con, "DROP TABLE IF EXISTS drug_lookup")
```

```
[1] 0
```

```
dbExecute(con, "DROP TABLE IF EXISTS circumstance_lookup")
```

```
[1] 0
```

```
a <- read.csv("NIBRS/nibrs_state_lookup.csv", colClasses = "character")
# index_list() will not show an index for integer primary keys (it's okay)
dbExecute(con, "
  CREATE TABLE state_lookup (
    state_code INTEGER NOT NULL PRIMARY KEY,
    state_abbr TEXT,
    state_name TEXT)")
```

```
[1] 0
```

```
dbAppendTable(con, "state_lookup", a)
```

```
[1] 56
```

```
a <- read.csv("NIBRS/nibrs_location_lookup.csv", colClasses = "character")
dbExecute(con, "
  CREATE TABLE location_lookup (
    location_code TEXT NOT NULL PRIMARY KEY,
    location_type TEXT)")
```

```
[1] 0
```

```
dbAppendTable(con, "location_lookup", a)
```

```
[1] 45
```

```
a <- read.csv("NIBRS/nibrs_weapon_lookup.csv", colClasses = "character")
dbExecute(con, "
  CREATE TABLE weapon_lookup (
    weapon_code TEXT NOT NULL PRIMARY KEY,
    weapon_type TEXT)")
```

```
[1] 0
```

```
dbAppendTable(con, "weapon_lookup", a)
```

```
[1] 17
```

```
a <- read.csv("NIBRS/nibrs_property_lookup.csv", colClasses = "character")
dbExecute(con, "
  CREATE TABLE property_lookup (
    property_code TEXT NOT NULL PRIMARY KEY,
    property_type TEXT)")
```

```
[1] 0
```

```
dbAppendTable(con, "property_lookup", a)
```

```
[1] 68
```

```
a <- read.csv("NIBRS/nibrs_drug_lookup.csv", colClasses = "character")
dbExecute(con, "
  CREATE TABLE drug_lookup (
    drug_code TEXT NOT NULL PRIMARY KEY,
    drug_type TEXT)")
```

```
[1] 0
```

```
dbAppendTable(con, "drug_lookup", a)
```

```
[1] 18
```

```
a <- read.csv("NIBRS/nibrs_circumstance_lookup.csv", colClasses = "character")
dbExecute(con, "
  CREATE TABLE circumstance_lookup (
    circumstance_code TEXT NOT NULL PRIMARY KEY,
    circumstance      TEXT)")
```

```
[1] 0
```

```
dbAppendTable(con, "circumstance_lookup", a)
```

```
[1] 17
```

```
dbListTables(con)
```

```
 [1] "admin"                 "arrestee"              "batch_header"
 [4] "circumstance_lookup"   "drug_lookup"          "groupb_arrest"
 [7] "homicide_review_sample" "location_lookup"      "offender"
[10] "offense"               "offense_lookup"        "property"
[13] "property_lookup"       "state_lookup"          "victim"
[16] "weapon_lookup"
```

Our database now has sixteen tables: the eight segment tables we loaded from the master file, the seven lookup tables, and the homicide sample.

8.5.2 `INNER JOIN`

An `INNER JOIN` returns only rows where the join key exists in **both** tables. Rows with no match in the other table are dropped entirely.

The syntax is:

```
FROM left_table
  INNER JOIN right_table
    ON left_table.key_column = right_table.key_column
```

The offense table just has offense codes. We can look at the most common crime codes reported to the Philadelphia PD.

```
dbGetQuery(con, "
  SELECT ucr_offense_code,
         COUNT(*) AS offense_count
  FROM   offense
  WHERE  offense.ori = 'PAPEP0000'
  GROUP BY ucr_offense_code
  ORDER BY offense_count DESC
  LIMIT 10")
```

```
   ucr_offense_code offense_count
1               290         22534
2               23C         19862
3               240         15920
4               13B         14485
5               23H         11385
6               13C          9313
7               26A          8864
8               23F          8783
9               13A          7815
10              520          5806
```

It's not very useful without knowing what those codes are. Let's use `offense_lookup` to replace UCR codes with readable crime names for the top offenses reported to the Philadelphia Police Department.

```
dbGetQuery(con, "
  SELECT offense_lookup.crime,
         COUNT(*) AS offense_count
  FROM   offense
    INNER JOIN offense_lookup
      ON offense.ucr_offense_code = offense_lookup.ucr_code
  WHERE  offense.ori = 'PAPEP0000'
  GROUP BY offense_lookup.crime
  ORDER BY offense_count DESC
  LIMIT 10")
```

```
                                      crime offense_count
1  Destruction/Damage/Vandalism of Property         22534
2                               Shoplifting         19862
3                     Motor Vehicle Theft           15920
4                          Simple Assault           14485
5                        All Other Larceny          11385
6                             Intimidation           9313
7   False Pretenses/Swindle/Confidence Game           8864
8                 Theft From Motor Vehicle           8783
9                       Aggravated Assault           7815
10                   Weapon Law Violations           5806
```

When a column name could exist in multiple tables (like `ori`), prefix it with the table name, `offense.ori`, so SQL knows exactly where to look. The general practice is to always prefix in JOIN queries for clarity, even when it is technically unambiguous.

Now let's join the offense table to `batch_header` to find the cities that report the most offenses. The `batch_header` has one row per agency and the `offense` table has one row per offense. The join brings city and state names into the offense counts.

```
dbGetQuery(con, "
  SELECT batch_header.city_name,
         batch_header.state_abbreviation,
         COUNT(*) AS offense_count
  FROM   offense
    INNER JOIN batch_header
      ON offense.ori = batch_header.ori
  GROUP BY batch_header.city_name, batch_header.state_abbreviation
  ORDER BY offense_count DESC
  LIMIT 10")
```

```
      city_name state_abbreviation offense_count
1      NEW YORK                NY        557740
2       HOUSTON                TX        281499
3       CHICAGO                IL        233671
4   LOS ANGELES                CA        199450
5   SAN ANTONIO                TX        156088
6  PHILADELPHIA                PA        151196
7     LAS VEGAS                NV        107123
8        DALLAS                TX         97251
9       MEMPHIS                TN         90898
10       AUSTIN                TX         84607
```

Where do crimes happen? Joining `location_lookup` answers that immediately.

```
dbGetQuery(con, "
  SELECT location_lookup.location_type,
         COUNT(*) AS offense_count
  FROM   offense
    INNER JOIN location_lookup
      ON offense.location_type = location_lookup.location_code
  GROUP BY location_lookup.location_type
  ORDER BY offense_count DESC
  LIMIT 10")
```

```
                   location_type offense_count
1                Residence/Home       4880137
2             Highway/Road/Alley       2433678
3             Parking Lot/Garage        964292
4                  Other/Unknown        657056
5      Department/Discount Store        567994
6            Grocery/Supermarket        333057
7                Specialty Store        301840
8              Convenience Store        301591
9     Commercial/Office Building        254213
10 School - Elementary/Secondary        228158
```

Residences and homes are the most frequently reported offense locations. These counts do not by themselves measure individual risk because they do not account for how much time people spend in each type of location.

What weapons are most commonly involved in aggravated assaults? The offense table stores up to three weapon codes per offense in `weapon_force_1`, `weapon_force_2`, and `weapon_force_3`.

`UNION ALL` stacks all three columns into one so we can count across them together, then we join `weapon_lookup` for readable labels.

```
dbGetQuery(con, "
  WITH all_weapons AS (
      SELECT weapon_force_1 AS weapon_code
      FROM offense
      WHERE ucr_offense_code = '13A' AND
            weapon_force_1 IS NOT NULL
    UNION ALL
      SELECT weapon_force_2
      FROM offense
      WHERE ucr_offense_code = '13A' AND
            weapon_force_2 IS NOT NULL
    UNION ALL
      SELECT weapon_force_3
      FROM offense
      WHERE ucr_offense_code = '13A' AND
            weapon_force_3 IS NOT NULL
  )
  SELECT weapon_lookup.weapon_type,
         COUNT(*) AS n
  FROM   all_weapons
    INNER JOIN weapon_lookup ON
       all_weapons.weapon_code = weapon_lookup.weapon_code
  GROUP BY weapon_lookup.weapon_type
  ORDER BY n DESC")
```

```
                          weapon_type      n
1  Personal Weapons (hands/feet/teeth) 145335
2             Knife/Cutting Instrument 106873
3                              Handgun  92670
4                                Other  68189
5                          Blunt Object  65076
6             Firearm (type not stated)  51246
7                  Motor Vehicle/Vessel  29150
8                          Asphyxiation  25293
9                                  None  15826
10                              Unknown  13381
11                                Rifle   6245
12                        Other Firearm   4411
13                              Shotgun   2735
```

```
14              Fire/Incendiary Device   1227
15      Drugs/Narcotics/Sleeping Pills   1064
16              Poison (including gas)    743
17                          Explosives    326
```

Note the use of `WITH` here. It creates a temporary result called `all_weapons` that the `FROM` clause in the main query can use. This is an example of a Common Table Expression (CTE), which we will discuss more later.

Now let's revisit the property value query from earlier. Instead of raw numeric codes we join `property_lookup` to get readable property type names.

```
dbGetQuery(con, "
  SELECT property_lookup.property_type,
         COUNT(*)                          AS n,
         MIN(value_of_property)            AS min_value,
         MAX(value_of_property)            AS max_value,
         ROUND(AVG(value_of_property), 0) AS avg_value,
         MEDIAN(value_of_property)         AS med_value
  FROM   property
    INNER JOIN property_lookup
      ON property.property_description = property_lookup.property_code
  WHERE  value_of_property IS NOT NULL
    AND  type_property_loss = '7'       -- stolen
  GROUP BY property_lookup.property_type
  ORDER BY med_value DESC
  LIMIT 10")
```

```
                            property_type      n min_value max_value avg_value
1                                  Trucks  50514         2  25000000     25708
2                                   Buses    584         2   1000000     24257
3                                Automobiles 352881         2 729000000     25220
4                                   Trailers  28172         2 375007500     23889
5                     Other Motor Vehicles  49057         2   8312023      9797
6  Heavy Construction/Industrial Equipment  20730         2  56028000     23772
7  Structures - Single Occupancy Dwellings    605         2   1049990     74047
8                   Recreational Vehicles  11779         2   1821150      9063
9                                Watercraft   2854         2   4000000     13632
10   Structures - Industrial/Manufacturing    244         2    257510     11844
   med_value
1      15000
2      12750
```

```
3       10000
4        4900
5        3500
6        3500
7        3000
8        3000
9        2425
10       2230
```

! Extreme values warrant scrutiny even after cleaning

Even after replacing the 1 and 999999999 with `NULL`, extreme outliers can still hide data entry errors. Here we see that there is a report of a stolen automobile worth $729M. It seems a Mobile, Alabama car dealership got swindled out of a $729M car on November 1, 2025. Curiously, there are two other cases of individuals losing $729M from a parking lot/garage and from their home. Plausible?
The NIBRS data contains no quality check preventing the entering of any dollar amount. When computing aggregate property values, consider using the median, capping or excluding implausible outliers, and always inspect the top of the distribution before reporting results.

To obtain more details about these incidents, I needed to join several tables. The query joins

- `property` to `property_lookup` to identify the type of property
- `offense` and `offense_lookup` to identify the associated crime
- `location_lookup` to describe where the offense occurred
- `victim` to obtain the victim type
- `batch_header` to identify the reporting city

The `offense` and `victim` tables are linked by the combination of `ori` and `incident_number`, while the lookup tables are linked by their respective code fields.

i Incident-level joins can produce combinations

The data do not directly link a particular property record to a particular victim. This query therefore joins them at the incident level. If an incident has multiple offenses and multiple victims, the result will contain every offense-by-victim combination. For example, an incident with two offenses and three victims can produce six rows for the same property record. These rows describe the records associated with the incident; they do not establish which victim was associated with each offense or item of property.

Let's time the query. Because it joins several large tables, it might take several minutes.

```
property729_query <- "
  SELECT batch_header.city_name,
         property.ori,
         property.incident_number,
         property.incident_date,
         property_lookup.property_type,
         offense_lookup.crime,
         location_lookup.location_type,
         victim.type_of_victim
  FROM property
    INNER JOIN property_lookup
      ON property.property_description = property_lookup.property_code
    INNER JOIN offense
      ON property.ori = offense.ori
         AND property.incident_number = offense.incident_number
    INNER JOIN offense_lookup
      ON offense.ucr_offense_code = offense_lookup.ucr_code
    INNER JOIN location_lookup
      ON offense.location_type = location_lookup.location_code
    INNER JOIN victim
      ON property.ori = victim.ori
         AND property.incident_number = victim.incident_number
    INNER JOIN batch_header
      ON property.ori = batch_header.ori
  WHERE property.value_of_property = 729000000
    AND property.type_property_loss = '7'
 "

# about 4 minutes
system.time(
  property729 <- dbGetQuery(con, property729_query)
)
```

```
   user  system elapsed
  31.67   67.69  103.11
```

```
property729
```

```
  city_name       ori incident_number incident_date property_type
1    MOBILE AL0020100    E15FML7GW5-F      20250829         Money
2    MOBILE AL0020100    GD9C994MFZHG      20251101   Automobiles
```

```
3    MOBILE AL0020100    GG1WYUKR8IAJ      20250907         Money
                                      crime            location_type
1                        All Other Larceny       Parking Lot/Garage
2 False Pretenses/Swindle/Confidence Game Auto Dealership New/Used
3                      Theft From Building           Residence/Home
  type_of_victim
1              I
2              B
3              I
```

8.5.3 SQL indexes

Although the filter finds only a few property records, SQLite still needs to locate matching rows in the much larger `offense` and `victim` tables. Without an appropriate index, it scans millions of rows looking for a match. That is not an efficient way to find records.

SQL offers you the option of building an index. SQL indexes are just like the ones you would use in a book. Look in the index, which is sorted in a meaningful way, and locate the page that contains the topic you wanted to find. Without suitable indexes, SQL may repeatedly scan many rows to match each `ori` and `incident_number`. Composite indexes on `(ori, incident_number)` are especially useful here because those two columns together identify the incident being joined. The primary keys on the lookup tables already provide indexes for their code columns, so we do not need to create additional lookup-table indexes. You only need to build the index once. Once built, it will stay with the database, even if you restart your computer.

Let's add indexes that support the slow joins. The composite indexes on `offense` and `victim` help SQLite jump directly to the matching incident, and the `batch_header` index speeds up the agency lookup. The initial filter on `property` is already fast in this example, so it does not need another index.

```
dbExecute(con, "
  CREATE INDEX IF NOT EXISTS idx_offense
  ON offense(ori, incident_number)")
```

```
[1] 0
```

```
dbExecute(con, "
  CREATE INDEX IF NOT EXISTS idx_victim
  ON victim(ori, incident_number)")
```

```
[1] 0
```

```
dbExecute(con, "
  CREATE INDEX IF NOT EXISTS idx_batch_header
  ON batch_header(ori)")
```

```
[1] 0
```

Now rerun exactly the same query. SQLite automatically chooses an index when its query planner determines that using one will be helpful.

```
system.time(
  property729_indexed <- dbGetQuery(con, property729_query)
)
```

```
   user  system elapsed
   0.96    0.92    2.25
```

```
property729_indexed
```

```
  city_name       ori incident_number incident_date property_type
1    MOBILE AL0020100    E15FML7GW5-F      20250829         Money
2    MOBILE AL0020100    GD9C994MFZHG      20251101   Automobiles
3    MOBILE AL0020100    GG1WYUKR8IAJ      20250907         Money
                                    crime            location_type
1                        All Other Larceny       Parking Lot/Garage
2 False Pretenses/Swindle/Confidence Game Auto Dealership New/Used
3                      Theft From Building           Residence/Home
  type_of_victim
1              I
2              B
3              I
```

Wow! So much faster. It takes some upfront computing time to create an index (maybe a minute) and the index takes space on the computer (the offense and victim indexes each require about 400 Mb… the `batch_header` index is 0.5 Mb).

Exercises

7. What are the five most common location types for robbery (UCR code 120)? Join `offense` to `location_lookup` and report the number of robbery offense records for each location type.

8. How many cases are recorded as justifiable homicides? Check both the `offense` table (UCR code 09C) and the `victim` table (circumstance codes 20 and 21). Use `circumstance_lookup` to report the circumstance descriptions.

Code	Description
20	Criminal killed by private citizen
21	Criminal killed by police officer

8.5.4 `LEFT JOIN`

An `INNER JOIN` drops any row from the left table that has no match in the right table. A `LEFT JOIN` keeps all rows from the left table and fills in `NULL` for any columns from the right table that did not have a match. In our work, `LEFT JOIN` is almost always the right choice. For example, we have offenses that we want to link to arrestees. If for a particular offense no arrest has been made (yet), then typically we do not want to drop the offense from our dataset. Instead, we just want an `NA` for arrestee to indicate that no arrest has been made. A left join is the right choice when you do not want to accidentally drop records.

Start by examining the arrestee table.

```
dbGetQuery(con, "SELECT * from arrestee LIMIT 5")
```

```
  segment state       ori incident_number incident_date
1      06    50 AK0010200    CB0BAA5-728N      20250719
2      06    50 AK0010200    CB0BAA5L728N      20250819
3      06    50 AK0010200    CB0BAABD728N      20250818
4      06    50 AK0010200    CB0BAABR728N      20250817
5      06    50 AK0010200    CB0BAAM-728N      20250817
  arrestee_sequence_number arrest_transaction_number arrest_date type_of_arrest
1                        1                      57609    20251226              T
2                        1                      57114    20250819              0
3                        1                      57107    20250818              0
4                        1                      57104    20250817              0
5                        1                      57103    20250817              0
```

```
  multiple_arrestee_segments_indicator ucr_arrest_offense_code weapon_1
1                                    N                    26B       01
2                                    N                    13B       01
3                                    M                    13A       01
4                                    N                    220       01
5                                    N                    13B       01
  automatic_indicator_1 weapon_2 age_of_arrestee sex_of_arrestee
1                  <NA>     <NA>              39               M
2                  <NA>     <NA>              34               M
3                  <NA>     <NA>              24               M
4                  <NA>     <NA>              34               M
5                  <NA>     <NA>              28               F
  race_of_arrestee ethnicity_of_arrestee resident_status_of_arrestee
1                W                     N                           N
2                I                     N                           R
3                W                     H                           R
4                I                     N                           R
5                A                     N                           R
  disposition_of_arrestee_under_18 window_clearance_flag
1                             <NA>                  <NA>
2                             <NA>                  <NA>
3                             <NA>                  <NA>
4                             <NA>                  <NA>
5                             <NA>                  <NA>
  window_ucr_offense_code_1 window_ucr_offense_code_2 window_ucr_offense_code_3
1                      <NA>                      <NA>                      <NA>
2                      <NA>                      <NA>                      <NA>
3                      <NA>                      <NA>                      <NA>
4                      <NA>                      <NA>                      <NA>
5                      <NA>                      <NA>                      <NA>
  window_ucr_offense_code_4 window_ucr_offense_code_5 window_ucr_offense_code_6
1                      <NA>                      <NA>                      <NA>
2                      <NA>                      <NA>                      <NA>
3                      <NA>                      <NA>                      <NA>
4                      <NA>                      <NA>                      <NA>
5                      <NA>                      <NA>                      <NA>
  window_ucr_offense_code_7 window_ucr_offense_code_8 window_ucr_offense_code_9
1                      <NA>                      <NA>                      <NA>
2                      <NA>                      <NA>                      <NA>
3                      <NA>                      <NA>                      <NA>
4                      <NA>                      <NA>                      <NA>
5                      <NA>                      <NA>                      <NA>
  window_ucr_offense_code_10
```

```
1                       <NA>
2                       <NA>
3                       <NA>
4                       <NA>
5                       <NA>
```

These show records of five arrests that Fairbanks officers made. The data include age, sex, race, and ethnicity of the arrestee. Four of these arrests were "on-view" (`type_of_arrest` = O) and one was taken into custody based on a warrant or prior report (`type_of_arrest` = T). A third option would be a summons or citation (`type_of_arrest` = S).

There can be multiple arrests associated with the same incident. Here is a 2025 incident in Alabama in which three offenses occurred.

```
dbGetQuery(con, "
  SELECT ori, incident_number, ucr_offense_code,
         offense_lookup.crime
  FROM offense
    LEFT JOIN offense_lookup
      ON offense.ucr_offense_code = offense_lookup.ucr_code
  WHERE ori='AL0010000' AND incident_number='E88U3T5U0O-3'")
```

```
        ori incident_number ucr_offense_code                    crime
1 AL0010000    E88U3T5U0O-3              35A  Drug/Narcotic Violations
2 AL0010000    E88U3T5U0O-3              35B Drug Equipment Violations
3 AL0010000    E88U3T5U0O-3              520     Weapon Law Violations
```

The report lists three offenders.

```
dbGetQuery(con, "
  SELECT ori, incident_number, offender_sequence_number,
         age_of_offender, race_of_offender, ethnicity_of_offender,
         sex_of_offender
  FROM offender
  WHERE ori='AL0010000' AND incident_number='E88U3T5U0O-3'")
```

```
        ori incident_number offender_sequence_number age_of_offender
1 AL0010000    E88U3T5U0O-3                         1              19
2 AL0010000    E88U3T5U0O-3                         4              31
3 AL0010000    E88U3T5U0O-3                         5              28
  race_of_offender ethnicity_of_offender sex_of_offender
```

```
1                B                    N               F
2                B                    N               M
3                B                    N               M
```

And all three were arrested. We can line up the offender and arrestee records in this case using age, race, and sex, but offender sequence numbers do not correspond to arrestee sequence numbers.

```
dbGetQuery(con, "
  SELECT ori, incident_number, arrestee_sequence_number,
         ucr_arrest_offense_code,
         age_of_arrestee, race_of_arrestee,
         ethnicity_of_arrestee, sex_of_arrestee
  FROM arrestee
  WHERE ori='AL0010000' AND incident_number='E88U3T5U0O-3'")
```

```
       ori incident_number arrestee_sequence_number ucr_arrest_offense_code
1 AL0010000    E88U3T5U0O-3                       2                     35A
2 AL0010000    E88U3T5U0O-3                       5                     35A
3 AL0010000    E88U3T5U0O-3                       6                     520
  age_of_arrestee race_of_arrestee ethnicity_of_arrestee sex_of_arrestee
1              28                B                    N               M
2              19                B                    N               F
3              31                B                    N               M
```

Now that we have some familiarity with the arrestee table, let's compute the percentage of completed burglary offense records (UCR code 220) whose incident has at least one arrest record. Because we will link many offense records to the arrestee table, it is best to create an arrestee index too.

```
dbExecute(con, "
  CREATE INDEX IF NOT EXISTS idx_arrestee
  ON arrestee(ori, incident_number)")
```

```
[1] 0
```

Not all incidents will have a matching row in the `arrestee` table, so we use a `LEFT JOIN` to keep all burglaries whether or not an arrest was made.

```
dbGetQuery(con, "
  SELECT offense.incident_date,
         offense.ori,
         offense.incident_number,
         arr.ori                  AS arrest_ori,
         arr.incident_number      AS arrest_incident_number
  FROM   offense
    LEFT JOIN (SELECT DISTINCT ori, incident_number
               FROM   arrestee) AS arr
      ON offense.ori = arr.ori
         AND offense.incident_number = arr.incident_number
  WHERE  offense.ucr_offense_code = '220'
    AND  offense.offense_attempted_completed = 'C'
  LIMIT 10")
```

```
   incident_date       ori incident_number arrest_ori arrest_incident_number
1       20250817 AK0010200    CB0BAABR728N  AK0010200           CB0BAABR728N
2       20250908 AK0010200    CB0BABRR728N       <NA>                  <NA>
3       20250802 AK0010200    CB0BAC5-728N  AK0010200           CB0BAC5-728N
4       20250803 AK0010200    CB0BACEL728N       <NA>                  <NA>
5       20250805 AK0010200    CB0BACR9728N  AK0010200           CB0BACR9728N
6       20250805 AK0010200    CB0BACRL728N  AK0010200           CB0BACRL728N
7       20250706 AK0010200    CB0BAEM-728N  AK0010200           CB0BAEM-728N
8       20250724 AK0010200    CB0BAF59728N       <NA>                  <NA>
9       20250715 AK0010200    CB0BAFEC728N       <NA>                  <NA>
10      20250720 AK0010200    CB0BAFM-728N       <NA>                  <NA>
```

Where the arrest ORI and incident numbers are `NA`, the left join could not find an ORI/incident number match over in the `arrestee` table.

One offense can result in multiple arrests. Note the "subquery" used inside the left join. If there are multiple arrests for an incident, this subquery reduces them to a single row with `DISTINCT`. Otherwise, if one burglary resulted in 10 arrests, then that incident would get counted 10 times. As written, each offense gets one row and will check whether there are *any* associated arrests.

A cleaner and more efficient route to check for the existence of a match is to use `EXISTS()`.

```
dbGetQuery(con, "
  SELECT offense.incident_date,
         offense.ori,
         offense.incident_number,
         EXISTS (SELECT 1
```

```
                FROM arrestee
                WHERE arrestee.ori = offense.ori AND
                      arrestee.incident_number = offense.incident_number)
              AS arrest_made
  FROM offense
  WHERE offense.ucr_offense_code = '220'
    AND offense.offense_attempted_completed = 'C'
  LIMIT 10
")
```

```
   incident_date       ori incident_number arrest_made
1       20250817 AK0010200    CB0BAABR728N           1
2       20250908 AK0010200    CB0BABRR728N           0
3       20250802 AK0010200    CB0BAC5-728N           1
4       20250803 AK0010200    CB0BACEL728N           0
5       20250805 AK0010200    CB0BACR9728N           1
6       20250805 AK0010200    CB0BACRL728N           1
7       20250706 AK0010200    CB0BAEM-728N           1
8       20250724 AK0010200    CB0BAF59728N           0
9       20250715 AK0010200    CB0BAFEC728N           0
10      20250720 AK0010200    CB0BAFM-728N           0
```

Now let's count them. SQL evaluates all expressions in a `SELECT` clause simultaneously, so an alias defined in one column cannot be referenced by another column in the same `SELECT`. We cannot calculate a percentage using `total_burglaries` and `burglaries_with_arrest` in the same `SELECT` that defines those aliases; SQL does not yet know what they are. A Common Table Expression (CTE) solves this by computing the counts first and dividing in a second step.

```
dbGetQuery(con, "
  WITH burglary_counts AS (
    SELECT COUNT(*) AS total_burglaries,
           SUM(CASE WHEN EXISTS (
                   SELECT 1
                   FROM arrestee
                   WHERE arrestee.ori = offense.ori
                     AND arrestee.incident_number = offense.incident_number)
                 THEN 1
                 ELSE 0
               END) AS burglaries_with_arrest
    FROM offense
```

```
    WHERE offense.ucr_offense_code = '220'
      AND offense.offense_attempted_completed = 'C')
  SELECT total_burglaries,
         burglaries_with_arrest,
         ROUND(100.0 * burglaries_with_arrest /
                       NULLIF(total_burglaries, 0), 1) AS pct_arrested
  FROM burglary_counts")
```

```
  total_burglaries burglaries_with_arrest pct_arrested
1           520265                 85388         16.4
```

`CASE WHEN ... THEN ... ELSE ... END` is SQL's version of an if-else expression. Here, we add 1 when the incident has at least one arrest record and otherwise add 0. Summing those indicators gives the number of burglary offense records associated with an arrest, not the number of individual arrests. Dividing by the total number of burglary offense records gives the percentage associated with an arrest.

> **Exercise**
>
> 9. Among completed motor vehicle theft offense records (UCR code 240), how many have an incident with at least one arrest record, and what percentage is that? Use `EXISTS()`.

8.6 Date functions

SQLite stores dates as plain text. NIBRS incident dates arrive in `YYYYMMDD` format, without hyphens. Let's confirm:

```
dbGetQuery(con, "
  SELECT incident_date
  FROM   offense
  ORDER BY RANDOM()
  LIMIT 10")
```

```
   incident_date
1       20251231
2       20250721
3       20250616
4       20250527
5       20251204
```

```
6       20251231
7       20250725
8       20250427
9       20250519
10      20250317
```

SQLite's date functions (`STRFTIME()`, `DATE()`, etc.) expect the standard `YYYY-MM-DD` format, so we need to reformat `incident_date` before we can use them. The `UPDATE` statement edits rows in place. We use `SUBSTR()` to pull apart the year, month, and day and reassemble them with hyphens. The `||` operator concatenates strings in SQL.

```
dbExecute(con, "
  UPDATE offense
  SET    incident_date = SUBSTR(incident_date, 1, 4) || '-' ||
                         SUBSTR(incident_date, 5, 2) || '-' ||
                         SUBSTR(incident_date, 7, 2)
  WHERE  incident_date NOT LIKE '____-__-__'")
```

```
[1] 13060005
```

The `WHERE incident_date NOT LIKE '____-__-__'` clause is a safeguard. Each `_` matches any single character, so the pattern matches dates already in `YYYY-MM-DD` format. This means re-running the chunk will not corrupt already-formatted dates. The same fix applies to `incident_date` and `arrest_date` in the `arrestee` table.

```
dbExecute(con, "
  UPDATE arrestee
  SET    incident_date = SUBSTR(incident_date, 1, 4) || '-' ||
                         SUBSTR(incident_date, 5, 2) || '-' ||
                         SUBSTR(incident_date, 7, 2)
  WHERE  incident_date NOT LIKE '____-__-__'")
```

```
[1] 3603306
```

```
dbExecute(con, "
  UPDATE arrestee
  SET    arrest_date = SUBSTR(arrest_date, 1, 4) || '-' ||
                       SUBSTR(arrest_date, 5, 2) || '-' ||
                       SUBSTR(arrest_date, 7, 2)
  WHERE  arrest_date NOT LIKE '____-__-__'")
```

```
[1] 3603306
```

Now `STRFTIME()` works directly. It takes a format string and a date column. Common format codes are `%Y` (4-digit year), `%m` (month 01–12), `%d` (day), `%H` (hour on a 24-hour clock), and `%w` (day of week, 0=Sunday).

Importantly, `STRFTIME()` always returns **text**, even when the result looks like a number. This is fine for labels such as year-month, but convert the result with `CAST(... AS INTEGER)` before making a numerical comparison or calculation. Here we extract the month as an integer and keep offenses from October through December:

```
dbGetQuery(con, "
  SELECT incident_date,
         CAST(STRFTIME('%m', incident_date) AS INTEGER) AS month
  FROM   offense
  WHERE  CAST(STRFTIME('%m', incident_date) AS INTEGER) >= 10
  LIMIT  10")
```

```
   incident_date month
1     2025-11-28    11
2     2025-12-15    12
3     2025-10-24    10
4     2025-12-15    12
5     2025-11-16    11
6     2025-11-17    11
7     2025-10-17    10
8     2025-11-18    11
9     2025-11-15    11
10    2025-12-24    12
```

Let's count offenses by month to look for seasonal patterns.

```
dbGetQuery(con, "
  SELECT STRFTIME('%Y-%m', incident_date) AS year_month,
         COUNT(*)                        AS offense_count
  FROM   offense
  GROUP BY year_month
  ORDER BY year_month")
```

```
   year_month offense_count
1     2025-01       1076411
```

```
2     2025-02        987616
3     2025-03       1138702
4     2025-04       1119069
5     2025-05       1175069
6     2025-06       1114329
7     2025-07       1163900
8     2025-08       1151479
9     2025-09       1116660
10    2025-10       1101579
11    2025-11        993636
12    2025-12        921555
```

Now let's look at whether the percentage associated with an arrest varies by the month in which the offense occurred. Remember that the March 2026 submission cutoff means crimes committed late in 2025 have less time to produce a recorded arrest.

```
dbGetQuery(con, "
  WITH clearance AS (
    SELECT STRFTIME('%m', incident_date) AS month,
           ucr_offense_code,
           COUNT(*) AS total,
           SUM(CASE WHEN EXISTS (
                   SELECT 1
                   FROM arrestee
                   WHERE arrestee.ori = offense.ori
                     AND arrestee.incident_number = offense.incident_number)
                 THEN 1
                 ELSE 0
               END) AS arrests
    FROM offense
    WHERE ucr_offense_code IN ('09A','220') -- homicide and burglary
    GROUP BY month, ucr_offense_code)
  SELECT month,
         ucr_offense_code,
         total,
         arrests,
         ROUND(100.0 * arrests / NULLIF(total, 0), 1) AS pct_arrested
  FROM clearance
  ORDER BY ucr_offense_code, month")
```

```
   month ucr_offense_code total arrests pct_arrested
1     01              09A  1019     579         56.8
```

```
2     02              09A   920     529         57.5
3     03              09A   979     568         58.0
4     04              09A  1010     546         54.1
5     05              09A  1042     565         54.2
6     06              09A  1072     570         53.2
7     07              09A  1099     578         52.6
8     08              09A   948     492         51.9
9     09              09A   990     509         51.4
10    10              09A   887     481         54.2
11    11              09A   863     418         48.4
12    12              09A   805     426         52.9
13    01              220 49912    7891         15.8
14    02              220 43957    7070         16.1
15    03              220 48786    8156         16.7
16    04              220 47584    7840         16.5
17    05              220 50861    8301         16.3
18    06              220 48911    7973         16.3
19    07              220 51052    8567         16.8
20    08              220 49965    8391         16.8
21    09              220 48206    7884         16.4
22    10              220 47957    7477         15.6
23    11              220 43591    6586         15.1
24    12              220 40837    5920         14.5
```

The percentage associated with an arrest drops for crimes committed late in the year, not necessarily because those crimes are less likely to be solved, but because there is less time between the crime and the NIBRS submission cutoff for an arrest to be recorded.

The March 2026 submission cutoff makes raw arrest percentages hard to compare. A homicide in January has over a year for an arrest to appear in the data, while a homicide in December has only a few months. To put every crime on equal footing, we can ask a more precise question: was there an arrest *within 60 days* of the offense? Every offense in the year has a full 60-day observation window.

To compute the time between two dates, SQLite provides `JULIANDAY()`, which converts a date to the number of days since a fixed reference point (noon on November 24, 4714 BC, the start of the "Julian day" used by astronomers). Subtracting one Julian day from another gives the number of days between two dates. Let's look at the time from offense to first arrest for a few homicides (UCR code 09A, murder and non-negligent manslaughter). Because the `arrestee` table can hold several arrests per incident, we first collapse it to the earliest arrest date per incident with `MIN(arrest_date)`.

```
dbGetQuery(con, "
  SELECT offense.ori,
         offense.incident_number,
         offense.incident_date,
         arr.first_arrest,
         JULIANDAY(arr.first_arrest) -
           JULIANDAY(offense.incident_date) AS days_to_arrest
  FROM offense
    INNER JOIN (SELECT ori, incident_number, MIN(arrest_date) AS first_arrest
                FROM arrestee
                GROUP BY ori, incident_number) AS arr
      ON offense.ori = arr.ori
         AND offense.incident_number = arr.incident_number
  WHERE  offense.ucr_offense_code = '09A'
  LIMIT 10")
```

```
         ori incident_number incident_date first_arrest days_to_arrest
1  AK0010200    CB0BIYM-728N    2025-10-07   2025-10-08              1
2  AK0010200    CB0BKEMR728N    2025-04-11   2025-04-11              0
3  AK0010200    CB0BRFTQ728N    2025-01-29   2025-02-08             10
4  AK0010200    CB0BROQ9728N    2025-01-15   2025-01-15              0
5  AK0011800    CB0V-MZU728N    2025-12-16   2025-12-16              0
6  AKAST0100    3I24JXYHRVF3    2025-08-05   2025-08-07              2
7  AKAST0100    5S26LDS6DTW5    2025-05-06   2025-05-06              0
8  AKAST0100    9Y2K4IBCUO39    2025-03-27   2025-03-27              0
9  AKAST0100    9Y2K4IY4M739    2025-03-10   2025-03-11              1
10 AKAST0100    AO2UPU7NS71A    2025-01-02   2025-01-03              1
```

Now let's compute the homicide clearance rate by state, where "cleared" means an arrest occurred within 60 days of the offense. We use a `LEFT JOIN` so that homicides with no arrest at all still count in the denominator, and we join `state_lookup` to label the states. Note that `days_to_arrest` is `NULL` whenever there was no arrest, and any comparison with `NULL` (like `NULL <= 60`) is not `TRUE`, so those incidents correctly fall outside the cleared count.

```
dbGetQuery(con, "
  WITH homicide_clearance AS (
    SELECT offense.state,
           JULIANDAY(arr.first_arrest) -
             JULIANDAY(offense.incident_date) AS days_to_arrest
    FROM offense
      LEFT JOIN (SELECT ori, incident_number, MIN(arrest_date) AS first_arrest
```

```
                FROM   arrestee
                GROUP BY ori, incident_number) AS arr
       ON offense.ori = arr.ori
          AND offense.incident_number = arr.incident_number
    WHERE  offense.ucr_offense_code = '09A'
  )
  SELECT state_lookup.state_name,
         COUNT(*) AS homicides,
         SUM(CASE WHEN days_to_arrest <= 60
                  THEN 1 ELSE 0 END) AS cleared_60,
         ROUND(100.0 * SUM(CASE WHEN days_to_arrest <= 60
                                THEN 1 ELSE 0 END) / COUNT(*), 1)
           AS pct_cleared_60
  FROM homicide_clearance
    INNER JOIN state_lookup
      ON homicide_clearance.state = state_lookup.state_code
  GROUP BY state_lookup.state_name
  HAVING homicides >= 50
  ORDER BY pct_cleared_60 DESC")
```

	state_name	homicides	cleared_60	pct_cleared_60
1	South Carolina	313	221	70.6
2	Arkansas	176	122	69.3
3	Minnesota	121	79	65.3
4	Nevada	139	87	62.6
5	Kansas	99	59	59.6
6	Colorado	187	109	58.3
7	Kentucky	110	64	58.2
8	New York	381	218	57.2
9	California	926	530	57.2
10	Arizona	230	131	57.0
11	Washington	197	112	56.9
12	Oregon	118	66	55.9
13	Connecticut	59	33	55.9
14	Utah	72	40	55.6
15	Louisiana	257	138	53.7
16	Oklahoma	196	104	53.1
17	Virginia	342	175	51.2
18	North Carolina	607	303	49.9
19	Pennsylvania	392	194	49.5
20	Alabama	406	197	48.5
21	Michigan	407	196	48.2

```
22                Texas      1348        635          47.1
23        West Virginia        62         29          46.8
24        Massachusetts        77         36          46.8
25             Maryland       311        144          46.3
26                 Ohio       469        215          45.8
27             Missouri       406        184          45.3
28           New Jersey       121         53          43.8
29              Indiana       228         99          43.4
30          Mississippi       158         68          43.0
31            Tennessee       438        186          42.5
32              Georgia       507        205          40.4
33              Florida       446        180          40.4
34            Wisconsin       194         78          40.2
35           New Mexico       163         65          39.9
36             Illinois       490        154          31.4
37 District of Columbia       113         33          29.2
```

The `HAVING` clause is new here. It filters *groups* after aggregation, whereas `WHERE` filters individual rows *before* aggregation. The distinction matters because of when each clause is evaluated. SQL processes a query in stages: it first uses `WHERE` to decide which rows to keep, then `GROUP BY` collapses those rows into groups and computes the aggregates (`COUNT(*)`, `SUM(...)`, and so on), and only then does `HAVING` get to filter based on those computed aggregates. At the moment `WHERE` runs, the group counts do not exist yet, so a condition like `homicides >= 50` could not possibly be evaluated there.

That ordering gives us a simple rule of thumb: if your filter condition involves an aggregate function, it must go in `HAVING` and if it involves only raw column values, it goes in `WHERE`. A condition like `offense.ucr_offense_code = '09A'` looks only at raw column values, so it belongs in `WHERE` (and we put it there, inside the CTE). A condition like `COUNT(*) >= 50` depends on an aggregate, so it must go in `HAVING`. As a bonus, filtering in `WHERE` is also faster whenever it is possible, because it discards rows early, before the database does the work of grouping and aggregating them, so prefer `WHERE` for any condition that does not require an aggregate.

Here we use `HAVING homicides >= 50` to keep only states with at least 50 homicides so that the percentages are not dominated by states with just a handful of cases. The resulting 60-day clearance rates are more comparable across states than raw cleared counts because every homicide had the same 60-day observation window.

> **Exercise**
>
> 10. In which month were the most robberies (UCR code 120) reported? Use `STRFTIME()` to extract the month, then group and order the results.

8.7 Working `WITH` CTEs

Let's use a CTE to find the states with the largest populations not yet covered by NIBRS. We will compute total population by state, split by whether the agency reports to NIBRS, then calculate what percentage of each state's population is covered.

One important SQL gotcha: agencies that do not report to NIBRS have `agency_nibrs_flag = NULL`. In SQL, any comparison with `NULL` (including `NULL != 'A'`) evaluates to `NULL`, not `TRUE`. So the condition `agency_nibrs_flag != 'A'` would silently exclude all non-reporting agencies rather than selecting them. The correct test is `agency_nibrs_flag IS NULL`.

```
dbGetQuery(con, "
  WITH  -- tabulate state population by state x NIBRS reporting
    pop_by_state AS (
      SELECT state_abbreviation,
             agency_nibrs_flag,
             SUM(total_pop) AS pop
      FROM   batch_header
      WHERE  covered_by_ori IS NULL   -- exclude those reporting through other
      GROUP BY state_abbreviation, agency_nibrs_flag)
  SELECT
    state_abbreviation,
    MAX(CASE WHEN agency_nibrs_flag='A' THEN pop ELSE 0 END)
      AS in_nibrs,
    MAX(CASE WHEN agency_nibrs_flag IS NULL THEN pop ELSE 0 END)
      AS not_in_nibrs,
    ROUND(100.0 * MAX(CASE WHEN agency_nibrs_flag='A' THEN pop ELSE 0 END) /
            (MAX(CASE WHEN agency_nibrs_flag='A' THEN pop ELSE 0 END) +
             MAX(CASE WHEN agency_nibrs_flag IS NULL THEN pop ELSE 0 END)), 0)
      AS pct_covered
  FROM pop_by_state
  GROUP BY state_abbreviation
  ORDER BY pct_covered")
```

```
   state_abbreviation in_nibrs not_in_nibrs pct_covered
1                  PR        0      3184835           0
2                  VI        0        83265           0
3                  PA  7022882      6036550          54
4                  AK   447272       289998          61
5                  FL 16650995      6811523          71
6                  NY 14762833      5239594          74
7                  CA 31563850      7791459          80
```

8	NJ	7787992	1760223	82
9	LA	3978197	639992	86
10	MS	2549881	404279	86
11	IN	6354577	618756	91
12	WY	543258	45495	92
13	AZ	7110506	513312	93
14	GA	10854799	447949	96
15	IL	12179447	539694	96
16	NM	2030529	94969	96
17	NC	11052470	145498	99
18	OH	11828068	72442	99
19	WI	5897461	75326	99
20	AL	5169886	23202	100
21	AR	3111216	3575	100
22	CO	6009256	3305	100
23	CT	3688496	0	100
24	DC	693645	0	100
25	DE	1059952	0	100
26	GM	153836	0	100
27	HI	1432820	0	100
28	IA	3236295	2092	100
29	ID	2029733	0	100
30	KS	2977220	0	100
31	KY	4606864	0	100
32	MA	7147638	6446	100
33	MD	6265347	0	100
34	ME	1414874	0	100
35	MI	10119869	8015	100
36	MN	5826591	3814	100
37	MO	6245658	24883	100
38	MT	1144694	0	100
39	NB	2016524	1482	100
40	ND	799358	0	100
41	NH	1415342	0	100
42	NV	3276834	5354	100
43	OK	4123288	0	100
44	OR	4262770	10816	100
45	RI	1113108	1413	100
46	SC	5570274	0	100
47	SD	935094	0	100
48	TN	7315076	0	100
49	TX	31596690	113131	100
50	UT	3535614	3290	100

```
51                VA  8880107            0         100
52                VT   644663            0         100
53                WA  8001020            0         100
54                WV  1765814          333         100
```

The states with the lowest NIBRS coverage tend to include Florida, Pennsylvania, California, and New York, all very populous states. This is an important limitation to keep in mind when interpreting national statistics from NIBRS.

You can use multiple CTEs in one `WITH` clause, separating them with commas. The query below uses two CTEs to identify the agency with the most shootings (aggravated assault, UCR 13A, completed, firearm involved), then joins in the agency name from the batch header.

```
dbGetQuery(con, "
  WITH  -- tabulate shootings by ORI
    shootings AS (
      SELECT ori, COUNT(*) AS shooting_count
      FROM   offense
      WHERE  ucr_offense_code = '13A'
        AND  offense_attempted_completed = 'C'
        AND  (weapon_force_1 IN ('11','12','13','14','15') OR
              weapon_force_2 IN ('11','12','13','14','15') OR
              weapon_force_3 IN ('11','12','13','14','15'))
      GROUP BY ori
    ),
    top_agency AS ( -- which ORI reports the most shootings?
      SELECT ori, shooting_count
      FROM   shootings
      ORDER BY shooting_count DESC
      LIMIT 1
    )
  SELECT batch_header.city_name,
         batch_header.state_abbreviation,
         top_agency.shooting_count
  FROM   top_agency
    INNER JOIN batch_header
      ON top_agency.ori = batch_header.ori")
```

```
  city_name state_abbreviation shooting_count
1   HOUSTON                TX           4540
```

> **Exercise**
>
> 11. What is the race distribution of shooting victims? Use a CTE to identify shooting incidents, join to `victim`, and search all ten victim UCR offense-code fields to confirm that each victim was connected to aggravated assault (13A).

8.8 Window functions

So far we have not really worked with the `victim` table much. Let's peek at the first few rows.

```
dbGetQuery(con, "
  SELECT *
  FROM   victim
  LIMIT  5")
```

```
  segment state       ori incident_number incident_date victim_sequence_number
1      04    50 AK0010200    C30BRCZ9728N      20250301                      1
2      04    50 AK0010200    C30BREBQ728N      20250601                      1
3      04    50 AK0010200    C30BRERH728N      20250106                      1
4      04    50 AK0010200    C30BRETM728N      20251128                      1
5      04    50 AK0010200    C30BRFTC728N      20250618                      1
  ucr_offense_code_1 ucr_offense_code_2 ucr_offense_code_3 ucr_offense_code_4
1                13C               <NA>               <NA>               <NA>
2                23G               <NA>               <NA>               <NA>
3                26F               <NA>               <NA>               <NA>
4                23D               <NA>               <NA>               <NA>
5                23G               <NA>               <NA>               <NA>
  ucr_offense_code_5 ucr_offense_code_6 ucr_offense_code_7 ucr_offense_code_8
1               <NA>               <NA>               <NA>               <NA>
2               <NA>               <NA>               <NA>               <NA>
3               <NA>               <NA>               <NA>               <NA>
4               <NA>               <NA>               <NA>               <NA>
5               <NA>               <NA>               <NA>               <NA>
  ucr_offense_code_9 ucr_offense_code_10 type_of_victim age_of_victim
1               <NA>                <NA>              I            25
2               <NA>                <NA>              I            59
3               <NA>                <NA>              I            54
4               <NA>                <NA>              I            39
5               <NA>                <NA>              B          <NA>
  sex_of_victim race_of_victim ethnicity_of_victim resident_status_of_victim
```

```
1             F              W                   N                          R
2             M              W                   N                          R
3             F              W                   N                          N
4             M              W                   N                          R
5          <NA>           <NA>                <NA>                       <NA>
  circumstance_1 circumstance_2 additional_justifiable_homicide_circumstances
1           <NA>           <NA>                                           <NA>
2           <NA>           <NA>                                           <NA>
3           <NA>           <NA>                                           <NA>
4           <NA>           <NA>                                           <NA>
5           <NA>           <NA>                                           <NA>
  injury_1 injury_2 injury_3 injury_4 injury_5 offender_number_to_be_related
1     <NA>     <NA>     <NA>     <NA>     <NA>                           01
2     <NA>     <NA>     <NA>     <NA>     <NA>                         <NA>
3     <NA>     <NA>     <NA>     <NA>     <NA>                           01
4     <NA>     <NA>     <NA>     <NA>     <NA>                           01
5     <NA>     <NA>     <NA>     <NA>     <NA>                         <NA>
  relationship_of_victim_to_offender relationship_2 relationship_3
1                                 OK           <NA>           <NA>
2                               <NA>           <NA>           <NA>
3                                 SB           <NA>           <NA>
4                                 AQ           <NA>           <NA>
5                               <NA>           <NA>           <NA>
  relationship_4 relationship_5 relationship_6 relationship_7 relationship_8
1           <NA>           <NA>           <NA>           <NA>           <NA>
2           <NA>           <NA>           <NA>           <NA>           <NA>
3           <NA>           <NA>           <NA>           <NA>           <NA>
4           <NA>           <NA>           <NA>           <NA>           <NA>
5           <NA>           <NA>           <NA>           <NA>           <NA>
  relationship_9 relationship_10 type_of_activity_officer_circumstance
1           <NA>            <NA>                                <NA>
2           <NA>            <NA>                                <NA>
3           <NA>            <NA>                                <NA>
4           <NA>            <NA>                                <NA>
5           <NA>            <NA>                                <NA>
  assignment_type_officer ori_other_jurisdiction_officer
1                    <NA>                          <NA>
2                    <NA>                          <NA>
3                    <NA>                          <NA>
4                    <NA>                          <NA>
5                    <NA>                          <NA>
```

Each row is one victim connected to one incident. `type_of_victim` distinguishes an individual

person (`I`) from a business (`B`), government (`G`), society (`S`), or other victim types. Many NIBRS offenses (shoplifting, vandalism, fraud, drug offenses) have a business or society as the "victim" rather than a person.

Code	Type of victim
I	Individual
B	Business
F	Financial Institution
G	Government
R	Religious Organization
S	Society/Public
L	Law Enforcement Officer
O	Other
U	Unknown

For individual victims, NIBRS records `age_of_victim`, `sex_of_victim` (`M`/`F`), `race_of_victim`, and `ethnicity_of_victim` as separate fields. The `ucr_offense_code_1` (through `_10`) columns record which offenses in the incident this particular victim was connected to, since a single incident can have several offenses and several victims who are not all tied to every offense.

A natural question is how victim demographics differ across crime types. We already know how to count victims by sex within each crime using `GROUP BY`. But we often want each count expressed as a *percentage of that crime's victims*, and that requires dividing each group's count by the total for its crime. With the tools we have so far, that means computing the per-crime totals in one query and joining them back. A **window function** does it in a single step.

A window function computes an aggregate across a set of related rows but, unlike `GROUP BY`, it does not collapse those rows. Instead it attaches the computed value to every row. You write an ordinary aggregate followed by an `OVER` clause that defines the "window" of rows to aggregate over. The simplest window is `OVER ()`, meaning "all rows in the result." Here we compute each race's share of all individual victims. We use a CTE to get the per-race counts first, then a window function to divide by the grand total.

```
dbGetQuery(con, "
  WITH victim_race AS (
    SELECT race_of_victim,
           COUNT(*) AS n
    FROM   victim
    WHERE  type_of_victim = 'I'      -- individuals only
    GROUP BY race_of_victim)
  SELECT race_of_victim,
```

```
        n,
        ROUND(100.0 * n / SUM(n) OVER (), 1) AS pct_of_victims
  FROM   victim_race
  ORDER BY n DESC")
```

```
  race_of_victim       n pct_of_victims
1              W 5602229           62.5
2              B 2414717           26.9
3              U  525152            5.9
4              A  310335            3.5
5              I   85660            1.0
6              P   29015            0.3
```

`SUM(n) OVER ()` adds up `n` across every row of the result and reports that single grand total alongside each row, so dividing gives each race's percentage. Without window functions we would have needed a second query (or a self-join) just to get that total. Normally you would see the aggregate function `SUM()` in the `SELECT` clause and automatically start thinking that you need a `GROUP BY` to make this work. Here, however, we do not want the query to collapse across any group. The `OVER ()` clause communicates how we want `SUM()` to aggregate, and it simply attaches that total to every row.

The real power comes from `PARTITION BY`, which splits the rows into groups and restarts the window aggregate within each group, much like `GROUP BY` but without collapsing the rows. The query below breaks down victim sex *within* each crime type. The `PARTITION BY offense_lookup.crime` makes `SUM(COUNT(*))` total only the victims of that one crime, so the percentages sum to 100 within each crime. By searching all ten offense-code fields with `IN`, the query counts each victim–crime association. A victim associated with multiple crimes will, therefore, contribute once to the count for each crime.

```
a <- dbGetQuery(con, "
  SELECT offense_lookup.crime,
         victim.sex_of_victim AS sex,
         COUNT(*) AS n,
         ROUND(100.0 * COUNT(*) /
               SUM(COUNT(*)) OVER (PARTITION BY offense_lookup.crime), 1)
           AS pct_within_crime
  FROM victim
    INNER JOIN offense_lookup
      ON offense_lookup.ucr_code IN (
        victim.ucr_offense_code_1, victim.ucr_offense_code_2,
        victim.ucr_offense_code_3, victim.ucr_offense_code_4,
        victim.ucr_offense_code_5, victim.ucr_offense_code_6,
```

```
       victim.ucr_offense_code_7, victim.ucr_offense_code_8,
       victim.ucr_offense_code_9, victim.ucr_offense_code_10)
  WHERE  victim.type_of_victim = 'I'
    AND  victim.sex_of_victim IN ('M','F')
  GROUP BY offense_lookup.crime, victim.sex_of_victim
  ORDER BY offense_lookup.crime, victim.sex_of_victim")
a
```

	crime	sex	n	pct_within_crime
1	Aggravated Assault	F	326239	47.2
2	Aggravated Assault	M	365665	52.8
3	All Other Larceny	F	507409	47.3
4	All Other Larceny	M	566174	52.7
5	Arson	F	8984	45.0
6	Arson	M	10975	55.0
7	Bribery	F	315	48.5
8	Bribery	M	335	51.5
9	Burglary/Breaking & Entering	F	217838	45.8
10	Burglary/Breaking & Entering	M	257539	54.2
11	Counterfeiting/Forgery	F	30796	47.3
12	Counterfeiting/Forgery	M	34378	52.7
13	Credit Card/Automated Teller Machine Fraud	F	90037	55.6
14	Credit Card/Automated Teller Machine Fraud	M	71916	44.4
15	Destruction/Damage/Vandalism of Property	F	581896	51.2
16	Destruction/Damage/Vandalism of Property	M	555205	48.8
17	Embezzlement	F	5271	46.4
18	Embezzlement	M	6086	53.6
19	Extortion/Blackmail	F	7813	26.7
20	Extortion/Blackmail	M	21486	73.3
21	False Pretenses/Swindle/Confidence Game	F	152807	52.7
22	False Pretenses/Swindle/Confidence Game	M	136918	47.3
23	Fondling	F	72022	81.3
24	Fondling	M	16594	18.7
25	Hacking/Computer Invasion	F	4827	56.1
26	Hacking/Computer Invasion	M	3781	43.9
27	Human Trafficking, Commercial Sex Acts	F	2604	88.4
28	Human Trafficking, Commercial Sex Acts	M	341	11.6
29	Human Trafficking, Involuntary Servitude	F	532	78.9
30	Human Trafficking, Involuntary Servitude	M	142	21.1
31	Identity Theft	F	100720	51.0
32	Identity Theft	M	96813	49.0
33	Impersonation	F	34945	50.8

```
34                                Impersonation   M   33903            49.2
35                                      Incest   F     965            75.7
36                                      Incest   M     309            24.3
37                                 Intimidation   F  439125            60.7
38                                 Intimidation   M  284158            39.3
39                         Justifiable Homicide   F      20             3.4
40                         Justifiable Homicide   M     564            96.6
41                         Kidnapping/Abduction   F   42107            78.1
42                         Kidnapping/Abduction   M   11805            21.9
43                          Motor Vehicle Theft   F  223629            39.7
44                          Motor Vehicle Theft   M  340018            60.3
45          Murder & Non-negligent Manslaughter   F    3024            24.2
46          Murder & Non-negligent Manslaughter   M    9451            75.8
47                       Negligent Manslaughter   F     759            35.8
48                       Negligent Manslaughter   M    1362            64.2
49                               Pocket-picking   F   12462            56.0
50                               Pocket-picking   M    9790            44.0
51                              Purse-snatching   F    7461            74.0
52                              Purse-snatching   M    2628            26.0
53                                         Rape   F   83979            93.0
54                                         Rape   M    6368             7.0
55                                      Robbery   F   53404            32.9
56                                      Robbery   M  108873            67.1
57                Sexual Assault With An Object   F    5528            86.7
58                Sexual Assault With An Object   M     848            13.3
59                                   Shoplifting   F   16396            45.0
60                                   Shoplifting   M   20079            55.0
61                                Simple Assault   F 1288705            58.2
62                                Simple Assault   M  925448            41.8
63                                        Sodomy   F    8804            58.1
64                                        Sodomy   M    6359            41.9
65                                Statutory Rape   F    6955            84.9
66                                Statutory Rape   M    1237            15.1
67                     Stolen Property Offenses   F   32655            38.5
68                     Stolen Property Offenses   M   52066            61.5
69                          Theft From Building   F  122858            53.3
70                          Theft From Building   M  107703            46.7
71  Theft From Coin-Operated Machine or Device   F     572            37.5
72  Theft From Coin-Operated Machine or Device   M     953            62.5
73                     Theft From Motor Vehicle   F  304777            41.7
74                     Theft From Motor Vehicle   M  425742            58.3
75 Theft of Motor Vehicle Parts or Accessories   F   70045            41.1
76 Theft of Motor Vehicle Parts or Accessories   M  100312            58.9
```

```
77                                Welfare Fraud   F    3235            72.4
78                                Welfare Fraud   M    1235            27.6
79                                   Wire Fraud   F   25587            54.7
80                                   Wire Fraud   M   21215            45.3
```

Notice the nested `SUM(COUNT(*))`: the inner `COUNT(*)` is the ordinary aggregate produced by the `GROUP BY` (the number of victims of a given crime and sex), and the outer `SUM(...) OVER (PARTITION BY ...)` adds those group counts together within each crime to give the crime's total.

Standard SQL has no built-in "pivot" option, but we can do that in R to get a different view of these results.

```
library(tidyr)
a |>
   pivot_wider(id_cols = crime,
               names_from = sex,
               values_from = pct_within_crime) |>
   arrange(desc(F)) |>
   # first 6 and last 6
   slice(1:6, (n()-5):n())
```

```
# A tibble: 12 x 3
   crime                                        F     M
   <chr>                                    <dbl> <dbl>
 1 Rape                                      93     7
 2 Human Trafficking, Commercial Sex Acts    88.4  11.6
 3 Sexual Assault With An Object             86.7  13.3
 4 Statutory Rape                            84.9  15.1
 5 Fondling                                  81.3  18.7
 6 Human Trafficking, Involuntary Servitude  78.9  21.1
 7 Theft From Coin-Operated Machine or Device 37.5 62.5
 8 Negligent Manslaughter                    35.8  64.2
 9 Robbery                                   32.9  67.1
10 Extortion/Blackmail                       26.7  73.3
11 Murder & Non-negligent Manslaughter       24.2  75.8
12 Justifiable Homicide                       3.4  96.6
```

The results reveal stark differences in who is victimized by different crimes, with offenses such as forcible rape skewing heavily female while robbery skews male.

> **Exercise**
>
> 12. Which 10 law enforcement agencies account for the largest shares of all arrest records? Count arrest records by ORI, then use a window function to calculate each agency's percentage of the national total.

When you are done and ready to shutdown, be sure to disconnect from your database.

```
dbDisconnect(con)
```

9 Answers to the exercises

1. Using the `victim` table, show the ORI, incident number, age, and sex for the first 10 individual victims. Individual victims have `type_of_victim = 'I'`.

```
dbGetQuery(con, "
  SELECT ori, incident_number, age_of_victim, sex_of_victim
  FROM   victim
  WHERE  type_of_victim = 'I'
  LIMIT  10")
```

```
          ori incident_number age_of_victim sex_of_victim
1  AK0010200    C30BRCZ9728N            25             F
2  AK0010200    C30BREBQ728N            59             M
3  AK0010200    C30BRERH728N            54             F
4  AK0010200    C30BRETM728N            39             M
5  AK0010200    C30BRO3-728N            11             M
6  AK0010200    C30BRVBM728N            55             M
7  AK0010200    C30BRVZH728N            43             F
8  AK0010200    C30BRVZR728N            38             F
9  AK0010200    C30BRY5D728N            36             F
10 AK0010200    CB0BAA3C728N            45             F
```

2. Using the `arrestee` table, show the incident number, arrest date, age, and arrest offense code for the first 10 arrest records from the Philadelphia Police Department (ORI PAPEP0000).

```
dbGetQuery(con, "
  SELECT incident_number, arrest_date, age_of_arrestee,
         ucr_arrest_offense_code
  FROM   arrestee
  WHERE  ori = 'PAPEP0000'
  LIMIT  10")
```

```
   incident_number arrest_date age_of_arrestee ucr_arrest_offense_code
1     2H-050726624  2025-02-06              15                      23C
2     2H-050726624  2025-02-06              13                      23C
3     2H-05072666W  2025-02-06              33                      520
4     2H-05072670I  2025-02-06              20                      26A
5     2H-0507267XW  2025-02-06              35                      26A
6     2H-050726C8I  2025-02-11              60                      240
7     2H-050726CKR  2025-02-08              26                      520
8     2H-050726G04  2025-02-05              44                      200
9     2H-050726L2D  2025-02-08              45                      26A
10    2H-050726LJI  2025-02-08              42                      520
```

3. How many completed robbery offense records (UCR code 120) were reported to the Philadelphia Police Department (ORI PAPEP0000)?

```
dbGetQuery(con, "
  SELECT COUNT(*) AS offense_count
  FROM   offense
  WHERE  ori = 'PAPEP0000'
    AND  ucr_offense_code = '120'
    AND  offense_attempted_completed = 'C'")
```

```
  offense_count
1          3176
```

4. How many distinct agencies reported at least one offense in Pennsylvania? Use `REGEXP` to identify ORIs beginning with PA.

```
dbGetQuery(con, "
  SELECT COUNT(DISTINCT ori) AS pa_agency_count
  FROM   offense
  WHERE  ori REGEXP '^PA'")
```

```
  pa_agency_count
1             228
```

5. How many offense records meet our definition of a shooting: aggravated assault (13A), completed (C), with firearm code 11, 12, 13, 14, or 15 in any weapon_force column?

```
dbGetQuery(con, "
  SELECT COUNT(*) AS shooting_count
  FROM   offense
  WHERE  ucr_offense_code = '13A'
    AND  offense_attempted_completed = 'C'
    AND  (weapon_force_1 IN ('11','12','13','14','15') OR
          weapon_force_2 IN ('11','12','13','14','15') OR
          weapon_force_3 IN ('11','12','13','14','15'))")
```

```
  shooting_count
1         155083
```

6. Which 10 states have the largest number of fully reporting agencies? Use the `batch_header` table and the `reported_all_12_months` column.

```
dbGetQuery(con, "
  SELECT state_abbreviation,
         COUNT(*) AS fully_reporting_agencies
  FROM   batch_header
  WHERE  reported_all_12_months = 1
  GROUP BY state_abbreviation
  ORDER BY fully_reporting_agencies DESC
  LIMIT 10")
```

```
   state_abbreviation fully_reporting_agencies
1                  TX                     1498
2                  CA                      712
3                  IL                      600
4                  MI                      564
5                  OK                      446
6                  OH                      426
7                  MO                      424
8                  MN                      372
9                  KY                      361
10                 VA                      351
```

7. What are the five most common location types for robbery (UCR code 120)? Join `offense` to `location_lookup` and report the number of robbery offense records for each location type.

```
dbGetQuery(con, "
  SELECT location_lookup.location_type,
         COUNT(*) AS robbery_count
  FROM   offense
    INNER JOIN location_lookup
      ON offense.location_type = location_lookup.location_code
  WHERE  offense.ucr_offense_code = '120'
  GROUP BY location_lookup.location_type
  ORDER BY robbery_count DESC
  LIMIT 5")
```

```
       location_type robbery_count
1 Highway/Road/Alley         47225
2     Residence/Home         27404
3 Parking Lot/Garage         12387
4  Convenience Store          9177
5      Other/Unknown          6169
```

8. How many cases are recorded as justifiable homicides? Check both the `offense` table (UCR code 09C) and the `victim` table (circumstance codes 20 and 21). Use `circumstance_lookup` to report the circumstance descriptions.

Using the offense table:

```
dbGetQuery(con, "
  SELECT COUNT(*) AS justifiable_homicide_count
  FROM   offense
  WHERE  ucr_offense_code = '09C'")
```

```
  justifiable_homicide_count
1                        574
```

Using the victim table, joining `circumstance_lookup` to replace both circumstance-code fields with descriptions:

```
dbGetQuery(con, "
  SELECT circumstance_lookup.circumstance,
         COUNT(*) AS victim_count
  FROM   victim
    INNER JOIN circumstance_lookup
      ON circumstance_lookup.circumstance_code IN (
```

```
          victim.circumstance_1, victim.circumstance_2)
  WHERE  circumstance_lookup.circumstance_code IN ('20','21')
  GROUP BY circumstance_lookup.circumstance")
```

```
                       circumstance victim_count
1  Criminal Killed by Police Officer          207
2 Criminal Killed by Private Citizen          378
```

9. Among completed motor vehicle theft offense records (UCR code 240), how many have an incident with at least one arrest record, and what percentage is that? Use `EXISTS()`.

```
dbGetQuery(con, "
  WITH theft_counts AS (
    SELECT COUNT(*) AS total_thefts,
           SUM(CASE WHEN EXISTS (
                 SELECT 1
                 FROM arrestee
                 WHERE arrestee.ori = offense.ori
                   AND arrestee.incident_number = offense.incident_number)
               THEN 1 ELSE 0 END) AS thefts_with_arrest
    FROM offense
    WHERE ucr_offense_code = '240'
      AND offense_attempted_completed = 'C'
  )
  SELECT total_thefts,
         thefts_with_arrest,
         ROUND(100.0 * thefts_with_arrest /
               NULLIF(total_thefts, 0), 1) AS pct_with_arrest
  FROM theft_counts")
```

```
  total_thefts thefts_with_arrest pct_with_arrest
1       536705              49309             9.2
```

10. In which month were the most robberies (UCR code 120) reported? Use `STRFTIME()` to extract the month, then group and order the results.

```
dbGetQuery(con, "
  SELECT STRFTIME('%m', incident_date) AS month,
         COUNT(*) AS robbery_count
  FROM   offense
  WHERE  ucr_offense_code = '120'
```

```
  GROUP BY month
  ORDER BY robbery_count DESC
  LIMIT 1")
```

```
  month robbery_count
1    05         13589
```

11. What is the race distribution of shooting victims? Use a CTE to identify shooting incidents, join to `victim`, and search all ten victim UCR offense-code fields to confirm that each victim was connected to aggravated assault (13A).

```
dbGetQuery(con, "
  WITH shooting_incidents AS (
    SELECT DISTINCT ori, incident_number
    FROM   offense
    WHERE  ucr_offense_code = '13A'
      AND  offense_attempted_completed = 'C'
      AND  (weapon_force_1 IN ('11','12','13','14','15') OR
            weapon_force_2 IN ('11','12','13','14','15') OR
            weapon_force_3 IN ('11','12','13','14','15'))
  )
  SELECT victim.race_of_victim,
         COUNT(*) AS victim_count
  FROM   shooting_incidents
    INNER JOIN victim
      ON shooting_incidents.ori = victim.ori
         AND shooting_incidents.incident_number = victim.incident_number
  WHERE '13A' IN (
    victim.ucr_offense_code_1, victim.ucr_offense_code_2,
    victim.ucr_offense_code_3, victim.ucr_offense_code_4,
    victim.ucr_offense_code_5, victim.ucr_offense_code_6,
    victim.ucr_offense_code_7, victim.ucr_offense_code_8,
    victim.ucr_offense_code_9, victim.ucr_offense_code_10)
  GROUP BY victim.race_of_victim
  ORDER BY victim_count DESC")
```

```
  race_of_victim victim_count
1              B       109357
2              W       104075
3              U         7538
4              A         2769
```

```
5              I         1682
6              P          500
```

12. Which 10 law enforcement agencies account for the largest shares of all arrest records? Count arrest records by ORI, then use a window function to calculate each agency's percentage of the national total.

```
dbGetQuery(con, "
  WITH agency_arrests AS (
    SELECT ori,
           COUNT(*) AS arrests
    FROM   arrestee
    GROUP BY ori
  )
  SELECT ori,
         arrests,
         ROUND(100.0 * arrests / SUM(arrests) OVER (), 2)
           AS pct_of_all_arrests
  FROM   agency_arrests
  ORDER BY arrests DESC
  LIMIT 10")
```

```
         ori arrests pct_of_all_arrests
1  NY0303000  203244               5.64
2  CA0194200   35200               0.98
3  NV0020100   34870               0.97
4  ILCPD0000   27563               0.76
5  PAPEP0000   25596               0.71
6  TXHPD0000   24065               0.67
7  TXSPD0000   19099               0.53
8  TXDPD0000   16960               0.47
9  FL0160000   16687               0.46
10 CODPD0000   16028               0.44
```

10 Creating `nibrs_format.csv`

The following code was used to create `nibrs_format.csv` from the Excel documentation file. It is included here for reference and does not need to be run.

```r
library(readxl)
library(tidyr)

fmtXL <- read_excel("NIBRS Records Description updated.xlsx",
                    sheet = "INCIDENT RECORD",
                    range = "A5:D819") |>
   rename(DataField=`Data Field Number`,
          TypeLength=`Type/ Length`)

# Helper to pull format rows for one segment and clean them up
clean_fmt <- function(rows, exclude_pos=NULL) {
  fmt <- fmtXL |>
    slice(rows) |>
    filter(!is.na(Position))
  if (!is.null(exclude_pos))
    fmt <- fmt |> filter(!(Position %in% exclude_pos))
  data.frame(
    col_name  = fmt$Description |>
      strsplit(split=" - ", fixed=TRUE) |>
      sapply(head, n=1) |>
      gsub("[^A-Za-z0-9]+", "_", x=_) |>
      gsub("_+$", "", x=_) |>
      tolower(),
    col_type  = fmt$TypeLength |>
      substring(1,1) |>
      recode_values(from=c("A","N"), to=c("c","n")),
    col_width = fmt$TypeLength |>
      substring(2) |>
      as.numeric()
  )
}

# Locate the start of each segment in fmtXL
i02 <- grep('LEVEL "02"', fmtXL$DataField)
i03 <- grep('LEVEL "03"', fmtXL$DataField)
i04 <- grep('LEVEL "04"', fmtXL$DataField)
i05 <- grep('LEVEL "05"', fmtXL$DataField)
i06 <- grep('LEVEL "06"', fmtXL$DataField)
i07 <- grep('LEVEL "07"', fmtXL$DataField)

nibrs_format <- bind_rows(
  clean_fmt(3:(i02-3),    "59-88")                                  |> mutate(segment="01"),
```

```
  clean_fmt((i02+1):(i03-3), c("38-40","46-48","49-57"))       |> mutate(segment="02"),
  clean_fmt((i03+2):(i04-3), c("14-22","20-22","58-102","58-72","103-132")) |> mutate(segment
  clean_fmt((i04+1):(i05-3), c("37-66","74-77","79-83","84-123")) |> mutate(segment="04"),
  clean_fmt((i05+1):(i06-3))                                   |> mutate(segment="05"),
  clean_fmt((i06+2):(i07-3), c("61-66","75-104"))              |> mutate(segment="06"),
  clean_fmt(-(1:i07),     "44-49")                             |> mutate(segment="07")
)

# The property segment has two columns both named ESTIMATED.QUANTITY;
# the second represents thousandths of the first, so rename it.
nibrs_format <- nibrs_format |>
  mutate(col_name = if_else(
    segment=="03" &
    col_name=="estimated_quantity" &
    duplicated(paste(segment, col_name)),
    "estimated_quantity_1000ths",
    col_name))

# Shorten the unwieldy type_property_loss_etc name
nibrs_format <- nibrs_format |>
  mutate(col_name = if_else(col_name=="type_property_loss_etc",
                            "type_property_loss",
                            col_name))

# Batch header lives on a different sheet
fmtBH <- read_excel("NIBRS Records Description updated.xlsx", skip=4) |>
  data.frame() |>
  filter(!is.na(Position) &
         !(Position %in% c("106-225","234-269","234","235","236","270-284")))

nibrs_format <- bind_rows(
  nibrs_format,
  data.frame(
    col_name  = fmtBH$Description |>
      strsplit(split=" - ", fixed=TRUE) |>
      sapply(head, n=1) |>
      gsub("[^A-Za-z0-9]+", "_", x=_) |>
      gsub("_+$", "", x=_) |>
      tolower(),
    col_type  = fmtBH$Type..Length |>
      substring(1,1) |>
      recode_values(from=c("A","N"), to=c("c","n")),
```

```
    col_width = fmtBH$Type..Length |>
      substring(2) |>
      as.numeric(),
    segment   = "BH"
  )
)

write_csv(nibrs_format, "NIBRS/nibrs_format.csv")
```