databases & dplyr

Lecture 20

Dr. Colin Rundel

The why of databases

Numbers every programmer should know

Task	Timing (ns)	Timing (µs)
L1 cache reference	0.5	0.0005
L2 cache reference	7	0.007
Main memory reference	100	0.1
Random seek SSD	150,000	150
Read 1 MB sequentially from memory	250,000	250
Read 1 MB sequentially from SSD	1,000,000	1,000
Disk seek	10,000,000	10,000
Read 1 MB sequentially from disk	20,000,000	20,000
Send packet CA->Netherlands->CA	150,000,000	150,000

From jboner/latency.txt & sirupsen/napkin-math

Implications for big data

Let's imagine we have a *10 GB* flat data file and that we want to select certain rows based on a particular criterion (filtering).

This requires a sequential read across the entire data set.

File Location	Performance	Time
in memory	$10\ GB \times (250\ \mu s/1\ MB)$	2.5 seconds
on disk (SSD)	$10\ GB \times (1\ ms/1\ MB)$	10 seconds
on disk (HD)	$10\ GB \times (20\ ms/1\ MB)$	200 seconds

This is just for *reading* the sequential data, if we make any modifications (*writing*) or the data is fragmented things are much worse.

Blocks

Cost:

Disk << SSD <<< Memory

Speed:

Disk <<< SSD << Memory

Disk is cheap but slow; memory is fast but limited.

What do we do when our data doesn't fit in memory?

Create *blocks* — group related rows and read multiple rows at a time, loading only what's needed (skip what we dont want).

Optimal block size depends on the task and the storage medium.

Linear vs Binary Search

Blocks reduce I/O, but any query still requires scanning all of them — a linear search requiring $\square(N)$ reads.

We can do better if we are careful about how we structure our data, specifically sorting some (or all) of the columns.

- Sorting is expensive, $\square(N \log N)$, but it only needs to be done once.
- After sorting, we can use a binary search for any subsetting tasks, $\square(\log N)$
- In databases these "sorted" columns are referred to as *indexes*.
- Indexes require additional storage, but are usually small enough to be kept in memory even when the blocks need to stay on disk.

So why databases?

Implementing blocks, indexes, and query optimization correctly is hard:

- The optimal strategy depends on your queries — and they change
- Data is modified (inserts, updates, deletes) — structures must stay consistent
- Multiple users may access data simultaneously — concurrency must be managed
- Failures happen — data must remain correct and recoverable

A database management system (DBMS) handles all of this for you, plus provides a standard query language (SQL) to express what you want, not how to get it.

Databases

SQLite

SQLite is a lightweight, serverless, relational database engine:

- Self-contained — the entire database lives in a single file (or in memory)
- No separate server process — the library runs inside your application
- Accessible directly via a CLI (`sqlite3`) or through language bindings (R, Python, etc.)
- Full SQL support — joins, indexes, transactions, views, and more
- Ubiquitous — embedded in browsers, phones, operating systems, and countless apps — likely the most widely deployed database engine in the world

It is not designed for high-concurrency or multi-user write workloads, but is an excellent tool for learning SQL and for single-user / embedded use cases.

R & databases - DBI

The DBI package provides a consistent R interface to most relational database. Rather than learning a different API for each database, DBI gives you one set of functions for:

- connecting/disconnecting
- creating and executing statements
- extracting results
- error/exception handling
- reading database metadata
- transaction management

RSQLite

RSQLite provides the backend needed to use DBI with SQLite databases. Postgres, MySQL, DuckDB, and many others follow similar DBI patterns.

```
1 library(RSQLite)
```

`RSQLite` re-exports DBI functions so we do not need to load `DBI` as well.

Once loaded, create a connection to your database:

```
1 con = dbConnect(RSQLite::SQLite(), "employees.sqlite")
2 str(con)
```

```
Formal class 'SQLiteConnection' [package "RSQLite"] with 8 slots
  ..@ ptr                :<externalptr>
  ..@ dbname             : chr "employees.sqlite"
  ..@ loadable.extensions: logi TRUE
  ..@ flags              : int 70
  ..@ vfs                : chr ""
  ..@ ref                :<environment: 0x10a38e720>
  ..@ bigint             : chr "integer64"
  ..@ extended_types     : logi FALSE
```

Example Table

Let's walk through DBI operations using a simple employee table.

```
employees = tibble(
  name   = c("Alice","Bob","Carol","Dave","Eve","Frank"),
  email  = c("alice@company.com", "bob@company.com",
             "carol@company.com", "dave@company.com",
             "eve@company.com",   "frank@company.com"),
  salary = c(52000, 40000, 30000, 33000, 44000, 37000),
  dept   = c("Accounting", "Accounting","Sales",
             "Accounting","Sales","Sales"),
)
```

```
# Show current tables
dbListTables(con)
```

```
character(0)
```

```
# Create a new table and populate it
dbWriteTable(con, name = "employees", value = employees)
dbListTables(con)
```

```
[1] "employees"
```

Removing Tables

```
# Create employs
dbWriteTable(con, "employs", employees)
dbListTables(con)
```

```
[1] "employees" "employs"
```

```
# Delete employs
dbRemoveTable(con,"employs")
dbListTables(con)
```

```
[1] "employees"
```

Querying Tables

Database queries are transactional (see ACID) and follow the steps:

```
1 (res = dbSendQuery(con, "SELECT * FROM employees"))
```

```
<SQLiteResult>
  SQL  SELECT * FROM employees
  ROWS Fetched: 0 [incomplete]
       Changed: 0
```

```
1 dbFetch(res)
```

```
   name             email salary       dept
1 Alice alice@company.com  52000 Accounting
2   Bob   bob@company.com  40000 Accounting
3 Carol carol@company.com  30000      Sales
4  Dave  dave@company.com  33000 Accounting
5   Eve   eve@company.com  44000      Sales
6 Frank frank@company.com  37000      Sales
```

```
1 dbHasCompleted(res)
```

```
[1] TRUE
```

```
1 dbClearResult(res)
```

For convenience

`dbGetQuery()` combines all the preceeding steps:

```
1 (res = dbGetQuery(con, "SELECT * FROM employees"))
```

```
   name             email salary       dept
1 Alice alice@company.com  52000 Accounting
2   Bob   bob@company.com  40000 Accounting
3 Carol carol@company.com  30000      Sales
4  Dave  dave@company.com  33000 Accounting
5   Eve   eve@company.com  44000      Sales
6 Frank frank@company.com  37000      Sales
```

Creating tables

`dbCreateTable()` will create a new table with a schema based on an existing data.frame / tibble, but it does not populate that table with data.

```
1 dbCreateTable(con, "iris", iris)
2 (res = dbGetQuery(con, "select * from iris"))
```

```
[1] Sepal.Length Sepal.Width  Petal.Length Petal.Width  Species
<0 rows> (or 0-length row.names)
```

Adding to tables

Use `dbAppendTable()` to add data to an existing table.

```
1 dbAppendTable(con, name = "iris", value = iris)
```

```
Warning: Factors converted to character
```

```
[1] 150
```

```
1 dbGetQuery(con, "select * from iris") |>
2   as_tibble()
```

```
# A tibble: 150 × 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <chr>
 1          5.1         3.5          1.4         0.2 setosa
 2          4.9         3            1.4         0.2 setosa
 3          4.7         3.2          1.3         0.2 setosa
 4          4.6         3.1          1.5         0.2 setosa
 5          5           3.6          1.4         0.2 setosa
 6          5.4         3.9          1.7         0.4 setosa
 7          4.6         3.4          1.4         0.3 setosa
 8          5           3.4          1.5         0.2 setosa
 9          4.4         2.9          1.4         0.2 setosa
10          4.9         3.1          1.5         0.1 setosa
# ℹ 140 more rows
```

Closing the connection

We can explicitly close the connection to the database when we're done with it, but R will automatically close it when the connection object is garbage collected.

```
1 con
```

```
<SQLiteConnection>
  Path: employees.sqlite
  Extensions: TRUE
```

```
1 dbDisconnect(con)
```

```
1 con
```

```
<SQLiteConnection>
  DISCONNECTED
```

dplyr & databases

Creating a database

We will now create a new SQLite database,

```
db = DBI::dbConnect(RSQLite::SQLite(), "flights.sqlite") # New flights database
```

and copy data using `dplyr`'s convenience functions

```
( flight_tbl = dplyr::copy_to(
    db, nycflights13::flights, name = "flights", temporary = FALSE) )
```

```
# Source:   table<`flights`> [?? x 19]
# Database: sqlite 3.51.2 [flights.sqlite]
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ℹ more rows
# ℹ 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dbl>
```

What have we created?

All of this data now lives in the database on the *filesystem* not in *memory*,

```
pryr::object_size(db)
```

```
2.46 kB
```

```
pryr::object_size(flight_tbl)
```

```
6.50 kB
```

```
pryr::object_size(nycflights13::flights)
```

```
40.65 MB
```

```
fs::dir_info(glob = "*.sqlite") |>
  select(path, type, size)
```

```
# A tibble: 2 × 3
  path             type         size
  <fs::path>       <fct> <fs::bytes>
1 employees.sqlite file          20K
2 flights.sqlite   file        21.1M
```

What is `flight_tbl`?

```
1 class(nycflights13::flights)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
1 class(flight_tbl)
```

```
[1] "tbl_SQLiteConnection" "tbl_dbi"
[3] "tbl_sql"              "tbl_lazy"
[5] "tbl"
```

```
1 str(flight_tbl)
```

```
List of 2
 $ src       :List of 2
  ..$ con  :Formal class 'SQLiteConnection' [package "RSQLite"] with 8 slots
  .. .. ..@ ptr               :<externalptr>
  .. .. ..@ dbname            : chr "flights.sqlite"
  .. .. ..@ loadable.extensions: logi TRUE
  .. .. ..@ flags             : int 70
  .. .. ..@ vfs               : chr ""
  .. .. ..@ ref               :<environment: 0x109b39c18>
  .. .. ..@ bigint            : chr "integer64"
  .. .. ..@ extended_types    : logi FALSE
  ..$ disco: NULL
  ..- attr(*, "class")= chr [1:4] "src_SQLiteConnection" "src_dbi" "src_sql" "src"
 $ lazy_query:List of 5
  ..$ x         : 'dbplyr_table_path' chr "`flights`"
  ..$ vars      : chr [1:19] "year" "month" "day" "dep_time" ...
  ..$ group_vars: chr(0)
```

```
 ..$ order_vars: NULL
 ..$ frame     : NULL
 ..- attr(*, "class")= chr [1:3] "lazy_base_remote_query" "lazy_base_query" "lazy_query"
- attr(*, "class")= chr [1:5] "tbl_SQLiteConnection" "tbl_dbi" "tbl_sql" "tbl_lazy" ...
```

Accessing existing tables

If we have a database with an existing table, we can create a reference to it using `dplyr::tbl()`,

```
1 dplyr::tbl(db, "flights")
```

```
# Source:   table<`flights`> [?? x 19]
# Database: sqlite 3.51.2 [flights.sqlite]
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ℹ more rows
# ℹ 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dbl>
```

Using dplyr with sqlite

Just like with local data frames, dplyr can be used to manipulate data in the database.

```
(oct_21 = flight_tbl |>
   filter(month == 10, day == 21) |>
   select(origin, dest, tailnum)
)
```

```
# Source:   SQL [?? x 3]
# Database: sqlite 3.51.2 [flights.sqlite]
   origin dest  tailnum
   <chr>  <chr> <chr>
 1 EWR    CLT   N152UW
 2 EWR    IAH   N535UA
 3 JFK    MIA   N5BSAA
 4 JFK    SJU   N531JB
 5 JFK    BQN   N827JB
 6 LGA    IAH   N15710
 7 JFK    IAD   N825AS
 8 EWR    TPA   N802UA
 9 LGA    ATL   N996DL
10 JFK    FLL   N627JB
# ℹ more rows
```

```
dplyr::collect(oct_21)
```

```
# A tibble: 991 × 3
   origin dest  tailnum
   <chr>  <chr> <chr>
 1 EWR    CLT   N152UW
 2 EWR    IAH   N535UA
 3 JFK    MIA   N5BSAA
 4 JFK    SJU   N531JB
 5 JFK    BQN   N827JB
 6 LGA    IAH   N15710
 7 JFK    IAD   N825AS
 8 EWR    TPA   N802UA
 9 LGA    ATL   N996DL
10 JFK    FLL   N627JB
# ℹ 981 more rows
```

Laziness

dplyr / dbplyr uses lazy evaluation as much as possible, particularly when working with non-local backends.

- When building a query, we don't want the entire table, often we want just enough to check if our query is working / makes sense.
- Since we would prefer to run one complex query over many simple queries, laziness allows for verbs to be strung together.
- Therefore, by default `dplyr`
  - won't connect and query the database until absolutely necessary (e.g. showing output),
  - and unless explicitly told to, will only query a handful of rows to give a sense of what the result will look like.
  - we can force evaluation via `compute()`, `collect()`, or `collapse()`

A crude benchmark

```
system.time({
  (oct_21 = flight_tbl |>
    filter(month == 10, day == 21) |>
    select(origin, dest, tailnum)
  )
})
```

```
 user  system elapsed
0.002   0.000   0.002
```

```
system.time({
  print(oct_21) |>
    capture.output() |>
    invisible()
})
```

```
 user  system elapsed
0.011   0.000   0.012
```

```
system.time({
  dplyr::collect(oct_21) |>
    capture.output() |>
    invisible()
})
```

```
 user  system elapsed
0.027   0.003   0.030
```

`show_query()` - dplyr to SQL

```
1 class(oct_21)
```

```
[1] "tbl_SQLiteConnection" "tbl_dbi"
[3] "tbl_sql"              "tbl_lazy"
[5] "tbl"
```

```
1 dplyr::show_query(oct_21)
```

```
<SQL>
SELECT `origin`, `dest`, `tailnum`
FROM `flights`
WHERE (`month` = 10.0) AND (`day` = 21.0)
```

More complex queries

```
1 oct_21 |>
2   summarize(
3     n=n(), .by = c(origin, dest)
4   )
```

```
# Source:   SQL [?? x 3]
# Database: sqlite 3.51.2 [flights.sqlite]
   origin dest      n
   <chr>  <chr> <int>
 1 EWR    ATL      15
 2 EWR    AUS       3
 3 EWR    AVL       1
 4 EWR    BNA       7
 5 EWR    BOS      17
 6 EWR    BTV       3
 7 EWR    BUF       2
 8 EWR    BWI       1
 9 EWR    CHS       4
10 EWR    CLE       4
# ℹ more rows
```

```
1 oct_21 |>
2   summarize(
3     n=n(), .by = c(origin, dest)
4   ) |>
5   dplyr::show_query()
```

```
<SQL>
SELECT `origin`, `dest`, COUNT(*) AS `n`
FROM (
  SELECT `origin`, `dest`, `tailnum`
  FROM `flights`
  WHERE (`month` = 10.0) AND (`day` = 21.0
) AS `q01`
GROUP BY `origin`, `dest`
```

```r
oct_21 |>
  count(origin, dest) |>
  dplyr::show_query()
```

```
<SQL>
SELECT `origin`, `dest`, COUNT(*) AS `n`
FROM (
  SELECT `origin`, `dest`, `tailnum`
  FROM `flights`
  WHERE (`month` = 10.0) AND (`day` = 21.0)
) AS `q01`
GROUP BY `origin`, `dest`
```

SQL Translation

In general, dplyr / dbplyr knows how to translate basic math, logical, and summary functions from R to SQL.

dbplyr has the function, `translate_sql()`, that lets you experiment with how R functions are translated to SQL.

```
con = dbplyr::simulate_dbi()
dbplyr::translate_sql(x == 1 & (y < 2 | z > 3), con=con)
```

```
<SQL> `x` = 1.0 AND (`y` < 2.0 OR `z` > 3.0)
```

```
dbplyr::translate_sql(x ^ 2 < 10, con=con)
```

```
<SQL> (POWER(`x`, 2.0)) < 10.0
```

```
dbplyr::translate_sql(x %% 2 == 10, con=con)
```

```
<SQL> (`x` % 2.0) = 10.0
```

```
dbplyr::translate_sql(sd(x), con=con)
```

```
Error in `sd()`:
! `sd()` is not available in this SQL variant.
```

```
dbplyr::translate_sql(mean(x), con=con)
```

```
Warning: Missing values are always removed in SQL aggregation functions.
Use `na.rm = TRUE` to silence this warning
This warning is displayed once every 8 hours.
```

```
<SQL> AVG(`x`) OVER ()
```

```
dbplyr::translate_sql(mean(x, na.rm=TRUE), con=con)
```

```
<SQL> AVG(`x`) OVER ()
```

```
dbplyr::translate_sql(paste(x,y), con=con)
```

```
<SQL> CONCAT_WS(' ', `x`, `y`)
```

```
dbplyr::translate_sql(cumsum(x), con=con)
```

```
Warning: Windowed expression `SUM(`x`)` does not have explicit order.
ℹ Please use `arrange()` or `window_order()` to make deterministic.
```

```
<SQL> SUM(`x`) OVER (ROWS UNBOUNDED PRECEDING)
```

```
dbplyr::translate_sql(lag(x), con=con)
```

```
<SQL> LAG(`x`, 1, NULL) OVER ()
```

Dialectic variations?

By default `dbplyr::translate_sql()` will translate R / dplyr code into ANSI SQL, if we want to see results specific to a certain database we can pass in a connection object,

```
1 dbplyr::translate_sql(sd(x), con = db)
```

```
<SQL> STDEV(`x`) OVER ()
```

```
1 dbplyr::translate_sql(paste(x,y), con = db)
```

```
<SQL> `x` || ' ' || `y`
```

```
1 dbplyr::translate_sql(cumsum(x), con = db)
```

```
Warning: Windowed expression `SUM(`x`)` does not have explicit order.
ℹ Please use `arrange()` or `window_order()` to make deterministic.
<SQL> SUM(`x`) OVER (ROWS UNBOUNDED PRECEDING)
```

```
1 dbplyr::translate_sql(lag(x), con = db)
```

```
<SQL> LAG(`x`, 1, NULL) OVER ()
```

Complications?

Not all R functions have a translation to SQL, and not all translations are perfect - when `dbplyr` encounters a function it doesn't know how to translate, it will include the function verbatim and hope for the best.

```
1 oct_21 |> mutate(tailnum_n_prefix = grepl("^N", tailnum))
```

```
Error in `collect()`:
! Failed to collect lazy table.
Caused by error:
! no such function: grepl
```

```
1 oct_21 |> mutate(tailnum_n_prefix = grepl("^N", tailnum)) |> show_query()
```

```
<SQL>
SELECT `origin`, `dest`, `tailnum`, grepl('^N', `tailnum`) AS `tailnum_n_prefix`
FROM `flights`
WHERE (`month` = 10.0) AND (`day` = 21.0)
```

Joins

Join verbs (`left_join()`, `inner_join()`, etc.) work with database tables exactly as they do with data frames:

```
flight_tbl |>
  left_join(airlines_tbl, by = "carrier") |>
  count(name, origin) |>
  arrange(desc(n))
```

```
# Source:     SQL [?? x 3]
# Database:   sqlite 3.51.2 [flights.sqlite]
# Ordered by: desc(n)
   name                     origin     n
   <chr>                    <chr>  <int>
 1 United Air Lines Inc.    EWR    46087
 2 ExpressJet Airlines Inc. EWR    43939
 3 JetBlue Airways          JFK    42076
 4 Delta Air Lines Inc.     LGA    23067
 5 Delta Air Lines Inc.     JFK    20701
 6 Envoy Air                LGA    16928
 7 American Airlines Inc.   LGA    15459
 8 Endeavor Air Inc.        JFK    14651
 9 American Airlines Inc.   JFK    13783
10 US Airways Inc.          LGA    13136
# ℹ more rows
```

```
flight_tbl |>
  left_join(airlines_tbl, by = "carrier") |>
  count(name, origin) |>
  arrange(desc(n)) |>
  show_query()
```

```
<SQL>
SELECT `name`, `origin`, COUNT(*) AS `n`
FROM (
  SELECT `flights`.*, `name`
  FROM `flights`
  LEFT JOIN `airlines`
    ON (`flights`.`carrier` = `airlines`.`carrie
) AS `q01`
GROUP BY `name`, `origin`
ORDER BY `n` DESC
```

Computation happens in the database

All dplyr operations on a database table are translated to SQL and executed by the database — no R computation happens until you `collect()`.

This means you cannot mix local objects with database tables:

```
1 local_df = data.frame(origin = c("JFK", "LGA"))
2 flight_tbl |> inner_join(local_df, by = "origin")
```

```
Error in `auto_copy()`:
! `x` and `y` must share the same src.
ℹ `x` is a <tbl_SQLiteConnection/tbl_dbi/tbl_sql/tbl_lazy/tbl>
  object.
ℹ `y` is a <data.frame> object.
ℹ Set `copy = TRUE` if `y` can be copied to the same source as `x`
  (may be slow).
```

To work around this, either `collect()` the database table first, or copy the local object into the database:

```
1 flight_tbl |> inner_join(local_df, by = "origin", copy = TRUE)
```

```
# Source:   SQL [?? x 19]
# Database: sqlite 3.51.2 [flights.sqlite]
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      542            540         2      923
 2  2013     1     1      544            545        -1     1004
 3  2013     1     1      557            600        -3      838
 4  2013     1     1      558            600        -2      849
 5  2013     1     1      558            600        -2      853
```

```
 6  2013     1     1      558           600        -2      924
 7  2013     1     1      559           559         0      702
 8  2013     1     1      606           610        -4      837
 9  2013     1     1      611           600        11      945
10  2013     1     1      613           610         3      925
# ℹ more rows
# ℹ 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dbl>
```

Window functions

Window functions compute a value for each row using values from a surrounding partition. Within a `group_by()` + `mutate()`, dplyr translates these to SQL `OVER()` clauses:

```
flight_tbl |>
  filter(!is.na(dep_delay)) |>
  group_by(origin) |>
  mutate(delay_rank = min_rank(desc(dep_delay))) |>
  filter(delay_rank <= 3) |>
  select(origin, dest, dep_delay, delay_rank) |>
  arrange(origin, delay_rank)
```

```
# Source:     SQL [?? x 4]
# Database:   sqlite 3.51.2 [flights.sqlite]
# Groups:     origin
# Ordered by: origin, delay_rank
  origin dest  dep_delay delay_rank
  <chr>  <chr>     <dbl>      <int>
1 EWR    ORD        1126          1
2 EWR    MIA         896          2
3 EWR    ORD         878          3
4 JFK    HNL        1301          1
5 JFK    CMH        1137          2
6 JFK    SFO        1014          3
7 LGA    MSP         911          1
8 LGA    ATL         898          2
9 LGA    DEN         853          3
```

```
flight_tbl |>
  filter(!is.na(dep_delay)) |>
  group_by(origin) |>
  mutate(delay_rank = min_rank(desc(dep_delay))) |>
  filter(delay_rank <= 3) |>
  select(origin, dest, dep_delay, delay_rank) |>
  arrange(origin, delay_rank) |>
  show_query()
```

```
<SQL>
SELECT `origin`, `dest`, `dep_delay`, `delay_rank`
FROM (
  SELECT
    `flights`.*,
    CASE
WHEN (NOT((`dep_delay` IS NULL))) THEN RANK() OVER (PAR
END AS `delay_rank`
  FROM `flights`
  WHERE (NOT((`dep_delay` IS NULL)))
) AS `q01`
WHERE (`delay_rank` <= 3.0)
ORDER BY `origin`, `delay_rank`
```

SQL to R / dplyr

Running SQL queries against R objects

There are two packages that implement this in R which take different approaches,

- `tidyquery` - this package parses your SQL code using the `queryparser` package and then translates the result into R / dplyr code.
- `sqldf` - transparently creates a database with the data and then runs the query using that database. Defaults to SQLite but other backends are available.

tidyquery

```
data(flights, package = "nycflights13")

tidyquery::query(
  "SELECT origin, dest, COUNT(*) AS n
   FROM flights
   WHERE month = 10 AND day = 21
   GROUP BY origin, dest"
)
```

```
# A tibble: 181 × 3
   origin dest      n
   <chr>  <chr> <int>
 1 EWR    ATL      15
 2 EWR    AUS       3
 3 EWR    AVL       1
 4 EWR    BNA       7
 5 EWR    BOS      17
 6 EWR    BTV       3
 7 EWR    BUF       2
 8 EWR    BWI       1
 9 EWR    CHS       4
10 EWR    CLE       4
# ℹ 171 more rows
```

```
flights |>
  tidyquery::query(
    "SELECT origin, dest, COUNT(*) AS n
     WHERE month = 10 AND day = 21
     GROUP BY origin, dest"
  ) |>
  arrange(desc(n))
```

```
# A tibble: 181 × 3
   origin dest      n
   <chr>  <chr> <int>
 1 JFK    LAX      32
 2 LGA    ORD      31
 3 LGA    ATL      30
 4 JFK    SFO      24
 5 LGA    CLT      22
 6 EWR    ORD      18
 7 EWR    SFO      18
 8 EWR    BOS      17
 9 LGA    MIA      17
10 EWR    LAX      16
# ℹ 171 more rows
```

Translating to dplyr

```
tidyquery::show_dplyr(
  "SELECT origin, dest, COUNT(*) AS n
   FROM flights
   WHERE month = 10 AND day = 21
   GROUP BY origin, dest"
)
```

```
flights %>%
  filter(month == 10 & day == 21) %>%
  group_by(origin, dest) %>%
  summarise(n = dplyr::n()) %>%
  ungroup()
```

sqldf

```
sqldf::sqldf(
  "SELECT origin, dest, COUNT(*) AS n
   FROM flights
   WHERE month = 10 AND day = 21
   GROUP BY origin, dest"
)
```

```
Warning in fun(libname, pkgname): no display nam
environment variable
```

```
    origin dest  n
1      EWR  ATL 15
2      EWR  AUS  3
3      EWR  AVL  1
4      EWR  BNA  7
5      EWR  BOS 17
6      EWR  BTV  3
7      EWR  BUF  2
8      EWR  BWI  1
9      EWR  CHS  4
10     EWR  CLE  4
11     EWR  CLT 15
12     EWR  CMH  3
13     EWR  CVG  9
14     EWR  DAY  4
15     EWR  DCA  3
16     EWR  DEN  8
17     EWR  DFW  9
18     EWR  DSM  2
19     EWR  DTW 10
20     EWR  FLL 10
21     EWR  GRR  2
22     EWR  GSO  4
```

```
sqldf::sqldf(
  "SELECT origin, dest, COUNT(*) AS n
   FROM flights
   WHERE month = 10 AND day = 21
   GROUP BY origin, dest"
) |>
  as_tibble() |>
  arrange(desc(n))
```

```
# A tibble: 181 × 3
   origin dest      n
   <chr>  <chr> <int>
 1 JFK    LAX      32
 2 LGA    ORD      31
 3 LGA    ATL      30
 4 JFK    SFO      24
 5 LGA    CLT      22
 6 EWR    ORD      18
 7 EWR    SFO      18
 8 EWR    BOS      17
 9 LGA    MIA      17
10 EWR    LAX      16
# ℹ 171 more rows
```

Conclusions

dplyr / dbplyr vs DBI

dplyr / dbplyr translates R expressions into SQL **SELECT** queries — it is a read-only interface:

Operation	dplyr / dbplyr	DBI
Query / filter / summarize	✓	✓
Insert rows	✗	✓ (`dbAppendTable()`, `dbExecute()`)
Update rows	✗	✓ (`dbExecute()`)
Delete rows	✗	✓ (`dbExecute()`)
Create / drop tables	✗	✓ (`dbCreateTable()`, `dbRemoveTable()`)

For anything beyond reading data, go through DBI directly.

Closing thoughts

The ability of dplyr to translate from R expression to SQL is an incredibly powerful tool making your data processing workflows portable across a wide variety of data backends.

Some tools and ecosystems that are worth learning about:

- Spark - sparkR, spark SQL, sparklyr
- DuckDB
- Apache Arrow