databases & dplyr

Lecture 18

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

Lets imagine we have a *10 GB* flat data file and that we want to select certain rows based on a particular criteria. 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* 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

So usually possible to grow our disk storage to accommodate our data. However, memory is usually the limiting resource, and if we can’t fit everything into memory?

Create *blocks* - group related data (i.e. rows) and read in multiple rows at a time. Optimal size will depend on the task and the properties of the disk.

Linear vs Binary Search

Even with blocks, any kind of querying / subsetting of rows requires a linear search, which requires $\square(N)$ 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 a databases these "sorted" columns are refered to as *indexes*.
- Indexes require additional storage, but usually small enough to be kept in memory even if blocks need to stay on disk.

and then?

This is just barely scratching the surface,

- Efficiency gains are not just for disk, access is access
- In general, trade off between storage and efficiency
- Reality is a lot more complicated for everything mentioned so far, lots of very smart people have spent a lot of time thinking about and implementing tools
- Different tasks with different requirements require different implementations and have different criteria for optimization

Databases

R & databases - the DBI package

Low level package for interfacing R with Database management systems (DBMS) that provides a common interface to achieve the following functionality:

- connect/disconnect from DB
- create and execute statements in the DB
- extract results/output from statements
- error/exception handling
- information (meta-data) from database objects
- transaction management (optional)

DBI Backends

DBI is a specification, not an implementation, and there are a number of packages that implement the DBI specification for different database systems. Backends for R-DBI lists all available backends, but some notable ones include:

- `RPostgres`
- `RMariaDB`
- `RSQLite`
- `odbc`
- `bigrquery`
- `duckdb`
- `sparklyr`

RSQLite

Provides the implementation necessary to use DBI to interface with an SQLite database.

```
library(RSQLite)
```

this package also loads the necessary DBI functions as well (via re-exporting).

Once loaded we can create a connection to our database,

```
con = dbConnect(RSQLite::SQLite(), ":memory:")
str(con)
```

```
Formal class 'SQLiteConnection' [package "RSQLite"] with 8 slots
  ..@ ptr                :<externalptr>
  ..@ dbname             : chr ":memory:"
  ..@ loadable.extensions: logi TRUE
  ..@ flags              : int 70
  ..@ vfs                : chr ""
  ..@ ref                :<environment: 0x1156a6470>
  ..@ bigint             : chr "integer64"
  ..@ extended_types     : logi FALSE
```

Example Table

```r
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@comany.com"),
  salary = c(52000, 40000, 30000, 33000, 44000, 37000),
  dept   = c("Accounting", "Accounting","Sales",
             "Accounting","Sales","Sales"),
)
```

```r
dbListTables(con)
```

```
character(0)
```

```r
dbWriteTable(con, name = "employees", value = employees)
dbListTables(con)
```

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

Removing Tables

```r
dbWriteTable(con, "employs", employees)
dbListTables(con)
```

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

```r
dbRemoveTable(con,"employs")
dbListTables(con)
```

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

Querying Tables

Databases queries are transactional (see ACID) and are broken up into 3 steps:

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

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

```r
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@comany.com  37000      Sales
```

```r
dbClearResult(res)
```

For convenience

There is also `dbGetQuery()` which combines all three steps,

```r
(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@comany.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.

```r
dbCreateTable(con, "iris", iris)
(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

Data can be added to an existing table via `dbAppendTable()`.

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

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

```
[1] 150
```

```r
dbGetQuery(con, "select * from iris") |>
  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

```r
con
```

```
<SQLiteConnection>
  Path: :memory:
  Extensions: TRUE
```

```r
dbDisconnect(con)
```

```r
con
```

```
<SQLiteConnection>
  DISCONNECTED
```

dplyr & databases

Creating a database

```r
db = DBI::dbConnect(RSQLite::SQLite(), "flights.sqlite")
( flight_tbl = dplyr::copy_to(
    db, nycflights13::flights, name = "flights", temporary = FALSE) )
```

```
# Source:   table<`flights`> [?? x 19]
# Database: sqlite 3.50.4 [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*,

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

```
2.46 kB
```

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

```
6.50 kB
```

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

```
40.65 MB
```

File size

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

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

What is `flight_tbl`?

```r
class(nycflights13::flights)
```

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

```r
class(flight_tbl)
```

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

```r
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: 0x116b81d98>
  .. .. ..@ 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

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

```
# Source:   table<`flights`> [?? x 19]
# Database: sqlite 3.50.4 [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

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

```
# Source:   SQL [?? x 3]
# Database: sqlite 3.50.4 [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
```

```r
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. show 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 
0.003   0.000   0.002 
```

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

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

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

```
 user  system elapsed 
0.028   0.002   0.031 
```

`show_query()` - dplyr to SQL

```r
class(oct_21)
```

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

```r
show_query(oct_21)
```

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

More complex queries

```r
oct_21 |>
  summarize(
    n=n(), .by = c(origin, dest)
  )
```

```
# Source:   SQL [?? x 3]
# Database: sqlite 3.50.4 [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
```

```r
oct_21 |>
  summarize(
    n=n(), .by = c(origin, dest)
  ) |>
  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) |>
  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 a function, `translate_sql()`, that lets you experiment with how R functions are translated to SQL.

```r
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)
```

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

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

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

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

```r
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 ()
```

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

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

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

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

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

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

```r
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)
```

```r
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,

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

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

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

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

```r
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)
```

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

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

Complications?

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

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

```r
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)
```

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

```r
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
```

```r
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

```r
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

```r
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 displ
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
```

```r
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
```

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