databases & dplyr

Lecture 16

Dr. Colin Rundel

The why of databases

Numbers every programmer should know

Task	Timing (ns)	Timing (μs)
L1 cache reference	0.5	
L2 cache reference	7	
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

Jeff Dean's original talk

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	$10\ \text{GB} \times (250\ \mu\text{s}/1\ \text{MB})$	2.5 seconds
on disk (SSD)	$10\ \text{GB} \times (1\ \text{ms}/1\ \text{MB})$	10 seconds
on disk (HD)	$10\ \text{GB} \times (20\ \text{ms}/1\ \text{MB})$	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 (N) accesses where N is the number of blocks.

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

- Sorting is expensive, $(N \log N)$, but it only needs to be done once.
- After sorting, we can use a binary search for any subsetting tasks ($(\log N))$).
- These “sorted” columns are known as *indexes*.
- Indexes require additional storage, but usually small enough to be kept in memory while blocks 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)

See r-dbi.org for more details

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.

Once loaded we can create a connection to our database,

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

## Formal class 'SQLiteConnection' [package "RSQLite"] with 5 slots
##   ..@ Id                 :<externalptr>
##   ..@ dbname             : chr ":memory:"
##   ..@ loadable.extensions: logi TRUE
##   ..@ flags              : int 6
##   ..@ vfs                : chr ""
```

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

dbWriteTable(con, name = "employees", value = employees)
## [1] TRUE

dbListTables(con)
## [1] "employees"
```

Removing Tables

```
dbWriteTable(con, "employs", employees)
## [1] TRUE

dbListTables(con)
## [1] "employees" "employs"

dbRemoveTable(con,"employs")
## [1] TRUE

dbListTables(con)
## [1] "employees"
```

Querying Tables

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

```
(res = dbSendQuery(con, "SELECT * FROM employees"))
## <SQLiteResult>
##   SQL  SELECT * FROM employees
##   ROWS Fetched: 0 [incomplete]
##        Changed: 0

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

dbClearResult(res)
## [1] TRUE
```

For cenvenience

There is also `dbGetQuery()` for simpler use cases,

```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

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

(res = dbSendQuery(con, "select * from iris"))
## <SQLiteResult>
##   SQL  select * from iris
##   ROWS Fetched: 0 [complete]
##        Changed: 0

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

```r
dbFetch(res) %>% as_tibble()
## # A tibble: 0 × 5
## # … with 5 variables: Sepal.Length <dbl>, Sepal.Width <dbl>, Petal.Length
## #   Petal.Width <dbl>, Species <chr>

dbClearResult(res)
```

Adding to tables

```
dbAppendTable(con, name = "iris", value = iris)
## [1] 150
## Warning message:
## Factors converted to character
```

```
res = dbSendQuery(con, "select * from iris")

dbFetch(res) %>% as_tibble()
## # A tibble: 150 x 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
## # … with 140 more rows
```

Ephemeral results

```
res
## <SQLiteResult>
##   SQL  select * from iris
##   ROWS Fetched: 150 [complete]
##        Changed: 0

dbFetch(res) %>% as_tibble()
## # A tibble: 0 x 5
## # … with 5 variables: Sepal.Length <dbl>, Sepal.Width <dbl>, Petal.Length
## #   Species <chr>

dbClearResult(res)
```

Closing the connection

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

dbDisconnect(con)
## [1] TRUE

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 3.39.4 [flights.sqlite]
    year month   day dep_time sched_…¹ dep_d…² arr_t…³ sched…⁴ arr_d…⁵
   <int> <int> <int>    <int>    <int>   <dbl>   <int>   <int>   <dbl>
 1  2013     1     1      517      515       2     830     819      11
 2  2013     1     1      533      529       4     850     830      20
 3  2013     1     1      542      540       2     923     850      33
 4  2013     1     1      544      545      -1    1004    1022     -18
 5  2013     1     1      554      600      -6     812     837     -25
 6  2013     1     1      554      558      -4     740     728      12
 7  2013     1     1      555      600      -5     913     854      19
 8  2013     1     1      557      600      -3     709     723     -14
 9  2013     1     1      557      600      -3     838     846      -8
10  2013     1     1      558      600      -2     753     745       8
```

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)
```

```
6.46 kB
```

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

```
40.65 MB
```

```r
fs::dir_info(glob = "*.sqlite")
```

```
# A tibble: 1 × 18
  path           type   size permiss…¹ modification_time   user  group
  <fs::path>     <fct> <fs:> <fs::per> <dttm>              <chr> <chr>
1 flights.sqlite file  21.1M rw-r--r-- 2022-10-24 09:51:59 rund… staff
# … with 11 more variables: device_id <dbl>, hard_links <dbl>,
#   special_device_id <dbl>, inode <dbl>, block_size <dbl>,
#   blocks <dbl>, flags <int>, generation <dbl>, access_time <dttm>,
```

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: 0x123030200>
  .. .. ..@ bigint             : chr "integer64"
  .. .. ..@ extended_types     : logi FALSE
  ..$ disco: NULL
  ..- attr(*, "class")= chr [1:4] "src_SQLiteConnection" "src_dbi" "src_sql" "src"
```

Accessing existing tables

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

```
# Source:   table<flights> [?? x 19]
# Database: sqlite 3.39.4 [flights.sqlite]
    year month   day dep_time sched_…¹ dep_d…² arr_t…³ sched…⁴ arr_d…⁵
   <int> <int> <int>    <int>    <int>   <dbl>   <int>   <int>   <dbl>
 1  2013     1     1      517      515       2     830     819      11
 2  2013     1     1      533      529       4     850     830      20
 3  2013     1     1      542      540       2     923     850      33
 4  2013     1     1      544      545      -1    1004    1022     -18
 5  2013     1     1      554      600      -6     812     837     -25
 6  2013     1     1      554      558      -4     740     728      12
 7  2013     1     1      555      600      -5     913     854      19
 8  2013     1     1      557      600      -3     709     723     -14
```

Using dplyr with sqlite

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

```
# Source:   SQL [?? x 3]
# Database: sqlite 3.39.4 [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
```

```
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
# … with 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   0.003 
```

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

```
 user  system elapsed 
0.019   0.001   0.020 
```

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

```
 user  system elapsed 
0.039   0.005   0.044 
```

dplyr -> SQL - `show_query()`

```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 %>%
  group_by(origin, dest) %>%
  summarize(n=n(), .groups = "drop")
```

```
# Source:   SQL [?? x 3]
# Database: sqlite 3.39.4 [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
```

```r
oct_21 %>%
  group_by(origin, dest) %>%
  summarize(n=n(), .groups = "drop") %>%
  show_query()
```

```
<SQL>
SELECT `origin`, `dest`, COUNT(*) AS `n`
FROM (
  SELECT `origin`, `dest`, `tailnum`
  FROM `flights`
  WHERE (`month` = 10.0) AND (`day` = 21.0)
)
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)
)
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
dbplyr::translate_sql(x == 1 & (y < 2 | z > 3))
```

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

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

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

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

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

```r
dbplyr::translate_sql(mean(x))
```

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

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

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

```r
dbplyr::translate_sql(sd(x))
```

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

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

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

```r
dbplyr::translate_sql(cumsum(x))
```

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

```r
dbplyr::translate_sql(lag(x))
```

```
<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,

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

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

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

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

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

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

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

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

Complications?

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

```
Error: no such function: grepl
```

```
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 -> R / dplyr

Running SQL queries against R objects

There are two packages that implement this in R which take very 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 teh 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
# … with 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
# … with 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"
)
```

```
   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
```

```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
# … with 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