Analytical Data Management with R

Hannes Mühleisen

Overview

1. **Motivations to use a Database**
2. System Scenarios
3. R and Databases State of the Art
4. Future Directions

Database Example

- Database that models a digital music store to keep track of artists and albums.
- Things we need to store:
  - Information about **artists**.
  - What **albums** those artists released.

1960 Solution: Flat files

- Store database as comma-separated value (CSV) files that we manage in our own code
    - Use separate file per "entity" (artist, album)
    - The analysis has to **parse** files each time they want to read/update records

Flat File Example

Artist (name, year, country)

```
“Backstreet Boys”,1994,"USA"

“Ice Cube”,1992,"USA"

“Notorious BIG”,1989,"USA"
```

Album (name,artist,year)

```
“Millenium", "Backstreet Boys", 1999

“DNA”, “Backstreet Boys”, 2019

“AmeriKKKa's Most Wanted”, “Ice Cube”, 1990
```

Flat File Example

"Get the year Ice Cube went solo"

Artist (name, year, country)

```
“Backstreet Boys”,1994,"USA"

“Ice Cube”,1992,"USA"

“Notorious BIG”,1989,"USA"
```

```
df <- read.csv("artists.csv", header=F,
 col.names=c("name", "year", "country"))
df[df$name=="Ice Cube", "year"]
```

Multiple passes through entire dataset!

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models

Data Integrity

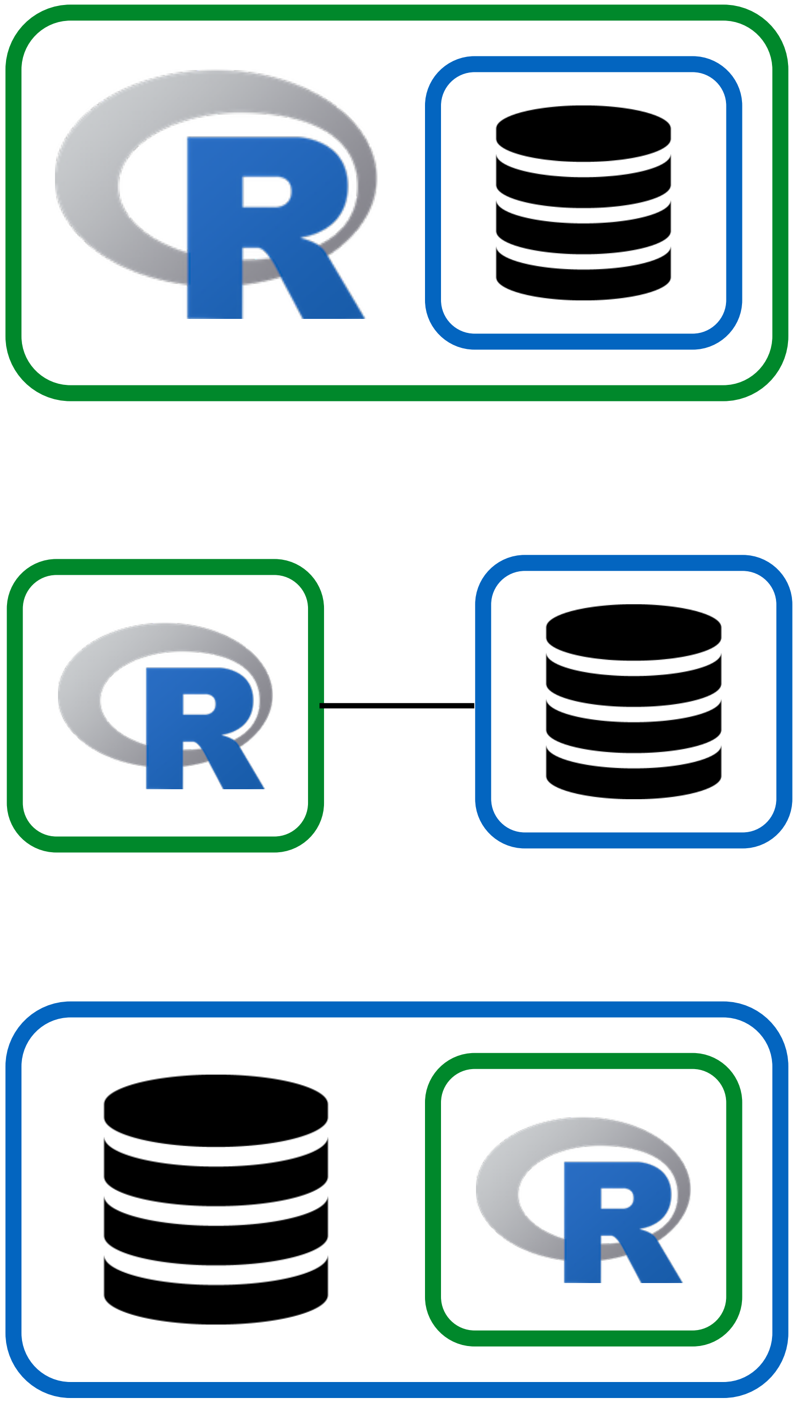
- How do we ensure that the artist is the same for each album entry?
- What if someone overwrites the album year with an invalid string?
- How do we store that there are multiple artists on an album?
- How do we update several tables with all-or-nothing semantics?
- How do we keep derived data up-to-date?

Overview

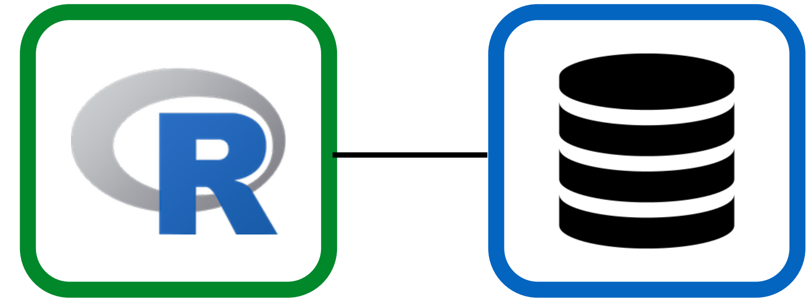
1. Motivations to use a Database
2. **System Scenarios**
3. R and Databases State of the Art
4. Future Directions

System Scenarios

- In-Process Database
- External Database
- User-Defined Functions

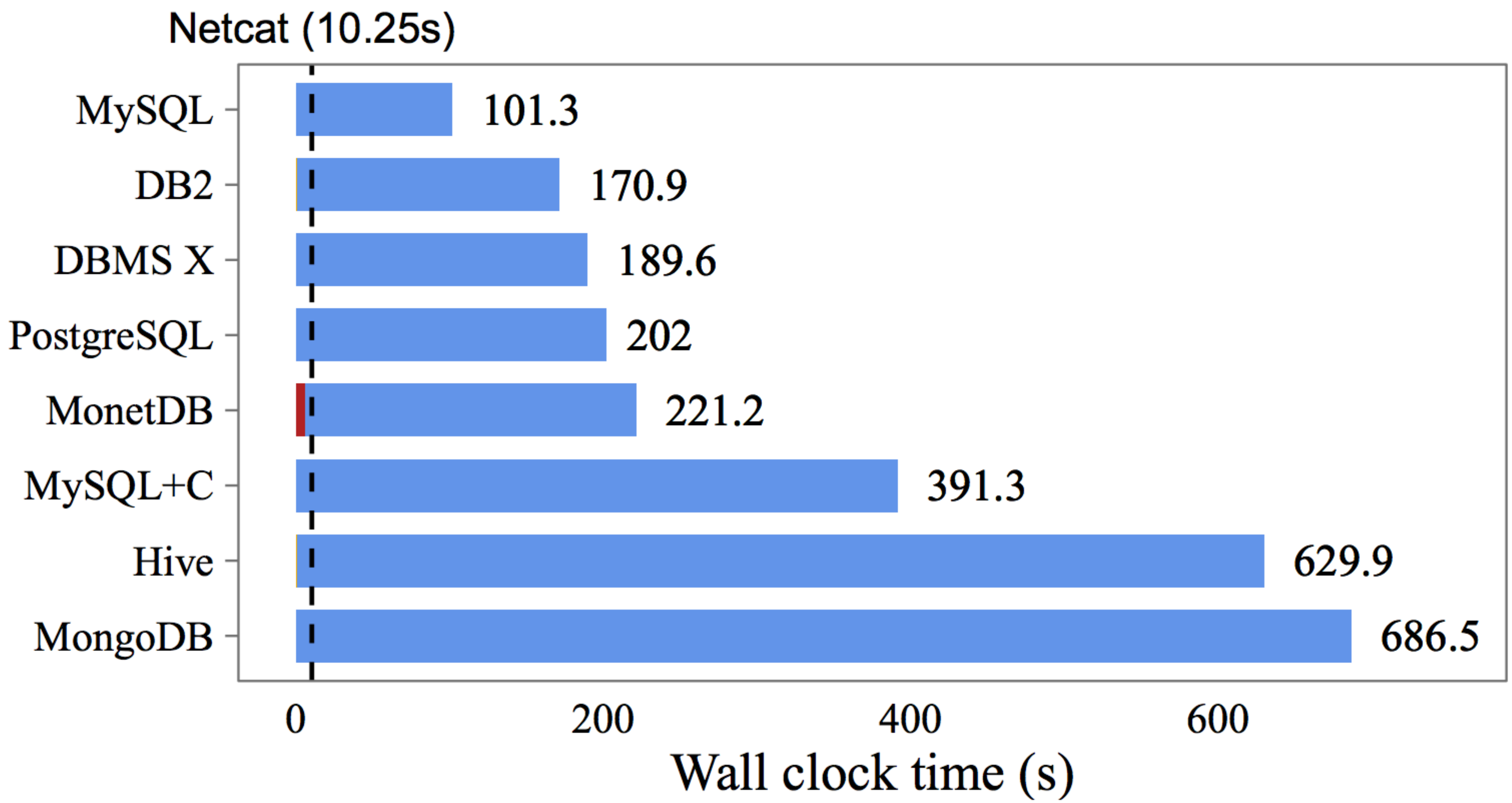


External DB



- MySQL, PostgreSQL, SQL Server, Oracle, Redshift
- Hive, Impala, BigQuery
- (Spark)
- Transferring large-is datasets slooow
    - Need complex SQL to fetch relevant data!

Client protocols?




```
SELECT * FROM lineitem_sf10;
```

[M. Raasveldt & H. Mühleisen:
Don't Hold My Data Hostage - A Case For Client Protocol Redesign, VLDB 2017]

In-Process DB

- Transactional **persistent** data management
- RSQLite, DuckDB, (MonetDBLite)
- Faster, but still conversion overhead :/
    - ALTREP to the rescue
    - Later...

User-Defined Functions

- PostgreSQL PL/R (Joe Conway)
- MonetDB R UDFs
- Oracle ~
- Spark ~
- SQL Server ~
- Can be also fast, but also still some translation overhead.

Kinds of UDFs for SELECT

- Filters
  - `SELECT b FROM t WHERE fun(a)`
  - UDF returns TRUE/FALSE, only rows where it returns TRUE are returned
- Projection
  - `SELECT fun(a, b) FROM t`
  - UDF returns a single scalar value, becomes part of query result
- Table-Returning
  - `SELECT * FROM fun(42)`
  - UDF returns a whole intermediate result table

Postgres PL/R

```
CREATE OR REPLACE FUNCTION get_emps() RETURNS
SETOF emp AS '
  names <- c("Joe","Jim","Jon")
  ages <- c(41,25,35)
  salaries <- c(250000,120000,50000)
  df <- data.frame(name = names, age = ages,
    salary = salaries)
  return(df)
' LANGUAGE 'plr';
```

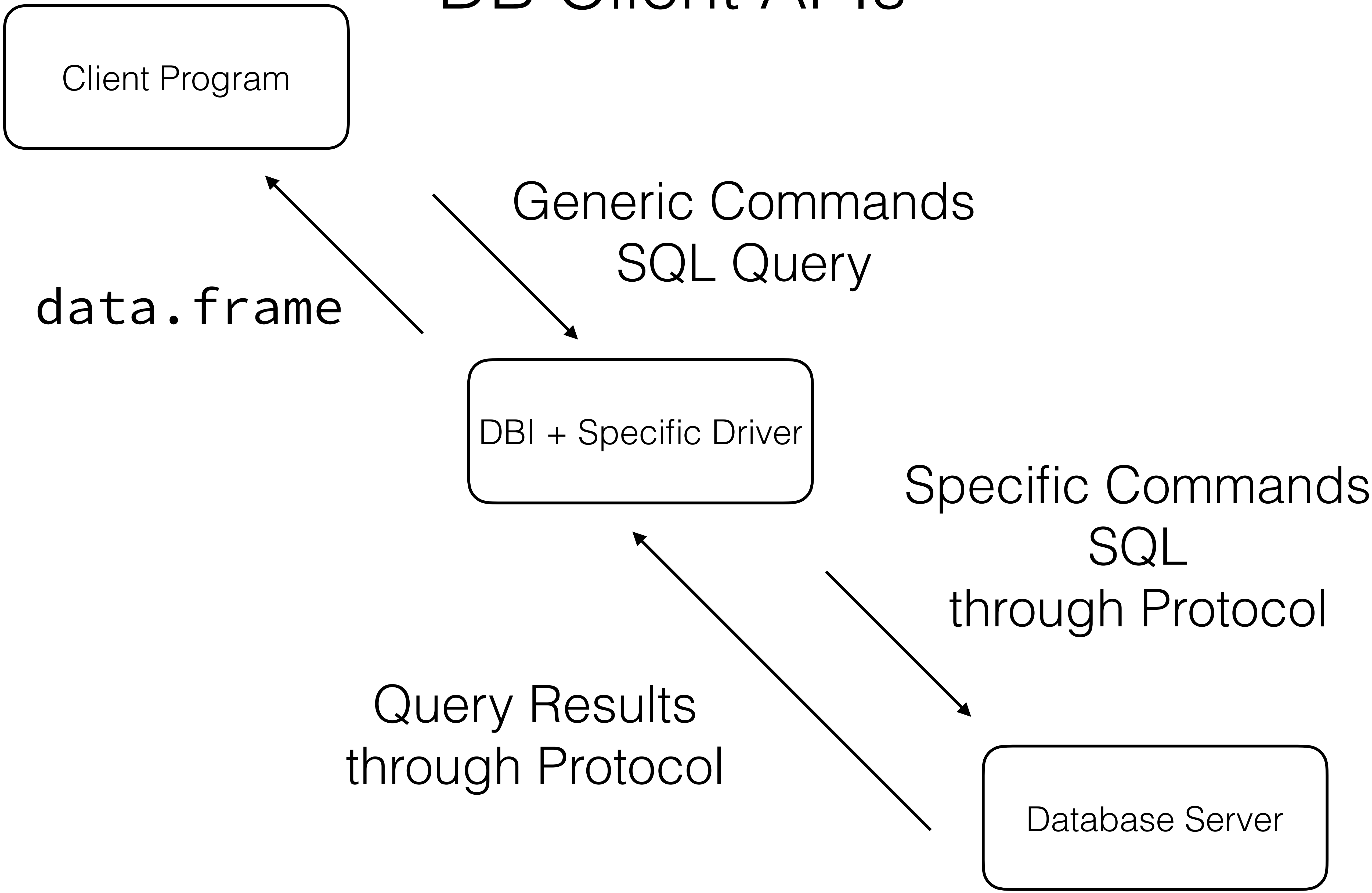
Postgres PL/R

```
select * from get_emps();
 name | age |  salary
------+-----+-----------
 Jim  |  41 | 250000.00
 Joe  |  25 | 120000.00
 Jon  |  35 |  50000.00
(3 rows)
```

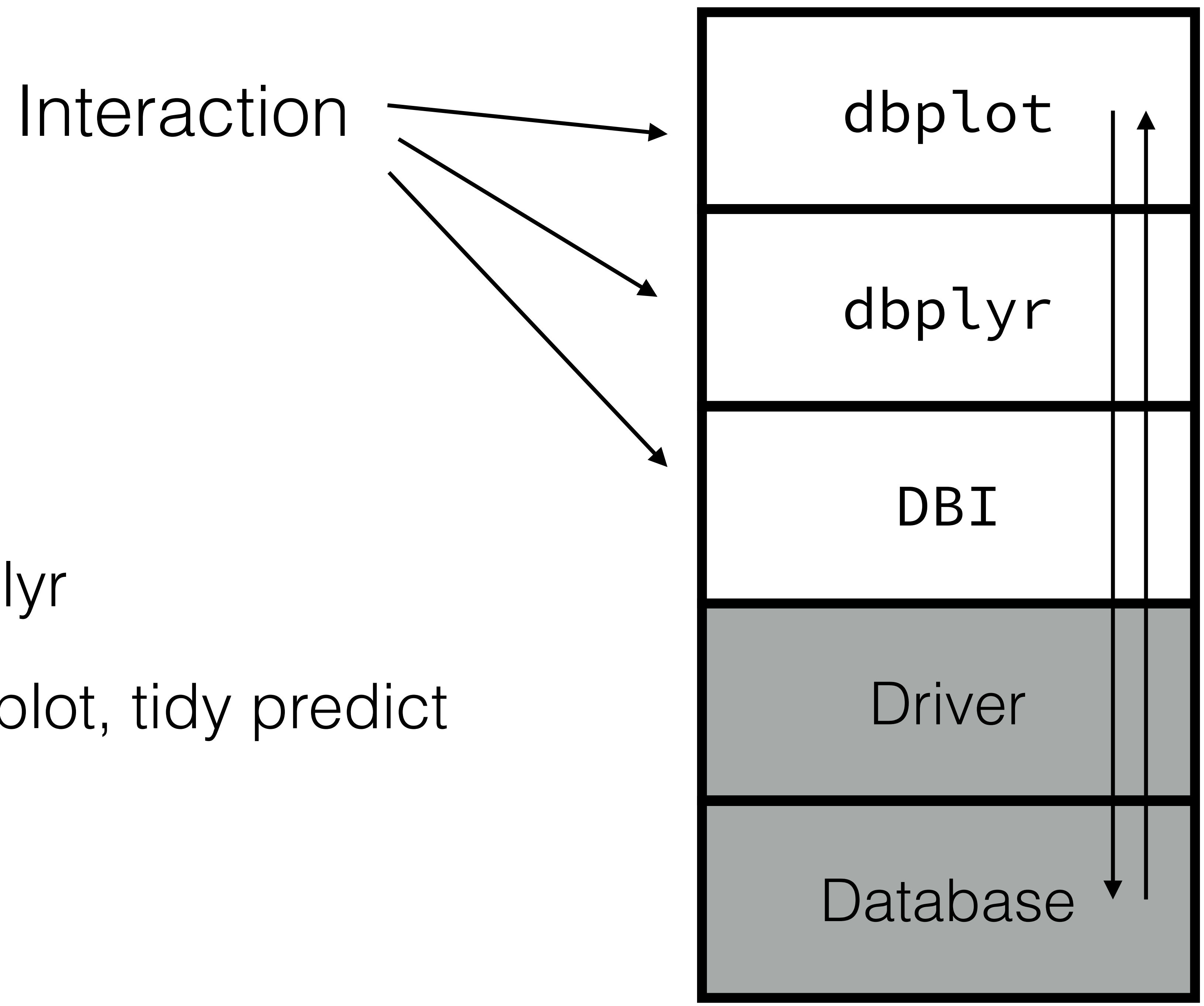
Overview

1. Motivations to use a Database
2. System Scenarios
3. **R and Databases State of the Art**
4. Future Directions

DB Client APIs



Database APIs for R



1. Low-Level: DBI
2. Verbs: dplyr/dbplyr
3. Applications: dbplot, tidy predict

DBI

- Basic API, adapts database-specific API/protocol into **unified** R API
- Queries are strings, mostly SQL
- Results are `data.frame` objects
- **`dbConnect`**/`dbDisconnect`
- `dbListTables`/`dbListFields`
- `dbWriteTable`
- `dbGetQuery`/`dbExecute`/`dbReadTable`

DBI

- Lots of implementations: RMySQL, ROracle, RPostgreSQL, RRedshiftSQL, RClickhouse, RGreenplum, RMariaDB, **RSQLite**, virtuoso, sparklyr
- Generic wrappers: RJDBC, odbc
    - Great if your DB vendor does not have R-specific driver
- Heroic effort by Kirill Müller: DBItest
    - Result: Driver quality varies :/

DBI?

- Upside: Can talk to databases
- Downside: Need to construct SQL strings :/
- Higher-level interface might be nice?

`dplyr`

- Data reorganisation thing in “xyzverse”
- `dbplyr`: extension to work with SQL DBs, ~~Spark~~, ...
- Mostly relational operators
- Lazy evaluation, call chaining
- Nicer than hand-rolling SQL (mostly)

dplyr "verbs" & pipes %>%

```
n %>%
 select(first_name, last_name, race_desc, sex,
 birth_age) %>%
 filter(as.integer(birth_age) > 66, sex=="MALE",
 race_desc == "WHITE") %>%
 group_by(first_name) %>%
 summarise(count=n()) %>%
 arrange(desc(count)) %>% head(10) -> old_white_men

print(old_white_men)
```

SQL translation

```
show_query(old_white_men)
```

```
SELECT *
FROM (SELECT `first_name`, `last_name`, `race_desc`,
`sex`, `birth_age`
FROM `ncvoter`)
WHERE ((CAST(`birth_age` AS INTEGER) > 66.0) AND
(`sex` = 'MALE') AND (`race_desc` = 'WHITE'))
```

Whats the advantage of this approach?

dplyr?

- Easy to use, hides huge query complexity
- If things go wrong, debugging is challenging
- Cost/Benefit of additional layers, weigh carefully!

Overview

1. Motivations to use a Database
2. System Scenarios
3. R and Databases State of the Art
4. **Future Directions**

ALTREP

- Luke Tierney, Gabe Becker & Tomas Kalibera
- Abstract vectors, `ELT()/GET_REGION()` methods
- Lazy conversion!

```
static void monetdb_altrep_init_int(DllInfo *dll) {
   R_altrep_class_t cls = R_make_altinteger_class(/* .. */);
   R_set_altinteger_Elt_method(cls, monetdb_altrep_elt_integer);
   /* .. */
}

static int monetdb_altrep_elt_integer(SEXP x, R_xlen_t i) {
   int raw = ((int*) bataddr(x)->theap.base)[i];
   return raw == int_nil ? NA_INTEGER : raw;
}
```

https://svn.r-project.org/R/branches/ALTREP/ALTREP.html#introduction

ALTREP, MonetDBLite & zero-copy

```
library(“DBI”)
con <- dbConnect(MonetDBLite::MonetDBLite(), "/tmp/dscdemo")

dbGetQuery(con, "SELECT COUNT(*) FROM onebillion”)
# 1 1e+09

system.time(a <- dbGetQuery(con, "SELECT i FROM onebillion”))
#    user  system elapsed
#  0.001   0.000   0.001

.Internal(inspect(a$i))
# @7fe2e66f5710 13 INTSXP g0c0 [NAM(2)] BAT #1352 int -> 
integer
```

ALTREP-wrapped MonetDB Column

RIP MonetDBLite

- First in-process embedded analytical DBMS
- on CRAN 2016-2019
- Showed use case for embedded analytics
- Also showed that re-using existing DBMS is rather difficult

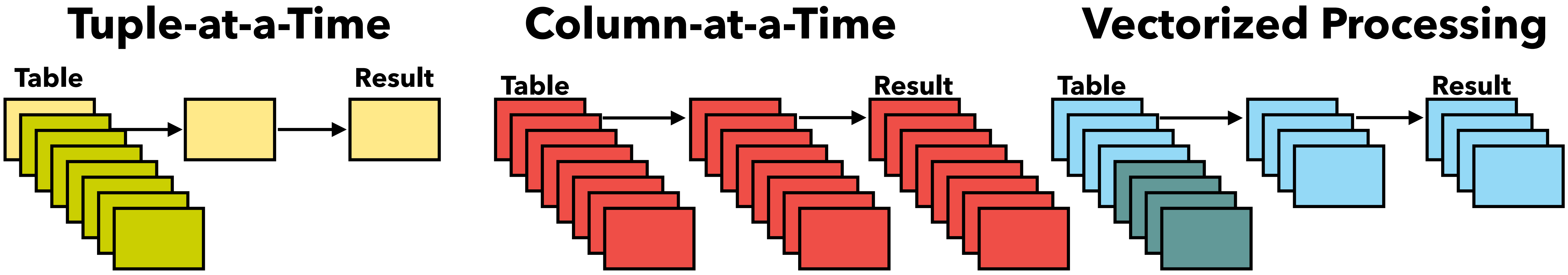
- Open-Source RDBMS created by the CWI Database Architectures research group
- Purpose-built **embedded analytical database**
- No external server management or configuration
- Fast data transfer between R and DuckDB
- Source Code: `https://github.com/cwida/duckdb`

Why DuckDB

- DuckDB is **optimized for analytical use cases**
  - Read-mostly workloads
  - Complex queries, read large parts of the data
  - Bulk appends/updates
- Traditional RDBMS (e.g. PostgreSQL, MySQL, SQLite):
  - Many small writes and updates
  - Simple queries, read only individual rows
- Tight Integration with Analytics in R/Python/...

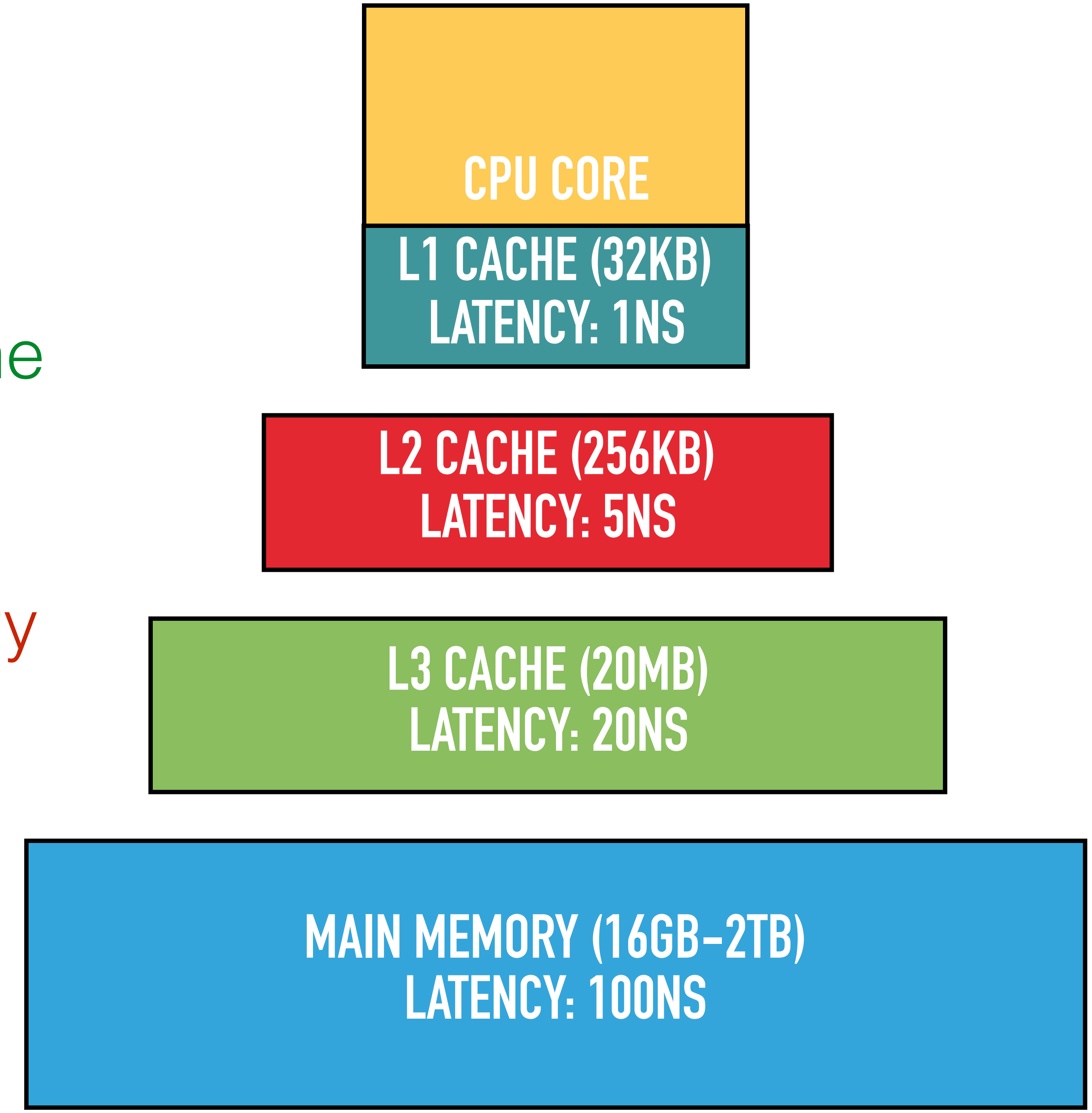
Why DuckDB

- **Vectorized Processing** (DuckDB)
    - Optimized for CPU Cache locality
    - SIMD instructions, Pipelining
    - Small intermediates (fit in L3 cache)



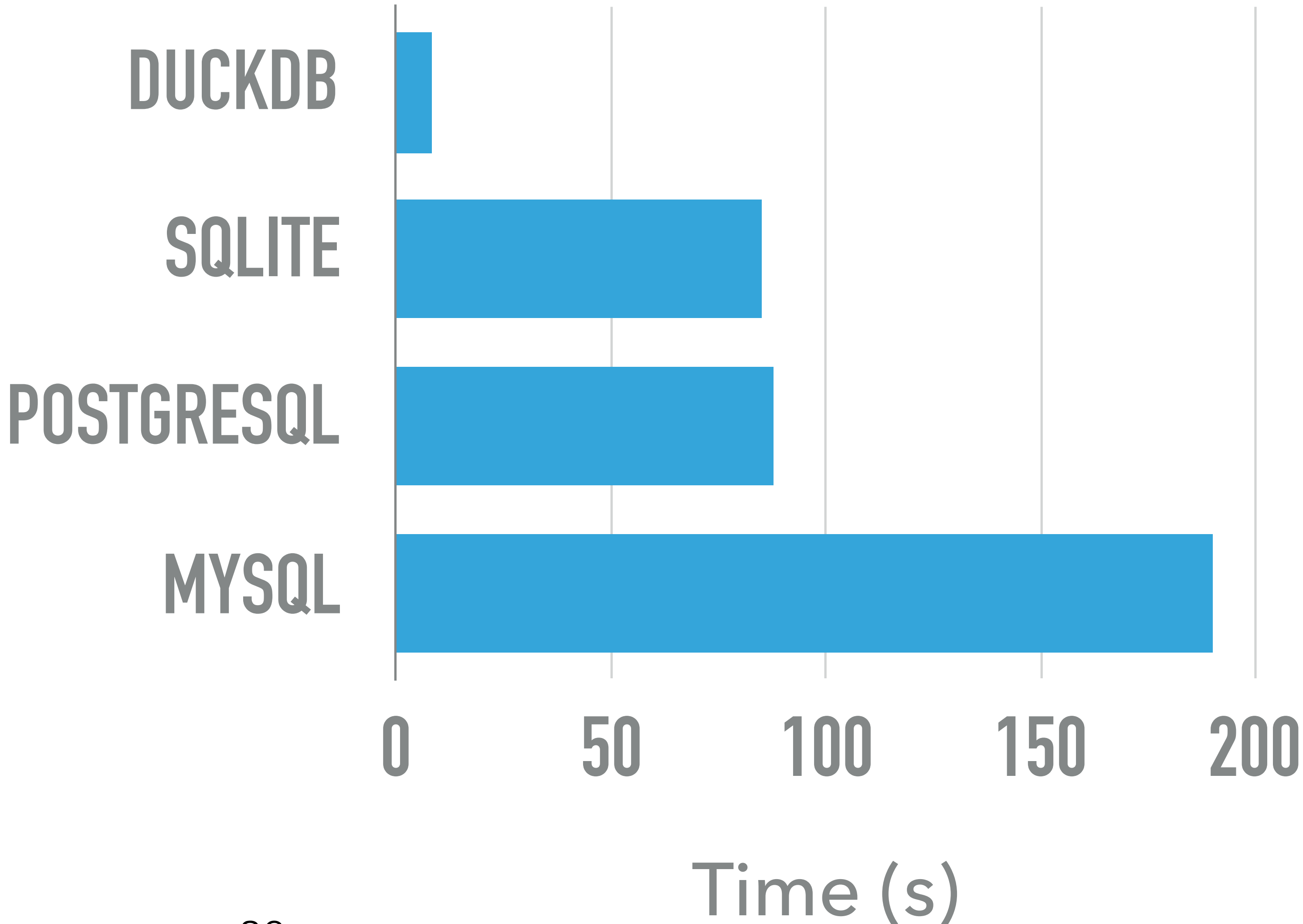
Why DuckDB

- Vectorized Processing
  - Intermediates fit in L3 cache
- Column-at-a-Time
  - Intermediates go to memory



Why DuckDB

- TPC-H Benchmark
  - Analytics benchmark based on shipping company
  - Process 20-40X faster than traditional systems because of processing model



For the adventurous

```
remotes::install_github("cwida/duckdb/tools/rpkg",
 build = FALSE)

con <- dbConnect(duckdb::duckdb(), ":memory:")
```

Overview

1. Motivations to use a Database
2. System Scenarios
3. R and Databases State of the Art
4. The future is DuckDB