In short, osquery turns the metadata and state information of your local system (or remote system(s)) into a SQL-compliant database. It also works on Windows, Linux, BSD and macOS. This means you can query a fleet of systems with a (mostly) normalized set of tables and get aggregated results. Operations and information security staff use this to manage systems and perform incident response tasks, but you can use it to get just about anything and there are even more powerful modes of operation for osquery. But, more on all the features of osquery[r] in another post.

If you are skeptical, here’s some proof (which I need to show regardless of your skepticism state). First, a local “connection”:

library(DBI)

library(osqueryr)

con <- DBI::dbConnect(Osquery())

head(dbListTables(con), 10)

## [1] "account\_policy\_data" "acpi\_tables" "ad\_config"

## [4] "alf" "alf\_exceptions" "alf\_explicit\_auths"

## [7] "alf\_services" "app\_schemes" "apps"

## [10] "apt\_sources"

dbListFields(con, "processes")

## [1] "cmdline" "cwd" "disk\_bytes\_read"

## [4] "disk\_bytes\_written" "egid" "euid"

## [7] "gid" "name" "nice"

## [10] "on\_disk" "parent" "path"

## [13] "pgroup" "pid" "resident\_size"

## [16] "root" "sgid" "start\_time"

## [19] "state" "suid" "system\_time"

## [22] "threads" "total\_size" "uid"

## [25] "user\_time" "wired\_size"

dbGetQuery(con, "SELECT name, system\_time FROM processes WHERE name LIKE '%fire%'")

## # A tibble: 2 x 2

## name system\_time

## 1 Firewall 3

## 2 firefox 517846

then, a remote "connection":

con2 <- osqueryr::dbConnect(Osquery(), host = "hrbrmstr@osq1")

head(dbListTables(con2), 10)

## [1] "account\_policy\_data" "acpi\_tables" "ad\_config"

## [4] "alf" "alf\_exceptions" "alf\_explicit\_auths"

## [7] "alf\_services" "app\_schemes" "apps"

## [10] "apt\_sources"

dbListFields(con2, "processes")

## [1] "cmdline" "cwd" "disk\_bytes\_read"

## [4] "disk\_bytes\_written" "egid" "euid"

## [7] "gid" "name" "nice"

## [10] "on\_disk" "parent" "path"

## [13] "pgroup" "pid" "resident\_size"

## [16] "root" "sgid" "start\_time"

## [19] "state" "suid" "system\_time"

## [22] "threads" "total\_size" "uid"

## [25] "user\_time" "wired\_size"

dbGetQuery(con2, "SELECT name, system\_time FROM processes WHERE name LIKE '%fire%'")

## # A tibble: 1 x 2

## name system\_time

## 1 firefox 1071992

**"You're talking an *awful* lot about the package when you said this was a post on 'standards' and 'consistency'."**

True, but we needed that bit above for context. To explain what this post has to do with "standards" and "consistency" I also need to tell you a bit more about *how* both osquery and the osqueryr package are implemented.

You can read about osquery in-depth starting at the link at the top of this post, but the authors of the tool *really* wanted a consistent idiom for accessing system metadata with usable, normalized output. They chose (to use a word they didn't but one that works for an R audience) a "data frame" as the output format and picked the universal language of "data frames" -- SQL -- as the inquiry interface. So, *right there* are examples of both standards and consistency: using SQL vs coming up with yet-another-query-language and avoiding the chaos of the myriad of outputs from various system commands by making all results conform to a rectangular data structure.

Let's take this one-step further with a specific example. All modern operating systems have the concept of a "process" and said processes have (mostly) similar attributes. However, the commands used to get a detailed listing of those processes differ (sometimes wildly) from OS to OS. The authors of osquery came up with a set of [schemas](https://osquery.io/schema/3.2.4) to ensure a common, rectangular output and naming conventions (note that some schemas are unique to a particular OS since some elements of operating systems have no useful counterparts on other operating systems).

The osquery authors also took consistency and standards to yet-another-level by taking advantage of a feature of SQLite called [virtual tables](https://sqlite.org/vtab.html). That enables them to have C/C++/Objective-C "glue" that gets called when a query is made so they can dispatch the intent to the proper functions or shell commands and then send all the results back -- **or** -- use the SQLite engine capabilities to do joining, filtering, UDF-calling, etc to produce rich, targeted rectangular output back.

By not reinventing the wheel and relying on well-accepted features like data frames, SQL and SQLite the authors could direct all their focus on solving the problem they posited.

**"Um, you're talking alot about everything but R now."**

We're getting to the good (i.e. "R") part now.

Because the authors didn't try to become SQL parser writer experts and relied on the standard SQL offerings of SQLite, the queries made are "real" SQL (if you've worked with more than one database engine, you know how they all implement different flavours of SQL).

Because these queries are "real" SQL, we can write an R DBI driver for it. The DBI package aims *"[to define] a common interface between R and database management systems (DBMS). The interface defines a small set of classes and methods similar in spirit to Perl's DBI, Java's JDBC, Python's DB-API, and Microsoft's ODBC. It defines a set of classes and methods defines what operations are possible and how they are performed."*

If you look at the osqueryr package source, you'll see a bunch of DBI boilerplate code (which is in the r-dbi [organization](https://github.com/r-dbi) example code) and only a handful of "touch points" for the actual calls to osqueryi (the command that processes SQL). No handling of anything but passing on SQL to the osqueryi engine and getting rectangular results back. By abstracting the system call details, R users can work with a familiar, consistent, standard interface and have full access to the power of osquery without firing up a terminal.

*But it gets even better.*

As noted above, one design aspect of osquery was to enable remote usage. Rather than come up with yet-another-daemon-and-custom-protocol, the osquery authors suggest [ssh[](https://www.openssh.com/)](https://www.openssh.com/) as one way of invoking the command on remote systems and getting the rectangular results back.

Because the osqueryr package used the [sys[](https://github.com/jeroen/sys)](https://github.com/jeroen/sys) package for making local system calls, there was only a tiny bit of extra effort required to switch from sys::exec\_internal() to a sibling call in the [ssh[](https://github.com/ropensci/ssh)](https://github.com/ropensci/ssh) package -- ssh::ssh\_exec\_internal() when remote connections were specified. (Said effort could have been zero if I chose a slightly different function in sys, too.)

Relying on well-accepted standards made both osqueryi and the R DBI-driver work seamlessly without much code at all and definitely without a rats nest of nested if/else statements and custom httr functions.

**But it gets *even* more better-er**

Some folks like & grok SQL, others don't. (Humans have preferences, *go figure*.)

A few years ago, Hadley (*do I even need to use his last name at this point in time?*) came up with the idea to have a more expressive and consistent way to work with data frames. We now know this as the tidyverse but one core element of the tidyverse is dplyr, which can really level-up your data frame game (no comments about data.table, or the beauty of base R, please). Not too long after the birth of dplyr came the ability to work with remote, rectangular, SQL-based data sources with (mostly) the same idioms.

And, not too long after that, the remote dplyr interface (now, dbplyr) got up close and personal with DBI. Which ultimately means that if you make a near-fully-compliant DBI interface to a SQL back-end you can now do something like this:

library(DBI)

library(dplyr)

library(osqueryr)

con <- DBI::dbConnect(Osquery())

osqdb <- src\_dbi(con)

procs <- tbl(osqdb, "processes")

listen <- tbl(osqdb, "listening\_ports")

left\_join(procs, listen, by="pid") %>%

filter(port != "", protocol == "17") %>% # 17 == TCP

distinct(name, port, address, pid)

## # Source: lazy query [?? x 4]

## # Database: OsqueryConnection

## address name pid port

## 1 0.0.0.0 BetterTouchTool 46317 57183

## 2 0.0.0.0 Dropbox 1214 17500

## 3 0.0.0.0 SystemUIServer 429 0

## 4 0.0.0.0 SystemUIServer 429 62240

## 5 0.0.0.0 UserEventAgent 336 0

## 6 0.0.0.0 WiFiAgent 493 0

## 7 0.0.0.0 WiFiProxy 725 0

## 8 0.0.0.0 com.docker.vpnkit 732 0

## 9 0.0.0.0 identityservicesd 354 0

## 10 0.0.0.0 loginwindow 111 0

## # ... with more rows

The src\_dbi() call wires up everything for us because d[b]plyr can rely on DBI doing it's standard & consistent job and DBI can rely on the SQLite processing crunchy goodness of osqueryi to ultimately get us a list of really dangerous (if not firewalled off) processes that are listening on all network interfaces. (*Note to self: find out why the BetterTouchTool and Dropbox authors feel the need to bind to 0.0.0.0…*)

FIN

What did standards and consistency get us?

* The osquery authors spent time solving a hard problem vs creating new data formats and protocols
* Rectangular data (i.e. "data frame") provides consistency and structure which ends up causing more freedom
* "Standard" SQL enables a consistent means to work with rectangular data
* ssh normalizes (secure) access across systems with a consistent protocol
* A robust, well-defined standard mechanism for working with SQL databases enabled nigh instantaneous wiring up of a whole new back-end to R
* ssh and sys common idioms made working with the new back-end on remote systems as easy as is on a local system
* Another robust, well-defined modern mechanism for working with rectangular data got wired up to this new back-end with (pretty much) one line of code because of the defined standard and expectation of consistency (and works for local and remote)