Chapter 8: Breaking Up With Your Relational Database

“I call it the law of the instrument, and it may be formulated as follows: Give a small boy a hammer, and he will find that everything he encounters needs pounding.”

Abraham Kaplan, The conduct of inquiry: methodology for behavioral science

It’s an all-too-familiar story. You’ve been faithful companions for years. You knew everything about your partner and came to depend and rely on it for many of your core needs. But, times have changed. Your needs are more nuanced and complex, and you’re starting to have doubts about your relational structure. Your thoughts and queries begin to stray; you survey and index the field and find new, vibrant and exotic options that you never knew of before. And, then, you realize the hard truth: it’s time to break up with your relational database.

Relational databases (RDBMS) have been around since the 1970s when Edgar Codd proposed1 “*a relational model of data for large shared data banks*” as an alternative to network models—heavily linked, on-disk structures—prevalent at that time (so much for ‘big data’ being a 21st century concept). Despite the hype surrounding newer database technologies, relational databases still have quite a bit to offer but should not be the only tool you look to when trying to solve a problem, find “badness” or organize your security data. In this chapter, we’ll explore these newer technologies through security use-cases but also show you how to breathe life into your existing RDBMS relationship.

type="general"

A Primer on SQL/RDMBS Databases

Due to the regular attention given to infosec’s “most wanted”—SQL Injection vulnerabilities—this chapter assumes the reader has some familiarity with traditional RDBMS systems such as MySQL (http://www.mysql.com/downloads/), MariaDB (https://mariadb.org/), Oracle (http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html) or PostgreSQL (http://www.postgresql.org/).

If you are coming at this chapter without prior experience in relational databases you will have an edge up on many readers that have a predisposition towards them, but some of the topics and references could be a bit confusing. This short primer on RDBMS systems should help introduce you to the basic concepts.

Most RDBMS systems have the following core attributes:

**Data is organized by *tables*,** **with *attributes* (*fields*) in *columns* and individual *records* stored in *rows***. For example, an RDBMS table to hold firewall log entries could have a structure that looks like Figure 8.1a with each log entry being a row and the individual data elements broken down into:

* A unique identifier for the firewall (*fwid*)
* A timestamp (*ts*)
* Source IP address (*src\_ip*)
* Source port (*src\_port*)
* Destination IP address (*dst\_ip*)
* Destination port (*dst\_port*)
* Accept/Deny (*action*)
* Number of bytes transferred (*num\_bytes*)

The complete structure of a table or set of tables is called a *schema*.

**Data in tables is referenced by *rows* and *fields*.** Individual fields or combinations of fields called *keys* ensure each record within a table can be uniquely identified and help distinguish the relationships between tables. The firewall and proxy (8.1b) tables in Figure 8.1 are “linked” together by source IP address (*src\_ip*) and both of them are “linked” to the asset database (8.1c) by their *id* fields.

Figure 8.1 [793725c08f01.eps]

Fields can also be part of one or more *indexes,* which are separate data structures that provide optimized ways to organize data in those fields and can dramatically speed up operations that lookup data (*queries*).

**Data is accessed and manipulated through a structured query language** **(*SQL*)**. SQL was designed to be both a human readable and platform independent way to perform insert, update and delete actions, plus run queries against the data. For the example database in Figure 8.1, we can query the destination information (timestamp and IP) for a source IP address in both the proxy and firewall tables with the following SQL statement

**SELECT** ts, dst\_ip

**FROM** proxy\_log\_entry

**WHERE** src\_ip = "10.20.30.40"

**UNION**

**SELECT** ts, dst\_ip

**FROM** fw\_log\_entry

**WHERE** src\_ip = "10.20.30.40";

**Application programs should not rely on the physical structure of the data**. There are a host of options when it comes to deciding how to physically store data in a database and indicating how indexes are organized. All of these choices should be fully abstracted from the application or user who should be able to execute the same high level query and have it work regardless of changes to physical representation.

The relational structure, mostly uniform query language and physical abstraction properties were major contributors to the popularity of SQL databases, especially since mapping problems like customer records and sales orders into fields, and rows is fairly straightforward and just “makes sense”. Yet, as we’ll see later in the chapter, the relational structure is not well suited for all types of data or problems.

Realizing The Container Has Constraints

Compared to Codd’s era, we are awash in computing resources. Memory, storage, CPU and network capacity are all relatively cheap and the need to accommodate the underlying architecture of physical storage when designing, building and using databases is (for the most part) no longer present. Furthermore, becoming an amateur DBA is now as simple as executing “sudo apt-get install mariadb-server” on any Debian-ish Linux box (with similar, easy installation options for Windows and MacOS). In some ways, it is this simplicity and ubiquity that has contributed to the fallacy that traditional SQL/RDBMS databases are destined for extinction due to “lack of scalability and functionality”.

The reality is that modern SQL databases are comparable to web servers, proxy servers, firewalls and mail servers in that their “out of the box” configuration is going to be in “jack of all trades” mode. The default features and capabilities will be enough to get you off and running, and may even perform moderately well as your record counts and schema complexities increase. But, when the types or amounts of data begin to push the boundaries of the default configuration you *will* run into problems. It’s important to understand the most common types of constraints you will face as your SQL needs grow and where to turn when you begin to encounter them.

Constrained By Schema

It may not be obvious at first glance, but there are significant differences between the following two SQL table structures:

**CREATE** **TABLE** `fw1` (

`src` **varchar**(15) **NOT** **NULL**,

`dst` **varchar**(15) **NOT** **NULL**,

`dpt` **int**(11) **NOT** **NULL**,

`d` **int**(11) **NOT** **NULL**)

**CREATE** **TABLE** `fw2` (

`src` **int**(10) **unsigned** **NOT** **NULL**,

`dst` **int**(10) **unsigned** **NOT** **NULL**,

`dpt` **smallint**(5) **unsigned** **NOT** **NULL**,

`d` **date** **NOT** **NULL**)

When creating a table to store “network” information, it’s tempting to use character storage for IP addresses since that’s how we humans interact with them. It’s also tempting to just handle a UNIX timestamp (as seen in the ‘*ts*’ field in Figure 8.1) as a big integer value since, well, that’s what it is. There are, however, potentially significant issues at play with these choices.

If the *src* and *dst* fields are indexed you may not notice any issues at first if all you’re doing is issuing queries for individual IP addresses, like this:

**SELECT** \* **FROM** fw1 **WHERE** src = "10.35.14.16"

The index will speedily find the rows containing the value for *src* and the database engine will return the results as quickly as it can transfer data from disk to your query client. If you do not have an index on those fields, then the same query will have to perform **a full table sequential scan**, which could be a fairly long operation when you have millions of rows.

If, say, you need to find all matching rows for portions of a subnet, you may be faced with creating complex regular expressions (regex) or carving up the IP space into multiple slices to get the benefit of intelligent query prefix optimization for SQL’s “LIKE” operator or split out the subnet into individual IP addresses to ensure you gain the benefit of full speed queries. Non-optimized wildcard searches will, again, result in a full table scan, performing regex string comparisons for every field value.

By switching to the numeric representation of IP addresses (as discussed in Chapter 4), you can gain disk space, memory size and query time efficiency since many index types are optimized for numeric range selections. Converting to/from integers is usually as simple as using built-in INET\_ATON or INET\_NTOA functions. Similarly, moving from a straight integer timestamp to a *date* field brings with it more straightforward query composition and increased query execution speed.

If you regularly work with specialized field types (e.g. IP addresses, geo-location data) you could even consider using different database platforms—such as PostgreSQL—that have direct support for a diverse array of custom fields.

RDBMS schemas also tend to be somewhat fixed structures. While it’s possible to add or remove columns to existing tables, there are real penalties for doing so, both at creation time and beyond. You will immediately incur a space penalty as the new field is added to each row (whether necessary or not) with that operation also occupying a decent amount of time on large, established table structures. Some RDBMS systems are able to compensate for these issues, but you may need to leave your “amateur DBA” status at the door as you start to become a professional database administrator in order to solve these issues.

type="general"

You got some ‘EXPLAINin’ to do!

To become a true database wizard requires delving into the dark arts of the subject matter. SQL queries are a bit like magic spells in that the wrong inflection can drastically change the results (usually for the worse). You can get an idea of how to tweak your schemas and optimize your queries with the EXPLAIN statement, available in most RDBMS systems.

EXPLAIN will, well, *explain* what the query engine will do with the SQL you’ve given it without executing it. For example, if we were to load the AlienVault database mentioned in Chapter 4 into a simple SQL database, it might look like this:

MariaDB> **DESCRIBE** avrep;

+--------+---------------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+--------+---------------------+------+-----+---------+-------+

| ipn | int(10) | YES | MUL | NULL | |

| bad | tinyint(3) unsigned | YES | | NULL | |

| con | tinyint(3) unsigned | YES | | NULL | |

| type | varchar(50) | YES | | NULL | |

| cc | varchar(2) | YES | | NULL | |

| city | varchar(30) | YES | | NULL | |

| latlon | varchar(30) | YES | | NULL | |

+--------+---------------------+------+-----+---------+-------+

To get a count of all IP addresses coming from China (CN), you might issue the following query:

MariaDB> **SELECT** **COUNT**(ipn) **FROM** avrep **WHERE** cc="CN";

You can see how optimal that query is (or isn’t) by prefixing it with EXPLAIN (we’ve added the EXTENDED and \G to make the output clearer for the book’s printed format):

**EXPLAIN** **EXTENDED**

-> **SELECT** **COUNT**(ipn) **FROM** avrep **WHERE** cc="CN"\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: avrep

type: ref

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 265597

Extra: Using where

For this query, no keys are being used, so this will require a table scan. You can optimize it by adding an index on the *cc* field:

**CREATE** **INDEX** cc\_idx **ON** avrep (cc);

and re-run EXPLAIN:

**EXPLAIN** **EXTENDED**

-> **SELECT** **COUNT**(ipn) **FROM** avrep **WHERE** cc="CN"\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: avrep

type: ref

possible\_keys: **cc\_idx**

key: **cc\_idx**

key\_len: **5**

ref: **const**

rows: **132798**

filtered: **100.00**

Extra: Using where

to see if there are any changes. In this case, the EXPLAIN output shows that the SQL query engine identified the index for the CC field and that using it will reduce the number of rows scanned.

It’s a good idea to use EXPLAIN on more complex queries, especially ones that may be run often. You may be able to identify bottlenecks that you are attributing to “those old SQL databases” when it’s really your schema or SQL composition that needs work.

Constrained By Storage/RAM

When this book hits the shelves in 2014, consumers will have access to 5TB hard drives. With that type of capacity being a general user commodity it’s difficult to contemplate how a database could be constrained by storage given that enterprise-class disks have even more options through larger and faster disks and disk arrays. Open source SQL databases such as MySQL or MariaDB can have individual tables as large as 256TB, which will fit comfortably on, say, a BTRFS (https://btrfs.wiki.kernel.org/index.php/Main\_Page) filesystem capable of holding 16EiB of data. What, then, are these storage/RAM “constraints”?

* **Speed**. If your analytics needs are modest, it’s tempting to stick with consumer-grade equipment for both cost and ease of deployment. However, that 5400RPM USB 2.0 disk may get quite long-in-the-tooth for even modestly sized projects given the way consumer drives are designed (since they aren’t expecting to serve database workloads). You *could* use consumer disks in a consumer storage array, but you’re only temporarily masking the problem. If your analytics workflow performance starts to degrade, consider investing in faster disks with increased cache. Plus, if the impacts are severe enough, it may be time to switch to true commodity *server* hardware with faster enterprise-class storage—or even solid-state disks (SSD)—and a proper industrial-class storage array.
* **Caching**. Databases use both disk and RAM in concert when performing most of their operations. Delving into RAM and cache discussions can stir up as much debate in the DBA community as sparking a similar conversation about desktop signature anti-virus in the defender community. Increasing the amount of RAM *will* help your database perform faster, especially when you need to issue the same query more than once (think a nested SELECT query used in multiple, but diverse main SELECT statements). RAM and disk caching will also help when inserting data into a database since write-caching can be employed to mask I/O bottlenecks.
* **Capability.** Just because you *can* store alottabytes in a table doesn’t mean you *should*. For example, storing three years of enterprise firewall log data in a single table *is* possible, but it’s truly a bad idea. By optimizing the underlying storage configuration and using table partitioning techniques available in most modern RDBMS systems, you can turn what may have been a marathon of a query into a sprint and probably still keep everything on one system.

Constrained By RAM

Lack of sufficient active RAM or using a traditional RDBMS with a configuration that cannot take advantage of large amounts of RAM is the harbinger of doom for any project that needs to scale. As indicated in the previous section, databases use RAM to (among other things) cache portions of tables that are on disk and also to cache query results. More advanced SQL databases can also use RAM for **in-memory tables**. If you know you’re going to have regular use of referential data (e.g. asset metadata, non-frequently changing IP lists), loading that information into an in-memory SQL table can reap huge rewards as you perform JOINs, UNIONs and sub-SELECTs, and it’s usually as simple as just identifying the query—which can be the full set of rows and fields from an existing table—you want to populate in an in-memory configuration. For example, if you wanted to store all the IP addresses contained in the AlienVault table in an in-memory table (to guarantee it stays there vs rely on the cache keeping it there) you could do the following:

**CREATE** **TABLE** avrep\_mem **ENGINE**=**MEMORY**

-> **SELECT** ipn **AS** ip

-> **FROM** avrep;

It’s best to avoid consumer-grade RAM and opt for high quality ECC (error-correcting code) memory to avoid the perils of data corruption.

Constrained By Data

There are definitely examples of “security data” that fit well into the relational model including firewall logs, web server logs, anti-malware logs and asset information. Each of those example sources easily maps into interconnected rows and columns. But, what about the JSON structure of an incident recorded in VERIS as seen in Chapter 6? While it’s *possible* to develop a relational structure for this data, it’s hardly an optimal solution.

To optimize efficiency, Codd came up with the notion of database **normalization**, which is just a way of describing a method to organize fields and tables to eliminate as many redundancies as is feasible and make it easier to modify or extend the database schema with as little impact as possible. “Over-normalizing” a database can make working with the underlying data awkward and complex. “Under-normalizing” a database can increase the complexity of the application code or database stored procedures and will—most likely—needlessly expand the size of your data store.

Normalizing tabular data that is designed to fit into tables is generally a straightforward task. Truly mapping and normalizing hierarchical data (like the JSON VERIS data) means converting the hierarchies into graph adjacency lists, materialized paths or nested sets that definitely increase query complexity. You could always go halfway and limit the nesting by storing large chunks of the JSON tree as BLOBs (binary large objects) in special fields, but that also makes queries complex **and** slow, since you’ll likely be performing full text searches of those fields.

RDBMS systems are great for a wide variety of problem sets and data types, but they should not be the only tool in your toolbox since there are so many custom options available, as we’ll see in the next section.

Exploring Alternative Data Stores

There are many longstanding and new database storage and database management systems that have shunned the conventions and conformity of SQL. These technologies are usually grouped under the term **NoSQL** (Not only SQL), which makes it easier to classify them, but also adds confusion since they can be radically different from each other. By “not being SQL” they offer alternate ways of designing solutions and storing information that can be of huge benefit when incorporating data analysis into your security strategy. We’ll take a look at some of the more prominent ones and sneak in a security use case or two along the way to give you an idea of where you might want to pick one over the other.

BerkeleyDB

Perl wonks will no doubt be familiar with Berkeley DB (BDB) and you can find support for it in R (RBerkeley), Python (pybsddb) and most other scripting/programming languages. BDB is a local (embedded) **key/value** store that does what the description suggests: lets you identify a *key* and store arbitrary data associated with it, then perform highly efficient lookups with the *key*. By it’s own definition, it’s neither a relational database, object-oriented database, network database or a database sever. It’s completely value-agnostic.

If you’ve ever worked with the default configuration of SpamAssassin () or postfix () or dealt with open source LDAP servers such as OpenLDAP (), you’ve encountered BDB.

Key/value stores perform well in situations where writes are infrequent but reads are potentially plentiful: i.e. *caches*. Consider, once again, the IPv4 address space. If you only needed to cache certain attributes of an IP address (e.g. geolocation data, reputation data) and only needed local resources, choosing BDB as your platform has some serious merit. It doesn’t have the overhead that comes with traditional RDBMS databases (though modern versions of BDB “speak” SQL) and can be optimized for the key and value data structures. Plus, the keys and values can also be language independent (i.e. you can populate BDB stores with R and read them with Python, or vice-versa). Here’s a very basic example of storing IP geolocation data with R and reading the same data back with Python:

*# R code to interface with BDB*

library(RBerkeley)

*# create and open BDB database*

dbh <- db\_create()

db <- db\_open(dbh, txnid = NULL, file = "av.db", type = "BTREE", flags = mkFlags(DB\_CREATE, DB\_EXCL))

*# store geolocation data*

db\_put(dbh, key = charToRaw("24.62.253.107"), data = charToRaw("43.2555,-70.8829"))

*# read it back to show it works*

coords <- rawToChar(db\_get(dbh, key = charToRaw("24.62.253.107")))

db\_close(dbh) *# close BDB db*

print(coords)

# [1] "43.2555,-70.8829"

*// Python code to interface with BDB*

from bsddb3 import db

import struct

import socket

*// initialize and open BDB database*

av\_db = db.DB()

av\_db.open('av.db',None,db.DB\_BTREE, db.DB\_DIRTY\_READ)

*// get first key/value pair*

cursor = av\_db.cursor()

av\_rec = cursor.first()

*// print it out to show it worked*

print av\_rec

// ('24.62.253.107', '43.2555,-70.8829')

av\_db.close() *// close BDB file*

It would be very straightforward to expand this example to, say, store entire AlienVault database, indexed by IP address with the other associated fields stored in the value component.

Berkeley DB also has solid thread support and scales as large as 256TB. If your workloads can deal with disk-seek times, you do not want the hassle of maintaining a server process or multi-node infrastructure for your caches and there’s a chance you need multi-platform and multi-language support, it’s definitely a good choice.

type="tip"

BDB Alternatives

Oracle is now the proprietor of Berkeley DB. While it’s still provided under a GNU AGPL v3 license, Oracle also offers a commercial version with fairly steep licensing options. If you are concerned that this may become fully commercial in the future, there are alternatives that provide the same feature set, including:

* Kyoto Cabinet (<http://fallabs.com/kyotocabinet/>)
* MapDB (<http://www.mapdb.org/faq-general.html>)

Redis

It’s tempting to think of Redis () as just a server-version of a key/value store since that’s what it looks like on the surface with it’s most basic commands, GET and SET and it’s basic data type being a binary safe string (so you can store virtually any type of data in the key or value components). What Redis *really* is, however, is more of an in-memory **data structure** **server** that is also persisted on disk (that also has many other useful features). The in-RAM requirement should not be glossed over lightly since every data structure and element **must** fit into RAM for Redis to work. This constraint should help prevent you from trying to shoehorn large relational or hierarchical structures into Redis (since that’s definitely not what it’s designed for).

Redis operates as a data structure server by providing a framework of operations for four fundamental data storage types: **lists**, **hashes**, **sets** and **sorted sets**.

*Lists* store single binary safe strings that are either pushed on to the front (LPUSH) or back (RPUSH) of the list. Lists make superb message queue structures and excel at keeping the “last *n*” number of items available.

*Hashes* expand the key/value NoSQL model by providing a way to identify and manipulate fields within the value component in a very space-efficient manner. We can replicate the geolocation Berkeley DB geolocation example quite easily with Redis hashes straight from the Redis command line interface:

redis> HMSET ip:24.62.253.107 lon 43.2555 lat -70.8829 zip 03878

redis> HMGET ip:24.62.253.107 lon lat

1) "43.2555"

2) "-70.8829"

The main difference here is that you can query this database server from any client on the network versus be constrained by just local file access.

*Sets* store non-repeating collections of binary safe strings. This makes them ideal for associating elements together for quick membership determination. For example, creating a “workstations” set and populating the members with IP addresses makes it trivial to determine whether an IP address you’ve seen in a packet is coming from a workstation node:

redis> SADD workstations "10.23.34.45"

redis> SADD workstations "10.32.43.54"

redis> SADD workstations "10.45.34.32"

redis> SADD workstations "10.34.23.45"

redis 127.0.0.1:6379> SISMEMBER workstations "10.10.10.10"

(integer) 0 #not in set

redis 127.0.0.1:6379> SISMEMBER workstations "10.23.34.45"

(integer) 1 #in setredis

*Sorted sets* provide a means to associate a raked value with a member of a set. You could, then create risk or reliability sets for each of the malicious host types in the AlienVault database, using the values from those fields or keep a running count of times you’ve seen those known-bad hosts attempt to access your resources (or when *your* resources have attempted to access those bad ones).

type="tip"

Advanced Redis Features

Redis supports **partitioning** which lets you use memory on other systems to hold portions of Redis data structures. This is similar to the way you can partition tables in MariaDB, MySQL and Oracle and helps you get around single-system memory constraints.

Redis also has a built-in **publish-subscribe** service. With it, you can create a number of clients that subscribe to a channel that is publishing log entries or just new, individual IP addresses that are on your internal “suspicious” list. When that new value is pushed, each client will get the message and can take some type of action, like running a set of analytics routines.

There is robust Redis support in Python and R and the API is very straightforward to work with. Say you want a centralized and efficient way to know whether you’ve seen an IP address in an indicator of compromise (IoC) you’ve received. Rather than rely on a query to return from your centralized log management system, setup a workload that takes IP addresses from the log streams and stores them in a centralized Redis simple key/value or hash data structure with as much metadata as you need. Here’s a Python example of how to “watch” a log file (in this case, a web server log) and store the data in Redis:

*# log watcher/Redis importer*

import time

import re

import redis

import pickle

*# setup regex to parse web log entries*

logparts = r'(\S+) (\S+) (\S+) \[(.\*?)\] "(\S+) (\S+) (\S+)" (\S+) (\S+)'

logpart = re.compile(logparts)

*# map field names to extracted regex values*

def field\_map(dictseq,name,func):

for d in dictseq:

d[name] = func(d[name])

yield d

*# extract data from weblog*

def web\_log(lines):

groups = (logpart.match(line) for line in lines)

tuples = (g.groups() for g in groups if g)

colnames = ('host','referrer','user',

'datetime','method', 'request',

'proto','status','bytes')

log = (dict(zip(colnames,t)) for t in tuples)

log = field\_map(log,"bytes",

lambda s: int(s) if s != '-' else 0)

log = field\_map(log,"status",int)

return log

# "tail" for python

def follow(thefile):

thefile.seek(0,2)

while True:

line = thefile.readline()

if not line:

time.sleep(0.1)

continue

yield line

# setup log watching

logfile = open("/var/log/nginx/web.access.log")

loglines = follow(logfile)

log = web\_log(loglines)

# setup Redis connection

red = redis.StrictRedis(host='redhost.example.com',

port=6379, db=0)

# for each entry, store pythonic-data structure in

# associated with a key (could also use Redis hash

# for more language-independence)

for line in log:

l = line['host']

a = red.get("ip:%s" % l)

if (a == None):

a = {}

a['ls'] = time.time()

a['ct'] = 1

red.set("ip:%s" % l,pickle.dumps(a))

else:

a = pickle.loads(a)

a['ls'] = time.time()

a['ct'] += 1

red.set("ip:%s" % l,pickle.dumps(a))

And, here’s the query component:

# query script

from datetime import datetime

import redis

import pickle

import sys

# setup Redis connection

red = redis.StrictRedis(host='localhost', port=6379, db=0)

# get IP address from the command line & query Redis

ipaddr = sys.argv[1]

ioc = red.get("ip:%s" % ipaddr)

# if found

if (ioc != None):

b = pickle.loads(ioc)

print("IP [%s] was last seen on [%s].\nTotal times seen ")

print("since we started counting: [%d]." %

(ipaddr, datetime.fromtimestamp(b['ls']),b['ct']))

else:

print("%s has not been seen, yet." % ipaddr)

Now, it’s quick work from the command line to know whether you’ve seen an IP address:

dds$ lastseen "24.62.253.107"

IP [24.62.253.107] was last seen on [2013-10-13 18:57:59.875430].

Total times seen since we started counting: [80787].

If you’re thinking, “I could just use grep”, remember that this is a constantly streaming, online activity from potentially hundreds or thousands of sources spanning weeks or months. If you architect it properly, Redis will always beat “grep”.

Hive

It’s virtually impossible to write a book about data analysis without mentioning “Hadoop” (http://wiki.apache.org/hadoop), and if you’re already investigating or using Hadoop then you may have come across Hive (http://wiki.apache.org/hadoop/Hive/LanguageManual). Hive sits on top of the Hadoop Distributed file System (HDFS) (http://hadoop.apache.org/docs/stable/hdfs\_user\_guide.html) that partitions data across—potentially—*thousands* of nodes. Hadoop **MapReduce** jobs execute across these nodes using this data. The *map* component takes a set of data elements, breaks them into key/value pairs and performs a comparison and/or computation on them. The *reduce* component takes these results, combines and “sifts” them to come up with a final result set (which may involve another comparison and/or computation).

Hive provides a SQL-like interface to this HDFS data. Rather then becoming an expert Java coder to compose and execute MapReduce jobs, Hive abstracts this complexity and converts SQL into MapReduce jobs for you. This is a very important point to remember. In the Hadoop ecosystem, *everything* boils down to a MapReduce job across very large amounts of data. The complexities of setting up a Hadoop environment and *keeping* it running are mixed into the cost/benefit analysis when choosing this as part of your analytics platform. Furthermore, while Hive provides the comfort of SQL, some key features of SQL do not come along for the ride. For example, the Hive query language (HiveQL) provides only limited support for SQL JOINs. If your needs go beyond combining tables on equality conditions, you cannot use Hive due to the limitations of the Hadoop MapReduce paradigm. You also need to use caution when ordering result sets with SQL’s ORDER BY, since Hive currently only uses a single reduce engine to perform that sorting task, creating potential bottlenecks. There are many other subtleties to Hive and HiveQL as well. While you may not need to become a Java expert, you will have to thoroughly understand how HiveQL queries translate to MapReduce jobs and learn how to optimize queries to take advantage of this platform.

If you have the time, space, budget *and* use-cases to setup Hadoop/HDFS/Hive, then it may be well worth the investment. Imagine being able to keep a full year’s archive of every log file from every system, network device, firewall and mail server online in a massively efficient data warehouse and perform basic inquiries across all of those components. *That’s* where the real power of Hive+Hadoop lies.

type="tip"

MongoDB

ElasticSearch

FEATURE ON MARIADB

FEATURE ON COMMERCIAL DATABASES VS OPEN SOURCE

In Summary

Becoming a truly effective as a security data scientist will require a shift in mindset from any monolithic relational database fidelity you may have. Solving real problems will require you to keep your options open, recognizing each database technology has unique benefits for specific tasks.

For Further Reading

Relational Database Design Clearly Explained, Second Edition (The Morgan Kaufmann Series in Data Management Systems) ISBN-13: 978-1558608207 Jan L. Harrington.

References

1Codd, Edgar Frank. "A relational model of data for large shared data banks." Communications of the ACM 26, no. 1 (1983): 64-69.