TE color code: GREEN is checked OK, YELLOW needs attention, RED (here) is an error that needs to be corrected.

[AU: Global for whole chapter, if any color is needed for the code in this chapter, please add it. If you have questions, please contact me right away. Thanks, Kevin (PjE)]

[AU: And please remember that if the code is changed, make the appropriate change in the code download as well and submit a new version of the code download for the chapter with your AR. Thanks, Kevin (PJE)]

Chapter 8: Breaking Up With Your Relational Database

[AU: In your original proposal for this chapter wasn’t one of the use cases to be a discussion of storing and accessing netflow data. What happened to that? Why is it not included in the chapter? Thanks, Kevin (PjE)

]

[[copy edited by Kezia Endsley]]

AR: Jay & I discussed it and the chapter was already diverse and complex enough without trying to walk folks through a full Hadoop installation and incorporation of NetFlow stream ingestion and analyses. It would really belong in almost a whole book as we thought about it more. I included a discussion of it in the Hadoop section just to give folks an idea of what it takes to do such processing.

“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*

[AU: Without being gratuitous, if you can think of any illustrations/figures that might be useful in the chapter and make use of this being a four-color book, that would be a useful addition. Thanks, Kevin (PJE)]

AR: Added color (subtle) to fig 1 and added a figure for Hadoop/MapReduce.

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 proposed “*a relational model of data for large shared data banks*” as an alternative to the network models—heavily inter-linked, on-disk structures—prevalent at that time.

[AU: I made a note out of the reference. Change okay? Thanks, Kevin (PjE)]

AR: note is cool. Didn’t dawn on me to do it that way.

type="note"

Codd proposed this in 1970 in "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, Vol. 13, No. 6, pp. 377-387. 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. However, they 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.

If you plan on typing the code from the chapter versus executing each snippet from the ch08.R or ch08.py source files you will need to download the data files in the ch08/data directory from the repository on the book’s web site for many of the listings to work correctly. You will also need to run the code in Listings 8-0 and 8-1 to setup your R and Python environment (respectively) for the code examples in this chapter.

*# Listing 8-0*

*# This code sets up the R environemnt for the chapter*

*# set working directory to chapter location*

*# (change for where you set up files in ch 2)*

setwd("~/book/ch08")

*# make sure the packages for this chapter*

*# are installed, install if necessary*

pkg <- c("RBerkeley")

new.pkg <- pkg[!(pkg %in% installed.packages())]

if (length(new.pkg)) {

install.packages(new.pkg)

}

*# Listing 8-1*

*# This code sets up the python environemnt for the chapter*

*# set working directory to chapter location*

*# (change for where you set up files in ch 2)*

setwd("~/book/ch08")

*# make sure the packages for this chapter are installed*

For the Python examples, you will need to ensure the proper libraries are installed. First, if you did install Python with Canopy (see Chapter 2), you’ll need to refer to this knowledge base article (<https://support.enthought.com/entries/23389761-Installing-packages-into-Canopy-User-Python-from-the-OS-command-line>) on the Enthought support site, which will enable you to install Python packages from external sources. You will also need a working Redis server before installing the Python redis component. Refer to the Redis quickstart quide (<http://redis.io/topics/quickstart>) for information on how to get Redis up and running. The following code sets up the necessary environment from a typical Debain-style system shell prompt:

dds$ *# install the Berkeley DB library*

dds$ sudo apt-get install libdb-dev

dds$ *# install Python package for Berkeley DB interface*

dds$ *# note that you may not need sudo depending on your environment*

dds$ sudo pip install bsddb3

dds$ *# install Python package for Redis interface*

dds$ sudo pip install redis

type="general"

A Primer on SQL/RDMBS Databases

Due to the regular attention given to InfoSec’s “most wanted”—SQL Injection (SQLi) 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 who have a predisposition toward 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 (*fw\_id*)
* 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*)

[[Author: I changed fwid in first bullet to fw\_id (underscore), per figure 8--okay? Kezia]]

AR: indeed. thx.

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 (Figure 8-1a) and proxy (Figure 8-1b) tables are “linked” together by source IP address (*src\_ip*) and both of them are “linked” to the asset database (Figure 8-1c) by their *id* fields.

Figure 8-1: Graphical representation of example firewall, proxy, and asset database tables [793725 c08f001.eps]

[[Author: Can you please add a descriptive caption for the figure above. Kezia]]

AR: done. Thx.

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, you 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 you’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. 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, this simplicity and ubiquity 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, simple SQL table structures:

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

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

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

dpt **int** **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**)

For those who may be new to SQL, the statement in in the first code block creates a database table with IP address src and dst fields stored as a string of characters (“0” … “9”), while the statement in the second block creates a table with those fields stored as an unsigned integer with a display width of 10 characters.

[AU: The TE added the preceding paragraph. Is it okay? Thanks, Kevin (PjE)]

AR: aye. I modified added some detail and fixed the last bit.

When you are creating a table to store “network” information, it’s tempting to use character storage for IP addresses since that’s how 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. Also, destination TCP/UDP ports (dpt) *technically* are integers. 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 you needed 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 you might have to split out the subnet into individual IP addresses to ensure you gain the benefit of full speed queries. Non-optimized wildcard searches—especially ones without a common prefix—will result in a full table scan, performing regex string comparisons for every field value.

[AU: Addition below okay referring back to the table okay? Please revise if it’s not clear or doesn’t refer to what you want it to refer to. Thanks, Kevin (PJE)]

AR: totally cool. Thx.

By switching to the numeric representation of IP addresses (shown in the preceding code for table fw2 and 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. Finally, switching dpt from an integer to a smallint will save you two bytes per record which can be important if you plan on using in-memory tables or start racking up billions of records.

If you regularly work with specialized field types (for example, IP addresses and geolocation 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. Although it’s possible to add or remove columns to/from 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 with that operation (whether necessary or not) also occupying a decent amount of time on large, established table structures. Some RDBMS systems can 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’ve 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 you 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 rerun 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 darn old school SQL databases” when it’s really your schema or SQL composition that needs work.

Constrained by Storage

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 “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 be the bottleneck for even modestly sized projects, given the way consumer drives are designed (since they aren’t designed to serve database workloads). You *could* use consumer disks in a consumer storage array, but this would only temporarily mask the problem. If your analytics workflow performance declines significantly when you increase the size of data sets, 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 antivirus 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 a huge quantity of data in a single table doesn’t mean you *should*. For example, storing 3 years of enterprise firewall log data in a single RDBMS table *is* possible, but it’s truly a bad idea because of all the performance problems this causes. 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 (for example, 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. It’s usually as simple as 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 versus 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 also 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 format, as seen in Chapter 7? Although it’s *possible* to develop a relational structure for this data, it’s hardly an optimal solution.

[AU: That should be Ch. 7, shouldn’t it? Thanks, Kevin (PjE)]

AR: Darnit, yeah. Thought I got that reference right. I think I looked at an old ToC. I fixed it to be 7. Apologies.

To optimize database table structures and query efficiency, Codd came up with the notion of *normalization*, which is just a way of describing a method to organize fields and tables so as 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. 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 you’ll see in the next section.

type="general"

Who/What Is This ‘Maria’?

Many readers may have used or come into contact with the MySQL RDBMS. For many years, it was a foundational element of the initial “LAMP” (Linux/Apache/MySQL/PHP) stack of components you would use to build websites. After Oracle acquired MySQL, there was a community-developed fork of the code created under the name “MariaDB.” MariaDB is a drop-in replacement for MySQL. You can uninstall MySQL (preserving data, of course) and install MariaDB and everything will “just work.”

MariaDB versioning and features have been on par with counterpart MySQL releases, but significant divergence is occurring with newer iterations, including support for cutting-edge storage engines, dynamic columns, and interfacing with NoSQL environments (Cassandra).

Choosing MariaDB over MySQL, PostgreSQL, or traditional commercial RDBMS offerings is a decision you and your security and analytics team members must make yourselves and may be highly dependent on corporate requirements, if you’re constrained by them. Even if you “can’t” use MariaDB, it’s definitely a project that should be on your watch list.

Exploring Alternative Data Stores

There are many longstanding and new database storage and database management systems that have shunned the conventions and conformity of straight-laced 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 the features and functionality each provides can be radically different. By “not being SQL” they all offer alternate ways of designing solutions and storing information that can be of huge benefit when incorporating data analysis into your security strategy. This section takes a look at some of the more prominent ones and sneaks in a security use case or two along the way to give you an idea of when you might want to pick one over the other.

BerkeleyDB

Perl wonks will no doubt be familiar with Berkeley DB (BDB) (http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html), and you can find support for it in R (RBerkeley), Python (pybsddb), and most other scripting/programming languages. BDB is a local (that is, embedded) *key/value* store that does what the description suggests: it lets you identify a *key* and store arbitrary data associated with it, and then perform highly efficient lookups with the *key*. By its own definition, it’s not a relational database, an object-oriented database, a network database, or a database server. Unlike keys and fields in RDBMS systems, BDB is completely value-agnostic.

If you’ve ever worked with the default configuration of SpamAssassin (http://spamassassin.apache.org/) or postfix (http://www.postfix.org/) or dealt with open source LDAP servers such as OpenLDAP (http://www.openldap.org/), you’ve encountered BDB.

Key/value stores perform well in situations where data writes are infrequent but reads are potentially plentiful, for example, *caches*. Consider, once again, the IPv4 address space. If you needed to cache only certain attributes of an IP address (for example, geolocation data or reputation data) and needed only 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 (that is, you can populate BDB stores with R and read them with Python, or viceversa). Listing 8-2 shows a very basic example of storing IP geolocation data with R:

*# Listing 8-2*

*# requires packages: RBerkeley*

*# 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"

type="note"

Note that R will return a warning message about the database handle being unusable after the db\_close() function call. This is just an informative message and can be ignored.

TE: Though I get the proper output, I also get this error message:

Warning message:

In db\_close(dbh) : 'db' handle may not be accessed again

AR: thx. That’s normal, but I made a note for folks

Listing 8-3 shows a similar example of reading the same data back with Python:

*# Listing 8-3*

*# Requires: bsddb3 and Berkeley DB library*

*# 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*

TE: The comment character is incorrect – should be “#” rather than “/”

I also get an error message that “No module named bsddb3”. The Enthought Canopy distribution does not include it. There is no “bsddb3” package available for installation in the Canopy package manager.

AR: fixed.

It would be very straightforward to expand this example to store the entire AlienVault database, indexed by IP address and 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-programming language support, it’s definitely a good choice.

type="general"

BDB Alternatives

Oracle is now the proprietor of Berkeley DB. Although 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

Redis is an open source, BSD licensed, advanced key-value store (http://redis.io/). 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. Its most basic commands are GET and SET, and its basic data type is 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 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. You could 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 differences here are that you can query this database server from any client on the network versus be constrained by just local file access and that everything is in memory, so lookups will be almost instantaneous.

* **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> SISMEMBER workstations "10.10.10.10"

(integer) 0 **// not in set**

redis> SISMEMBER workstations "10.23.34.45"

(integer) 1 **// in set**

TE: there seems to be a statement/command missing, to change the prompt to: redis 127.0.0.1:6379>

AR: my bad. I wanted to keep the prompt generic since 127.0.0.1:6379 part is not going to be fully deterministic on end-user systems

* **Sorted sets** provide a means to associate a ranked value with a member of a set. You could create risk or reliability sets for each of the malicious host types in the AlienVault database, using the values from those fields. You could also 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).

[AU: Should that be “ranked” above? Just checking. Thanks, Kevin (PjE)]

AR: aye. Ranked. Fixed. Thx.

type="general"

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 RAM 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 make their way on to your internal “suspicious” list. When any new value is pushed, each client will get the message and can take some type of action, like running a set of analytics routines or parsing and storing the information into multiple SQL and NoSQL data stores for later processing.

There is robust Redis support in Python (redis-py) and R (rredis), 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 from some external source. Rather than rely on a query to return from your clunky 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. Listing 8-4 provides a Python example of how to “watch” a log file (in this case, a web server log) and store the data in Redis.

type="note"

Listings 8-4 and 8-5 will work better as standalone shell scripts (each in their own file, as directed in the comments for each listing) versus within the Canopy environment. You will also need to have a web server running. To fully mimic the examples, you can install nginx (the one used in the 8-4 example) via sudo apt-get install nginx at a shell prompt and start it with sudo /etc/init.d/nginx start to generate output for the logs.

*# Listing 8-4*

*Web server l*

*# Save this as "watcher.py"*

*# Start it in one shell window prompt with*

*# python watcher.py*

*# Requires: Python redis package*

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*

*# change this to an active, accessible web server log*

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

loglines = follow(logfile)

log = web\_log(loglines)

*# setup Redis connection*

*# for large environments, you will substitute*

*# localhost with a dedicated server host name*

red = redis.StrictRedis(host='localhost',

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))

TE: I can’t run this because the free Canopy distribution does not come with py-redis, and it costs $200 to subscribe for 1 year to Canopy basic to get it.

[AU: Please confirm all this is good to go for sure since the TE is unable to confirm. Thanks, Kevin (PJE)]

AR: I added code and notes in the setup portion of the chapter plus a feature note here. Thx.

Listing 8-5 shows the query component:

*# Listing 8-5*

*# Redis log watcher python query script*

*# Save this as "lastseen.py"*

*# Start it in one shell window prompt with*

*# python query.py*

*# Requires: Python redis package*

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 (substitute 24.62.253.107 with a known address to get a “found” result in your setup):

dds$ python lastseen.py 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 http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.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 and combines them to come up with a final result set (which may involve another comparison and/or computation).

type="general"

MapReduce Redux

MapReduce is a Google creation (<http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/mapreduce-osdi04.pdf>) and was designed to enable efficient computations across huge (e.g. multi-thousand node) clusters. It does this by splitting the data across the entire cluster then instructing worker nodes to perform some operation on that local data set (the *map*). Those intermediary results are then collected and summarized by other worker nodes into a final operation (the *reduce*). Figure 8-2 illustrates the process.

Figure 8-2 Illustration of MapReduce [793725 c08f002.eps]

[AU: Have you already introduced a thumbnail definition of what MapReduce is? Will this audience automatically know? Include a quick one if not. Thanks, Kevin (PjE)]

AR: done. Thx.

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.

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.

type="general"

Analyzing “At-Scale” NetFlow Data With Hadoop

If you ever enter a conversation about data and Hadoop, the concepts of *volume, velocity,* and *variety* will inevitably come up. *Volume* refers to how much data you have. *Velocity* refers to how fast that data is coming in or being analyzed. *Variety* speaks to the diversity of data types being ingested and processed. Most security data falls somewhere on the lower end or middle section of each of those spectrums. However, even in a medium-sized network, NetFlow data can easily peg all the way to the upper bounds on the *velocity* and *volume* scales.

If you aren’t familiar with NetFlow, here’s the definition straight from RFC 3954 (<http://www.ietf.org/rfc/rfc3954.txt>):

A *flow* is defined as a unidirectional sequence of packets with some common properties that pass through a network device. These collected flows are exported to an external device, the *NetFlow collector*. Network flows are highly granular; for example, flow records include details such as IP addresses, packet and byte counts, timestamps, Type of Service (ToS), application ports, input and output interfaces, etc.

NetFlow data is extremely useful for security analytics, but can be challenging to work with. For example, if you have a 10 Gbps link that is only 50% utilized, you can expect to churn out *2.3TB of NetFlow data* ***per hour***. This is definitely a job for Hadoop since the input stream gathering and storage functions can be distributed across a large cluster (since it would overwhelm a single host) and converted on the fly to work with Hadoop native file formats. Then, you can begin to design MapReduce jobs such as performing anomaly detection or analyzing DDoS captures for patterns.

Tools such as PacketPig (<http://hortonworks.com/blog/big-data-security-part-one-introducing-packetpig/>) can help reduce some of the tedium involved in getting NetFlow data into an environment where analysis can be performed, but it cannot abstract the complexity of such an environment. You will need to thoroughly understand numerous NoSQL technologies if you wish to head down the path of analyzing NetFlow data at-scale.

If you have the time, space, personnel, budget, *and* use cases to set up Hadoop/HDFS/Hive, it may be well worth the investment. Imagine being able to keep a full year’s online archive of every log file from every system, network device, firewall, and mail server 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="general"

What about HBase, Cassandra, Pig …?

The full Hadoop ecosystem continues to expand at a relentless pace. Advancements within the environment itself (for example, Hadoop 2.0) as well as integration with the environment (for example, Cassandra, MongoDB) and unique vendor-specific offerings are introducing nascent alternatives that have their own strengths, tradeoffs, and idiosyncrasies.

You will need to spend some effort looking at all the options you have available and mapping them to your perceived needs. Then choose a direction and stick with it. A Hadoop analytics environment—much like Rome—cannot be built in a day. Despite the continuing advancements, this ecosystem is far from mature and you will be forging new ground over a long period of time with each step you take.

MongoDB

MongoDB (http://www.mongodb.org/) could be called the “MySQL of NoSQL” databases as it has a large and active community, is easy to deploy in development, and scales fairly well in production. At its core, Mongo provides a way to do extremely quick prototyping given the schema-less nature of the platform. Unlike traditional SQL databases, where you need to define the fields you will be using up front, Mongo lets you start with a basic pseudo-schema and refine your needs along the way.

type="note"

To follow along with these examples, install MongoDB via sudo apt-get install mongodb at a shell prompt and then start it with sudo /etc/init.d/mongodb start.

For example, it’s very straightforward to start storing IP geolocation info from the AlienVault reputation database for an IP address with the following commands starting at a Linux shell prompt:

dds$ mongo

> db.av.insert ( { ip:"193.147.49.42",

geo:"40.4085,-3.6921" })

> db.av.find({ ip:"193.147.49.42" })

{ "\_id" : ObjectId("525bfbe02074bfa7aaad8316"),

"ip" : "193.147.49.42",

"geo" : "40.4085,-3.6921" }

and then choose to add other information later, like the type of malicious activity the host is engaged in:

> db.av.update ( { ip:"193.147.49.42" },

{ $set : { maltype:"Scanning Host" } } )

> db.av.find({ ip:"193.147.49.42" })

{ "\_id" : ObjectId("525bfbe02074bfa7aaad8316"),

"geo" : "40.4085,-3.6921",

"ip" : "193.147.49.42",

"maltype" : "Scanning Host" }

You do pay a price for these incremental field updates given the way Mongo stores the data and manages the on-the-fly schema changes, and you may need to dump and reload the database to regain storage and query efficiency if you perform these types of changes in production versus just experiment during development.

Mongo breathes JSON and uses binary JSON (BJSON) in API calls. This means you need to be comfortable with JavaScript notation and will definitely want to keep the JSONLint (http://jsonlint.com/) URL handy to assist you when errors crop up in your input data. The use of JSON provides the capability of storing deeply nested or hierarchical records and structures, which will require you to re-think any notions you may have on normalization. If you’re used to performing RDBMS normalization, then you’ll need to take a step back, ignore most of what you’ve been taught or have learned and embrace the verbosity of this side of the NoSQL universe.

For example, malicious nodes in the AlienVault database can have multiple malicious activities associated with them. In traditional, normalized SQL, you would likely setup a separate table with host key and malicious node type field and have a row for each entry:

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

|193.147.49.42|Scanning Host|

|193.147.49.42|Spamming |

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

then, perform a JOIN when retrieving results. With Mongo, you would store those components as a JSON array within the record:

> db.av.update ( { ip:"193.147.49.42" },

{ $set : { maltype:[ "Scanning Host","Spamming" ] } } )

It may be difficult to see the value of this additional complexity with such a trivial example, but the power this holds starts to become much clearer if you look back at the VERIS JSON data in Chapter 6. Creating a normalized table structure to store all the fields in an incident is possible, but not necessary given Mongo’s ability to efficiently store, process and query complex field structures. If you have Mongo installed from the previous example, you can install the git tool via sudo apt-get install git, which will allow you to follow the steps in Listing 8-6 (found in other/ch08.sh) download and import the complex incident data in the entire VERIS Community Database from their github repository (<https://github.com/vz-risk/>) in about 5 minutes, without the need to create a database or table schema ahead of time.

*# Listing 8-6*

*# Retrieve VCDB files, import into mongo and perform a query*

*# Requires mongodb and git*

*# clone the VCDB github repository*

dds$ git clone https://github.com/vz-risk/VCDB.git

*# import all the incdients*

dds$ cd VCDB/incidents

dds$ ls | head -5

0012CC25-9167-40D8-8FE3-3D0DFD8FB6BB.json

002599D4-A872-433B-9980-BD9F257B283F.json

005C42A3-3FE8-47B5-866B-AFBB5E3F5B95.json

0096EF99-D9CB-4869-9F3D-F4E0D84F419B.json

00CC39F6-D2E0-4FF4-9383-AE3E28922015.json

dds$ for f in \*.json ; do \

mongoimport -d veris -c public --jsonArray $f ;

done

*# find all financial firms with security incident in the VCDB*

*# 52 is NAICS code for financial firms*

dds$ echo 'db.public.find({"victim.industry": { $regex : "^52" } },

{ "victim.victim\_id" : 1, \_id : 0 } )' | mongo veris

{ "victim" : [

{ "victim\_id" : "Blue Cross & Blue Shield of Rhode Island" } ] }

{ "victim" : [

{ "victim\_id" : "Group Health Incorporated" } ] }

{ "victim" : [

{ "victim\_id" : "Delta Dental of Pennsylvania" },

{ "victim\_id" : "ZDI" } ] }

{ "victim" : [

{ "victim\_id" : "UK National Health Service" } ] }

{ "victim" : [

{ "victim\_id" : "Mundo.com" },

{ "victim\_id" : "Public Defender of Venezula" },

{ "victim\_id" : "Caroni Seguros SA" } ] }

…

TE: I don’t have an environment to test the code above, and don’t know enough about MongoDB to validate this.

[AU: Please make sure to confirm this code given the TE is not familiar with it. Thanks, Kevin (PjE)]

AR: done.

We’ve highlighted instances where Mongo has understood some incidents have multiple victims.

[AU: When you insert your color code be sure to let me know what needs to be highlighted so that this statement will be true. Thanks, Kevin (PjE)]

If your record count is large enough to span multiple Mongo nodes, these simple queries will work unaltered. Mongo can also perform data aggregation or even run MapReduce jobs across a whole cluster, mimicking some of the functionality of both Hadoop and more traditional SQL databases.

Mongo can also be used as a tool in your data acquisition and cleanup processes, where you may have traditionally used built-in structures in your programming or scripting languages. For example, log processing is one of the less glamorous activities of security data analysis. They come in all shapes and sizes and some, like Cisco’s IronPort e-mail logs, require extra processing to get into a form useful for analytics. Take a look at the following sample:

*# Listing 8-7*

*# Example of an IronPort log file*

Fri Oct 18 11:05:01 2011 Info: Start MID 346564 ICID 1042862

Fri Oct 18 11:05:01 2011 Info: MID 346564 ICID 1042862 From:

<dave@example.com>

Fri Oct 18 11:05:01 2011 Info: MID 346564 ICID 1042862 RID 0 To:

<steve@test.com>

Fri Oct 18 11:05:01 2011 Info: MID 346564 Message-ID

'<112067.438985349-em02@steel>'

Fri Oct 18 11:05:01 2011 Info: MID 346564 Subject ‘TPS Reports Due'

Fri Oct 18 11:05:02 2011 Info: MID 346564 ready 864 bytes from

<dave@example.com>

Fri Oct 18 11:05:02 2011 Info: MID 346564 matched all recipients for

per-recipient policy local domains in the outbound table

Fri Oct 18 11:05:03 2011 Info: MID 346564 interim AV verdict using

Sophos CLEAN

Fri Oct 18 11:05:03 2011 Info: MID 346564 antivirus negative

Fri Oct 18 11:05:03 2011 Info: MID 346564 DLP no violation

Fri Oct 18 11:05:03 2011 Info: MID 346564 queued for delivery

Fri Oct 18 11:05:03 2011 Info: Delivery start DCID 178987 MID 346564

to RID [0]

Fri Oct 18 11:05:04 2011 Info: Message done DCID 178987 MID 346564

to RID [0]

Fri Oct 18 11:05:04 2011 Info: MID 346564 RID [0] Response 'ok:

Message 10569973 accepted'

Fri Oct 18 11:05:04 2011 Info: Message finished MID 346564 done

[[Author: Changed a few of the single quotes above to "straight" ones, as needed in code. Kezia]]

AR: Thx. I wish Word wouldn’t insert them automatically

Because Mongo allows incremental schema build out, you can use that feature to create records for each message (MID) as you parse the log file. You can then add fields as you go, ending up with a final, complete database and an idea of how a complete per-record schema might look. The Mongo entry for the previous record could look like the one found in Listing 8-8.

*# Listing 8-8*

*# Example of an IronPort log file translated to JSON via MongoDB*

{

mid : "346564",

icid : "1042862",

from : "dave@example.com",

to : "steve@test.com",

messageID : "112067.438985349-em02@steel",

subj: "TPS Reports Due",

bytes: "864"

matchStatus : 1,

delivered : 1,

av : { engine : "Sophos", verdict: "CLEAN" },

dlp : { violation : "none" },

start : "Fri Oct 18 11:05:01",

finish : "Fri Oct 18 11:05:01"

}

Once all the records have been created, you can use Mongo and Python or R to perform time series analysis, z-scaled anomaly detection, clustering, or a host of other analyses.

type="general"

Why Not Use Mongo for Everything?

It’s possible to fall into the trap of trying to use Mongo for everything, especially since it allows you to be a bit lazy up front. Although it’s great for some tasks, the platform still has some rough edges at the time this chapter was written. You might want to take into account when deciding on Mongo for a project:

* Record counting operations are improving but are still slower than other database platforms due to the way Mongo uses the underlying b-tree database file structures.
* Field names are not compressed and take up real space *per-record*. This leads to practices such as using sip instead of src\_ip or sourceIP and u for “username”, making queries somewhat unreadable unless you’re extremely familiar with the data.
* Maintenance operations are still required and can impair operations. You *will* need to compact the database regularly and this can be a time-consuming, blocking operation across a whole cluster. Although this is most likely not a problem for your analytics environment, be careful if you’re using Mongo to present an interactive data interface to other users.
* By default, writes to a Mongo database work a bit like UDP packets in that it’s “send, and pray it’s received.” You need to explicitly set options for enabling “write concern” to get more TCP-like behavior. This can have a serious impact on performance such as the need to guarantee writes of log entries you are aggregating into Mongo.

Special Purpose Databases

It’s far too easy to get snarled on what truly constitutes a “database”. For those still entrenched in the SQL world, NoSQL is a serious affront to their sensibilities. For those who’ve adjusted to the NoSQL paradigm, tools such as ElasticSearch (http://www.elasticsearch.org/) and Neo4j (http://www.neo4j.org/) may be equally as world jarring.

Databases will be an essential element in your analytics workflow, which might look like this:

[AU: Should the following be a numbered list? Are these items in order since they are a workflow? Thanks, Kevin (PjE)]

AR: Muscle memory always has me going to ListBulleted. Fixed. Thx.

1. Identify the data sources (for example, logs, traditional databases, alerts)
2. Collect, transform (if needed), and store the data
3. Query the data store
4. Provide analytics on the results

If you choose to work with raw SQL or NoSQL databases, then you will need to perform most of the setup and cleanup tasks on your own, which requires DBA-like intimacy with the underlying database platforms.

ElasticSearch for Logs

If you’re focused on the goal of analytics more than the journey of how to get your data there, you may be interested in tools like ElasticSearch that abstract the complexities of the backend and give you an input, query, and analytics interface to work with on the front end.

ElasticSearch consumes practically anything you give it and provides straightforward ways to ask it questions and get data out of it. You just need to feed it semi- or unstructured data and fold in some domain intelligence to enable smart indexing. It works its multi-node NoSQL magic in conjunction with a layer of full-text searching to give you almost instantaneous query results even for large amounts of data. It’s *highly* geared toward log data and supports an aggregation framework similar to that of Mongo.

If you are analyzing a wide variety of logs in your security work, ElasticSearch may be something you should consider investigating.

Neo4j for “Connections”

As indicated in several previous chapters, many areas of information security analytics involve looking at connections between nodes. You’ve also seen how network graph structures can make working with these connections a bit easier. Although it’s possible to model graph structures in SQL databases or Mongo, Redis, and so on, it’s easier to use a something like Neo4j that provides direct support for network graph models and operations.

If the igraph operations in Chapter 4 intrigued you, then you’ll be even more impressed with the feature set in Neo4j because it essentially scales similar computations and analytics across millions or billions of nodes. You can import high level vertex + edge connection data into Neo4j from NetFlow sources, firewall, proxy, e-mail, and DNS logs and augment the connection and node information with detail data from each of those sources.

You still need to have a graph model in mind when you’re designing for Neo4j and will need to learn a new graph-specific query language—Cypher—to get work done. However, many fine-grained tasks will require that you either roll up your sleeves and code a bit in Java or Python or use the Neo4j REST interface to funnel query output into your analytics platform of choice.

Summary

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

This chapter has presented a survey of various technologies combined with small examples in many different types of SQL and NoSQL database environments. We’ve outlined strengths and weaknesses in the choices you have and even provided some counseling on how to enhance interactions with your traditional SQL stores.

We’ve focused on some core database offerings, but have not provided an exhaustive reference since that would be a book on its own. You will need to keep abreast of developments in the database space—both SQL and NoSQL—to see where you may need to make adjustments in the future. If you are working with larger and larger amounts of data, it may be time to wade a bit deeper into the Hadoop ecosystem, provided you understand the level of commitment required and the constraints you will be facing.

Finally, you’ve seen that databases can take many forms, that they can be used as a means to an end (for example, log parsing) as well as an end in and of themselves.

Recommended Readings

The following are some recommended readings that can further your understanding on some of the topics we touch on in this chapter. For full information on these recommendations and for the sources we cite in the chapter, please see Appendix B.

***Relational Database Design Clearly Explained, Second Edition,* by Jan L. Harrington**—One of the most complete and accessible resources available; especially helpful to nascent arrivals to the world of RDBMS systems.

***Professional Hadoop Solutions* by Boris Lublinsky, Kevin T. Smith, and Alexey Yakubovich**—An excellent and thorough introduction to the Hadoop ecosystem with modern, real-world examples and advice on how to secure your Hadoop analytics environments.

***Professional NoSQL* by Shashank Tiwari**—Far more comprehensive reference on NoSQL database technologies that digs a bit deeper into many of the options described in this chapter.

[AU: Please put the full reference for that source below (and the recommended readings above) into the references appendix. Thanks, Kevin (PJE)]

AR: we will do that.