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 on-disk linked 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

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 for a firewall log entry could have an RDBMS structure that looks like Figure 8.1(a) 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 identify the relationships between tables. The firewall and proxy (b) tables in Figure 8.1 are “linked” together by source IP address (*src\_ip*) and both of them are “linked” to the asset database (c) by their *id* fields.

Figure 8.1 [793725c08f01.eps]

Fields can also be part of one or more *indexes,* which can help 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 for a source IP address in both the proxy and firewall tables with the following SQL statement

**SELECT** fw\_log\_entry.ts **AS** fw\_ts,

proxy\_log\_entry.ts **AS** pxy\_ts,

dest\_url, dest\_ip, dest\_port

**FROM** proxy\_log\_entry, fw\_log\_entry

**ORDER** **BY** fw\_ts, pxy\_ts;

**Application programs do (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 choose 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 or operation 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 options for Windows and MacOS). In some ways, it is this simplicity 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 very similar 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 features and capabilities will be enough to get you off and running, and may even perform moderately well as your record counts and schema complexity 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 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:

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

The index will speedily find the values 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 need to find all matching rows for portions of a subnet, you may be faced with creating complex regular expressions 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 result in a full table scan, performing regex-like 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. 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 work with specialized field types (e.g. IP addresses, geo-location data) quite a bit, 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 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 problems.

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 even larger and faster disks/disk arrays. Even open source SQL databases such as MySQL or MariaDB can have individual tables as large as 256TB, which will fit comfortably on, say, a ZFS filesystem capable of holding 16EB of data. What, then, are these “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 operations.

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. Databases use RAM to 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.

Constrained By Data

There are definitely examples of “security data” that fit well into the relational model including firewall logs, web server logs and asset information. Each of those example sources easily maps into 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.

Exploring Alternative Data Stores

There are many longstanding and new database management systems that have shunned the conventions and conformity of SQL

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.