Wrapping Up

We believe the future of data management lies in the polyglot persistence model (using more than one database in a project)—while the worldview of the general-purpose RDBMS fog drifts away.

Let’s take this opportunity to see where our seven databases fit together in the greater database ecosystem.

By this point, we have explored the details of each and mentioned a few commonalities and differences.

We’ll see how they contribute to the vast and expanding landscape of data storage options.

9.1 Genres Redux

We’ve seen that how databases store their data can be largely divided into five genres: relational, key-value, columnar, document, and graph.

Let’s take a moment and recap their differences and see what each style is good for and not so good for—when you’d want to use them and when to avoid them.

Relational

This is the most common classic database pattern. Relational database management systems (RDBMSs) are set-theory-based systems implemented as two-dimensional tables with rows and columns.

Relational databases strictly enforce type and are generally numeric, strings, dates, and uninterpreted blobs

Good For:

Because of the structured nature of relational databases, they make sense when the layout of the data is known in advance but not how you plan to use that data later may not be.

Or, in other words, you pay the organizational complexity up front to achieve query flexibility later.

Many business problems are aptly modeled this way, from orders to shipments and from inventory to shopping carts.

You may not know in advance how you’ll want to query the data later—how many orders did we process in February?—but the data is quite regular in nature, so enforcing that regularity is helpful.

Not-So-Good For:

When your data is highly variable or deeply hierarchical, relational databases aren’t the best fit.

Because you must specify a schema up front, data problems that exhibit a high degree of record-to-record variation will be problematic.

Consider developing a database to describe all the creatures in nature.

Creating a full list of all features to account for (hasHair, numLegs, laysEggs, and so on) would be intractable.

In such a case, you’d want a database that makes less restrictions in advance on what you can put into it.

Key-Value

The key-value (KV) store was the simplest model we covered.

KV maps simple keys to (possibly) more complex values like a huge hashtable.

Because of their relative simplicity, this genre of database has the most flexibility of implementation.

Hash lookups are fast, so in the case of Redis, speed was its primary concern.

Hash lookups are also easily distributed, and so Riak took advantage of this fact for focusing on simple-to-manage clusters.

Of course, its simplicity can be a downside for any data with complex modeling requirements.

Good For:

With little or no need to maintain indexes, key-value stores are often designed to be horizontally scalable, extremely fast, or both.

They’re particularly suited for problems where the data are not highly related.

For example, in a web application, users’ session data meet this criteria; each user’s session activity will be different and largely unrelated to the activity of other users.

Not-So-Good For:

Often lacking indexes and scanning capabilities, KV stores won’t help you if you need to be able to perform queries on your data, other than basic CRUD operations (Create, Read, Update, Delete).

Columnar

Columnar databases (aka *column-oriented*, aka *column family*) share many similarities with both KV and RDBMS stores.

Like with a key-value database, values are queried by matching keys.

Like relational, their values are groups of zero or more columns, though each row is capable of populating however many it wants.

Unlike either, columnar databases store like data by columns, rather than keeping data together by rows.

Columns are inexpensive to add, versioning is trivial, and there is no real storage cost for unpopulated values.

We saw how HBase is a classic implementation of this genre.

Good For:

Columnar databases have been traditionally developed with horizontal scalability as a primary design goal.

As such, they’re particularly suited to “Big Data” problems, living on clusters of tens, hundreds, or thousands of nodes.

They also tend to have built-in support for features such as compression and versioning.

The canonical example of a good columnar data storage problem is indexing web pages.

Pages on the Web are highly textual (benefits from compression), somewhat interrelated, and change over time (benefits from versioning).

Not-So-Good For:

Different columnar databases have different features and therefore different drawbacks.

But one thing they have in common is that it’s best to design your schema based on how you plan to query the data.

This means you should have some idea in advance of how your data will be used, not just what it’ll consist of.

If data usage patterns can’t be defined in advance—for example, fast ad hoc reporting—then a columnar database may not be the best fit.

Document

Document databases allow for any number of fields per object and even allow objects to be nested to any depth as values of other fields.

The common representation of these objects is as JavaScript Object Notation (JSON), adhered to by both MongoDB and CouchDB—though this is by no means a conceptual requirement.

Since documents don’t relate to each other like relational databases, they are relatively easy to shard and replicate across several servers, making distributed implementations fairly common.

MongoHQ tends to tackle availability by supporting the creation of datacenters that manage huge datasets for the Web.

Meanwhile, CouchDB focuses on being simple and durable, where availability is achieved by master-master replication of fairly autonomous nodes.

There is high overlap between these projects.

Good For:

Document databases are suited to problems involving highly variable domains. When you don’t know in advance what exactly your data will look like, document databases are a good bet.

Also, because of the nature of documents, they often map well to object-oriented programming models.

This means less impedance mismatch when moving data between the database model and application model.

Not-So-Good For:

If you’re used to performing elaborate join queries on highly normalized relational database schemas, you’ll find the capabilities of document databases lacking.

A document should generally contain most or all of the relevant information required for normal use.

So while in a relational database you’d naturally normalize your data to reduce or eliminate copies that can get out of sync, with document databases, denormalized data is the norm.

Graph

Graph databases are an emerging class of database that focuses more on the free interrelation of data than the actual values.

Neo4j, as our open source example, is growing in popularity for many social network applications.

Unlike other database styles that group collections of like objects into common buckets, graph databases are more free-form—queries consist of following edges shared by two nodes or, namely, *traversing* nodes.

As more projects use them, graph databases are growing the straightforward social examples to occupy more nuanced use cases, such as recommendation engines, access control lists, and geographic data.

Good For:

Graph databases seem to be tailor-made for networking applications.

The prototypical example is a social network, where nodes represent users who have various kinds of relationships to each other.

Modeling this kind of data using any of the other styles is often a tough fit, but a graph database would accept it with relish.

They are also perfect matches for an object-oriented system.

If you can model your data on a whiteboard, you can model it in a graph.

Not-So-Good For:

Because of the high degree of interconnectedness between nodes, graph databases are generally not suitable for network partitioning.

After an initial search users typically want to review the results using a form of graph visualization tool. Users will frequently then select one or more vertices of interest and ask to load more vertices that may be connected to their current selection. In graph-speak, this operation is often called "spidering" or "spidering out".

Spidering the graph quickly means you can’t afford network hops to other database nodes, so graph databases don’t scale out well.

It’s likely that if you use a graph database, it’ll be one piece of a larger system, with the bulk of the data stored elsewhere and only the relationships maintained in the graph.

Making a Choice

Data is the new oil.

We sit upon a vast ocean of data, yet until it’s refined into information, it’s unusable (and with a more crude comparison, there’s a lot of money in data these days).

The ease of collecting and ultimately storing, mining, and refining the data out there starts with the database you choose.

Deciding which database to choose is often more complex than merely considering which genre maps best to a given domain’s data.

Though a social graph may seem to clearly function best with a graph database, if you’re Facebook, you simply have far too much data to choose one.

You are more likely going to choose a “Big Data” implementation, such as HBase or Riak.

This will force your hand into choosing a columnar or key-value store.

In other cases, though you may believe a relational database is clearly the best option for bank transactions, it’s worth knowing that Neo4j also supports ACID transactions, expanding your options.

These examples serve to point out that there are other avenues beyond genre to consider when choosing which database—or databases—best serve your problem scope.

As a general rule, as the size of data increases, the capacity of certain database styles wane.

Column-oriented datastore implementations are often built to scale across datacenters and support the largest “Big Data” sets, while graphs generally support the smallest.

This is not always the case, however

Riak is a large-scale key-value store meant to shard data across hundreds or thousands of nodes, while Redis was built to run on one—with the possibility of a few master-slave replicas or client-managed shards.

There are several more dimensions to consider when choosing a database, such as durability, availability, consistency, scalability, and security.

You have to decide whether ad hoc queryability is important or if mapreduce will suffice.

Do you prefer to use an HTTP/REST interface, or are you willing to require a driver for a custom binary protocol?

Even smaller scope concerns, such as the existence of bulk data loaders, might be important for you to think about.

To simplify the comparison between these databases, see the table.

The table is not meant to be an exhaustive list of features.

Instead, it’s meant to be a tool to quickly compare these databases we’ve already covered.

Note the versions of each database.

These features change in the blink of an eye, so we highly recommend double-checking these values for more recent versions.

Where Do We Go from Here?

Modern application scaling problems now fall largely in the realm of data management

We’ve reached a point in application evolution where programming language, framework, and operating system choice—even hardware and operations (thanks to virtual machine hosts and “the cloud”)—are becoming so cheap and easy as to become largely trivial problems driven as much by preference as necessity. e.g. lambda functions on AWS

If you want to scale your application in this age, you should think quite a bit about which database, or databases, you choose— it’s more than likely your true bottleneck.

The next steps from here are to pursue in detail the databases that piqued your interest or continue learning about other options like Cassandra, Drizzle, or OrientDB.