InfoWorld

SEPTEMBER 13, 2017

GET TECHNOLOGY RIGHT®

INSIDER

NoSQL standouts: The best key-value databases

Aerospike, Hazelcast, Memcached, Microsoft Azure Cosmos DB, and Redis put different twists on fast and simple data storage

Most any application needs some form of persistence—a way to store the data outside of the application for safekeeping. The most basic way is to write data to the file system, but that can quickly become a slow and unwieldy way to solve the problem. A full-blown database provides a powerful way to index and retrieve data, but may also be overkill. Sometimes all you need is a quick way to take a freeform piece of information, associate it with a label, stash it somewhere, and pull it back out again in a jiffy.

Enter the key-value store. It's essentially a database, but one with a highly specific purpose and a deliberately constrained design. Its job is to let you take data (a "value"), apply a label to it (a "key"), and store it either in-memory or in some storage system that's optimized for fast retrieval. Applications use key-value databases for everything from caching objects to sharing commonly used data among application nodes.

Many relational databases can function as key-value stores, but that's a little like using a tractor-trailer to go on grocery runs. It works, but it's dramatically inefficient, and there are far less top-heavy ways to solve the problem. A key-value store provides just enough infrastructure for simple value storage and retrieval, integrates more directly with applications that use it, and scales in a more granular way with the application workload.

Here we've examined five widely used products (including one cloud service) that are explicitly billed as key-value databases, or which offer key-value storage as a central feature. All have their differences. Hazelcast and Memcached tend toward minimalism, and don't even bother to back the data in question on disk. Aerospike, Cosmos DB, and Redis are fuller featured, but still revolve around the key-value metaphor.

See the table to compare features. Read on for brief discussions of each database.

	Aerospike	Hazelcast IMDG	Microsoft Azure Cosmos DB	Memcached	Redis
Platforms	LWMO	Java	Cloud-only	LWMO	LWMO
Current version	3.14.1.1	3.8	N/A	1.5.1	4.0.1
Initial release	2012	2008	2017	2003	2009
License	AGPL	Apache 2	Proprietary	BSD	BSD
Disk-backed	Yes	No	Yes	No	YesBSD
Clustering	Yes	Yes	Yes	No	Yes
Sharding/partitioning	Yes	Yes	Yes	No	Yes
Native scripting	Yes	Java	Yes	No	Yes
Transactions	Per key	Yes	Yes	No	Yes
Embeddable	Yes*	Yes	No	Yes*	Yes*

Key: L=Linux, W=Windows, M=MacOS, S=Solaris, I=iOS, A=Android, O=Other,
*=Through a third-party implementation.

Aerospike

If Redis is Memcached on steroids, Aerospike could be said to be Redis on steroids. Like Redis, Aerospike is a key-value store that can operate as a persistent database or a data cache. Aerospike is designed to be easy to cluster and easy to scale, the better to support enterprise workloads.

Much in Aerospike echoes both other key-value stores and other NoSQL databases. Data is stored and retrieved by way of keys, and the data can be kept in a number of fundamental data types including 64-bit integers, strings, double-precision floats, and raw binary data serialized from a number of common programming languages.

Aerospike also can store data in complex types—lists of values, collections of key-value pairs called maps, and geospatial data in the GeoJSON format. Aerospike can perform native processing on geospatial data—e.g., determine which locations stored in the database are closest to each other by just performing a query—making it an attractive option for developers of applications that rely on location

Data stored in Aerospike can be organized into a number of hierarchical containers. Each kind of container lets you set different behavioral properties on the data inside it. For instance, the topmost level of containers, namespaces, determines whether the data will be stored on disk or in RAM or both, whether the data is replicated within the cluster or across clusters, and when or how data is expired or evicted. Through namespaces, Aerospike allows developers to keep the most frequently accessed data in memory for the fastest possible response.

Aerospike can keep its data on most any filesystem, but it has been written specifically to take advantage of SSDs. That said, don't expect to drop Aerospike on any old SSD and expect good results. Aerospike's developers maintain a list of approved devices, and they have created a tool, ACT, to rate the performance of SSD storage devices under Aerospike workloads.

Aerospike, like most NoSQL systems, uses a shared-nothing architecture for the sake of replication and clustering. Aerospike has no master nodes and no manual sharding. Every node is identical. Data is randomly distributed across the nodes and automatically rebalanced to keep bottlenecks from forming. If you want to, you can set rules for how aggressively data is rebalanced. Multiple clusters, running in different network segments or even different datacenters, can be configured to synchronize against one another.

Like Redis, Aerospike allows developers to write Lua scripts, or UDFs (user-defined functions), that run inside the Aerospike engine. UDFs can be used to read or alter records, but they are best used as a way to perform high-speed, read-only, map-reduce operations across collections or "streams" of records on multiple nodes..

Hazelcast IMDG

Hazelcast comes billed as an "in-memory data grid," essentially a way to pool RAM and CPU resources across multiple machines to allow data sets to be distributed across those machines and manipulated in-memory. Hazelcast can be used as a key-value store, and, according to its makers, as an alternative to products like Pivotal Gemfire, Software AG Terracotta, or Oracle Coherence.

Hazelcast is built with Java and has a Java-centric ecosystem. Each node in a Hazelcast cluster runs an instance of Hazelcast's core library, IMDG, on the JVM. The way Hazelcast works with data is also closely mapped to Java's language structures. Java's Map interface, for instance, is used by Hazelcast to provide key-value storage. As with Memcached, nothing is written to disk; everything is kept in-memory at all times.

Hazelcast can be run as a distributed service or embedded directly inside a Java application. Clients are currently available for Java, Scala, .Net, C/C++, Python, and Node.js, and one for Go is in the works.

Hazelcast clusters have no master/slave setup; everything is peer-to-peer. Data is automatically sharded and distributed across all members of the cluster. One benefit Hazelcast can provide in a distributed environment is "near cache," where commonly requested objects are migrated to the server making the requests. This way, the requests can be performed directly in-memory on the same system, without requiring a round trip across the network.

Aside from key-value pairs, many other kinds of data structures can be stored and distributed through Hazelcast. Some are simple implementations of Java objects, like Map. Others are specific to Hazelcast. MultiMap, for instance, is a variant on key-value storage that can store multiple values under the same key.

Hazelcast also has measures in place to ensure that operations only proceed if at least a certain number of nodes are online. However, this behavior has to be configured manually, and it only works for certain data structures.

Memcached

Memcached is about as basic and fast as key-value storage gets. Originally written as an acceleration layer for the blogging platform LiveJournal, Memcached has since become a ubiquitous component of web technology stacks. If you have many small fragments of data that can be associated with a simple key and don't need to be replicated between cache instances, Memcached is just about right.

Memcached is most commonly used for caching queries from a database and keeping the results in memory. In fact, Memcached does not back its data store with anything. All keys are held in memory only, so they evaporate whenever the Memcached instance or the server hosting it is reset. Thus Memcached can't really be used as a substitute for a database.

Any data that can be serialized to a binary stream can be stashed in Memcached. Values can be set to expire after a certain length of time, or on-demand, by referencing the keys to the values from an application. The amount of memory you can devote to any given instance of Memcached is entirely up to you, and multiple servers can run Memcached side-by-side as a way to spread out the load. Further, Memcached scales linearly with the number of cores available in a system because it is a multithreaded application.

Memcached's simplicity is both its biggest asset and its biggest drawback. For instance, even though you can run multiple instances of Memcached, whether on the same server or on multiple nodes across a network, there is no automatic federation or synchronization of data between instances. The data inserted into a given Memcached instance is available only from that instance, period.

Most popular programming languages have client libraries for Memcached. For instance, libmemcached allows C/C++ programs to work directly with Memcached instances. It also allows Memcached to be embedded in C programs.

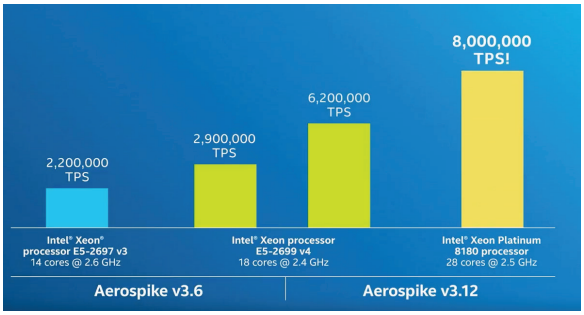
Microsoft Azure Cosmos DB

Most databases have one overarching paradigm: document store, key-value store, wide column store, graph database, and so on. Not so Azure Cosmos DB. Derived from Microsoft's NoSQL database as a service, DocumentDB, Cosmos DB is Microsoft's attempt to create a single database that can use a multiplicity of paradigms.

Cosmos DB uses what's called an atom-record-sequence storage system to support different data models. Atoms are primitive types such as strings, integers, Boolean values, and so on. Records are collections of atoms, like "structs" in C. Sequences are arrays of either atoms or records. Cosmos DB uses these building blocks to replicate the behavior of multiple database types: schemaless JSON documents (DocumentDB and MongoDB), graphs (Gremlin, Apache TinkerPop), and tables.

Table storage is how Cosmos DB provides key-value functionality. When you query a table, you use a set of keys—a "partition key" and a "row key"—to retrieve data. Partition keys can be thought of as bucket or table references, while row keys are used to retrieve the row with the data. The row in question can have multiple data values, but there's nothing that says you can't create a table with only one type of data stored in any particular row. Data can be retrieved via .Net code or REST API call.

Cosmos DB also offers global reach. Data stored in Cosmos DB can be automatically replicated throughout all 36 regions of the Azure cloud. You can also specify one of five consistency levels for reads or queries, depending on the needs of your application. If you want the lowest possible latency for reads at the expense of consistency, choose the eventual consistency model. If you want strong consistency, you can have it, but at the cost of your data being confined to a single Azure region. Three other options strike different balances between these poles.



Redis

If Memcached doesn't offer enough, consider Redis. Redis starts with the same basic idea behind Memcached, an in-memory key-value data store, but takes it further. Redis not only can store and manipulate more complex data structures than just simple binary blobs, but also supports on-disk persistence. Thus Redis can serve as a full-fledged database, instead of just a cache or a quick-and-dirty dumping ground for data.

The creators of Redis call it a "data structures server." The most basic data structure in Redis is a string, and you can use Redis to stash nothing but strings if that's all you need. But Redis can also store data elements inside larger collections—lists, sets, hashes, and more sophisticated structures.

Applications interact with Redis in much the same way as with Memcached. Take a key, associate it with a certain chunk of data, and use the key to obtain the data. Any binary sequence can be used as a key, up to 512MB, although shorter is better. Keys can have time-to-live values or be evicted according to least-recently-used rules.

To do more complex things with the data, you can draw on Redis's specialized data types. These are more akin to the data types found in programming languages than those found in other databases, with each type suited to different use cases.

Consider the Redis list, which is a collection of string elements organized using the same kind of linked-list structure found in Java. Redis lists are great for things like stacks or lists of elements to be read in a fixed order, because adding or removing elements to or from the head or tail of the list takes the same amount of time regardless of the list size. However, if you want random access to items, you're better off using a Redis sorted set.

Redis provides the ability to queue and execute operations atomically in the form of a transaction. Unlike transactions in other databases, though, Redis transactions don't automatically roll back if a command in a transaction fails. Redis's creators rationalize this by claiming that commands only fail due to programming errors, not conditions within Redis itself.

There are a few other areas within Redis where the developer needs to assume additional responsibility. For instance, keys are freeform, meaning they don't have an implicit schema associated with them. If you want to enforce a schema for how keys are constructed, such as an object:type:thing naming convention, you will have to implement it in your application. Redis will not do this for you.

Unlike Memcached, where the data goes poof once the server stops, Redis keeps a periodically updated snapshot of the dataset on disk. The default way to do this is to have Redis write out changes every x seconds if y changes have been made. Another option is "append only," where changes are appended to the existing snapshot, which is then periodically truncated in the background. One advantage to the former approach is that snapshot files, once written, can be backed up or copied out without having to stop Redis first—convenient if you're paranoid about maintaining data integrity.

As a cache layer in front of other applications, Redis offers more flexibility than Memcached, starting with a variety of cache eviction policies to manage the data. Aside from a simple time-to-live policy, Redis also lets you do things like remove keys at random, or give preference to removing keys with shorter time-to-live so that newer data can be added more efficiently. The plethora of choices can be confusing at first, but the recommended default works for the vast majority of use cases, and you can always change eviction policies on the fly, programmatically.

Redis includes an interpreter for the Lua language to run batch operations on Redis. You can think of Lua scripts as Redis's version of stored procedures—a way to accomplish tasks that are slightly too complicated for Redis alone but that don't need a full-blown application. Be warned that Lua scripts, when running, constitute blocking operations on the Redis instance. Nothing else can happen while a Lua script is executing.

Redis 4, which arrived in 2017, introduced a modules system, giving developers a way to add custom data structures and functionality to Redis. Some of the functions that can be added by way of modules include JSON data types, trainable neural networks, and full-text search functionality. It's always worth exploring whether any functionality currently in your application could be offloaded to Redis, where the work can be performed closer to the data.

— Sedar Yegulalp — Senior Writer, Infoworld

Serdar Yegulalp is a senior writer at InfoWorld, covering software development and operations tools, machine learning, containerization, and reviews of products in those categories. Before joining InfoWorld, Serdar wrote for the original Windows Magazine, InformationWeek, the briefly resurrected Byte, and a slew of other publications. When he's not covering IT, he's writing SF and fantasy published under his own personal imprint, Genji Press.