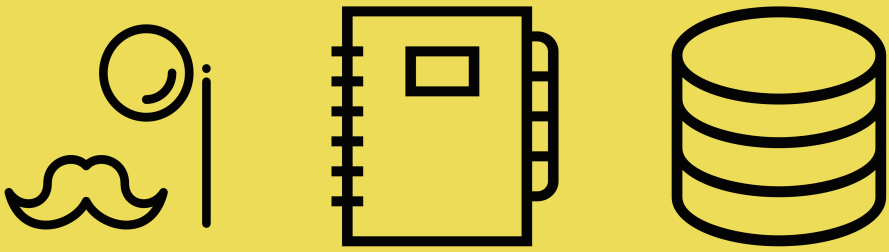

New Data: a Field Guide.

Intro to NoSQL and New SQL

Covers the major families of new storage

- Key-value
- Document
- Graph
- Object
- Columnar
- New SQL (for scale and profit)

Provides overview and use cases for each

J Patrick Davenport

New Data: a Field Guide

A Primer on No and New SQL

J Patrick Davenport

This book is for sale at http://leanpub.com/NewDataAFieldGuide

This version was published on 2015-07-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 J Patrick Davenport

Contents

Welcome to the Technology Jungle

Historically, picking a data store for a new application was a non-event. The paradigm was predetermined: relational database management system (RDBMS). The provider of the RDBMS was predetermined: startups used PostgreSQL or MySQL while many mid-to-large companies went with one of the Big 3 (Oracle, Microsoft or IBM). In fact, even the data modeling process was predetermined by years of experience within the RDBMS paradigm and the associated provider's preference (like stored procs for many MS SQL Server jobs). No one had reason to question this. Few alternatives existed, and those weren't cheap. Why choose something that is both exotic and expensive?

Times have changed. SQL has competitors (or companions) in the NoSQL space. Stressors like high data volumes, rapid change and developer preferences caused a flood of new paradigms. In the wake of these new paradigms, many are left confused as to the proper application of the technologies. New tech brings new terms. Sometimes the industry equivocates on terms (we prefer to call it "overloading"). Added to this, each camp and, even more granular, each vendor within a camp swears that its product is the best thing since set theory. It becomes hard to tease out fact from marketing. This book proposes to do just that.

This book will provide concise descriptions and applications of three major technologies related to data storage: NoSQL, New SQL and Big Data. You will learn the relative terms and theories from a 30,000 ft perspective. The goal is to give readers enough information to better focus their research.

Why Did You Write This?

There are three reasons I undertook this project. First, I've always wanted to explore the world of the hipster. These trendsetters discussed the benefits of adopting some NoSQL tools long before they were cool. They extolled the virtues of a schema-less model. They reveled in the labor of not having traditional ACID transactions. They seldom talked about how their new toys were a bad fit. The sheer fandom made these new ideas interesting.

Second, I was able to see what a non-RDBMS paradigm could do first hand. A few years back, I was honored with the responsibility of extending the Medicare Fraud Detection System for the United States. The challenge was **simple**: create a system that allowed the execution of an arbitrary number of fraud models that read an arbitrary number of years worth of Medicare claims data. A given fraud model might look at just today's data or possibly at 3 years worth of data. The models would also have to calculate an arbitrary number of statics to determine if fraud existed. Finally, all of these calculations had to run within a 24 hr cycle.

All of the traditional, rubber stamped data stores failed to meet these requirements. So we looked elsewhere. We found Hadoop. Hadoop is a product available from Apache labeled under Big Data. In fact it is *the* pedagogic Big Data solution. We didn't know Hadoop, but we dove in. Within 3 months (I know: blindingly quick for a government job) we had a working fraud detection system. One component of the new system replaced one of the old. The old took 8 hours to execute. The new took just 45 minutes. We eventually reduced that time to 30 minutes.

My third reason is simple. A lot of hype and confusion surrounds the topics of NoSQL, New SQL and Big Data, from hundreds of blog and counter blog posts to numerous books on specific topics. However, few general survey books are written on the paradigms. Of those (I found only a couple), none are constructed as a survey

for semi-technical people or technical people who want to get the gist of the technologies. I wanted a book I could give my team or management so we'd all be conversant in general ideas. If and when we decided to use a specific implementation of a tool, it would only be after we had discussed the other options. To have a proper discussion, we all need a common reference.

General Outline of the Book

The book is divided into three major sections.

The first section provides an introduction to the terms. Here, we'll discuss topics like ACID, BASE, Master->Master, etc. Most of these terms apply to more than one of the paradigms. Rather than explaining them within a context of a particular paradigm, and thus forcing a reading order, they are all front-loaded.

The second section looks at NoSQL families. By my estimation, NoSQL has five major families in addition to some hybridized versions. Those families are: Key-Value, Columnar, Document, Graph and Object databases. For each member we'll discuss:

- The architecture
- Common use-case applications
- Products in the family
- Resources for future reading

The third section looks at New SQL and Big Data. These two are grouped together because of the push to make Big Data more SQL friendly by both Apache and the folks at Concurrent and because NewSQL gives up some traditional SQL features in order to get the systems to scale.

There isn't a recommended order of reading. If you already feel comfortable with the general terms, skip it. Most people, however, will want to at least skim the terms section.

So Many Terms, So Little Time

One of the fun parts about researching all of these technologies is learning their underlying theory. Authors go on and on about "My Other Cap is Theorem." They talk about the difference between ACID and BASE. Everything is about being distributed. It's enough to make one's head spin. Even series of Master level courses discuss these topics. Some might want to skip the terms and just see the ROI. However, you'll benefit by examining these terms first. Understanding the ideas is both part of the fun and part of the foundation of larger concepts. It is hard for a manager or developer to fully grok a tool or paradigm without first obtaining at least a basic understanding of that paradigm's terminology. Fortunately, the terms are often shared across paradigms.

Point Zero - Cluster?

This is probably readily known, but should be stated for completeness, if nothing else. A cluster is a logical set of computers. You will probably call each computer a node. Normally you need two. However many of the NoSQL systems work just fine as a cluster with one node. You can add to it as time goes by. Depending on the implementation of the data store, you don't even have to do much to get your data balanced and the whole system running smoothly across the nodes.

I Don't Care Who Made It as Long as It Works

A common refrain you'll hear across the new data providers is "commodity hardware". These are server class computers. You should buy ECC RAM. The difference is that these boxes are a) inexpensive and b) interchangeable. They run standard linux. They don't carry special configurations like you'd see when using Oracle. Presently you can buy them in the 2k-5k range. It is certainly possible to run most of these systems on POS boxes sitting in a cubical (great for testing or boot strapping a startup). Just keep in mind that cheaper components break quicker.

One of the goals on these distributed designs is to allow a node to fail without an OPs representative having to scramble to fix it. For example, at Yahoo! if a normal nodes goes down in one of their Hadoop clusters, it gets fixed during a normal repair cycle. As a result, if your cluster is large enough, quality is less of a factor. You want to aim for mid-quality. Too high brings little value; too low brings major headaches.

Where's My Data

The first thing that you'll probably see in any discussion on most of these data stores is that these stores are distributed. You might have also seen it referred to as being horizontally scalable. During these conversations, you will have probably also seen a contrast to vertically scalable. Because of this, a reasonable place to start is with definitions for these terms.

A vertically scalable (VS) data store is one that's improved by shoving more components in the box. This could mean adding more cores, more RAM, more storage or any combination thereof. This is the traditional model of database design. It's worked well in the

past. Moore's Law[1] pretty much doubled computer performance every 18 months. This model gets to be fairly expensive given enough time. The effect will plateau. Soon you run out of RAM slots, drive bays and CPU sockets. You might end up with a really beefy box. Regardless, you end up with *a* beefy box. One network issue or motherboard issue and you're dead in the water.

Horizontal scaling (HS) looks to bring the cost down and extend the time till plateau. In a horizontally scalable data store, you don't add RAM, drives or CPUs to get better performance, although doing so will still yield improvements. Instead you add commodity servers as you need to expand. By doubling the number of computers in your cluster (group of computers), you should see a near doubling of performance.

Let's compare the two designs. In HS, your data is spread across (striped) the computers (nodes) in the cluster. Depending on how the particular data store does this, it's possible to know exactly where the data is based on the distribution algorithm. This can make lookups really quick. In a VS, all of the data is in one place. Every RDMBS worth its salt has indexes. So like HS, if your query is for an index item, the lookup is really quick. If the data is not indexed, a HS system might have to query multiple systems at once. This makes the query take as long as the slowest node in the cluster. A VS will scan local files.

In a HS system if you lose a node, you lose, at worst, the information on that node[2]. Depending on how your data store works, you might not have even lost that. Most HS have the ability to fail over to a hot copy of the data on the master node. In a VS, if you lose the database server, you're out of luck.

Quick note: HS is not only in NoSQL/New SQL/Big Data. You can get the same ability in Oracle, SQL Server and DB2. The difference

[1] http://en.wikipedia.org/wiki/Moore's_law

[2] Free as in Beer, of course.

between the two is that NoSQL provides the benefits of HS out of the box and for low cost.

Failover Beethoven Tell the Client the News

Let's take a deeper look at failover. Failover is a way for the system to automatically switch to a different data source in the case of a failure. As stated most VS servers lack fail over. If you lose the server, you're dead in the water until you revive it or get a new one.

HS systems provide some form of failover, but you really have to look to see what form this takes. Some allow you to lose a master and still read data. All writes get rejected until the master is back up, but you've got partial availability. In another form, there is a "passive master". This server is copying all the data in the master with each transaction. In the event of a master failure, the passive master steps in to service both reads and writes. Because the passive master was involved in all of the transactions, it is fairly consistent with the state of the master. You will also see some systems as being multi-master or master-master. In this instance there is no one system that is "true". Each master can take writes. The writes are coordinated between the masters to synchronize the data. The benefit of such a system is higher throughput. It does bring issues when trying to understand which record is the "truth".

Another point of failover is how the clients are updated. Many of the HA solutions have a client that knows about each node or at least a majority of the nodes in the cluster. If a node goes down, the client black lists the node. When the node returns to good health, the client corrects its list.

When the Shard Hits the Fan, We Might Replicate That

Two terms that everyone must know is *shard* or *sharding* and *replication*. Often they are treated as if equivalent. They are not. They are orthogonal to each other. Sometimes you need one or both.

Sharding is when you split a large dataset across multiple nodes. Each data entry has a *shard key*. Often this is the Primary Key (PKs) to borrow from the RDBMS terms. The shard key identifies which of the nodes owns the data. As with PKs, *shard keys* provide the fastest lookup mechanism for a particular document. Like PKs from their relational brothers, the key may be generated automatically or derived from the data itself.

A natural example of sharding is signing into some large event like school registration day or a conference. The reception tables are split into groups like last names A-F, G-O, P-Z. We've got three shards. Ideally this grouping handles about the same number of people per group to get the best throughput into the event.

Replication is the process of copying data to more than one place. Its the same data. Each copy is called a *replica*. Normally you want to have replicas in different nodes. While there is added cost in nodes and their respective storage, there are at least two benefits. The first is you've got a backup of the data. At fail over, one of the replicas can set in for the failed node without much interruption. The other benefit is throughput on reads. If the client and data store support it, the client can read from a replica either directly or via a proxy off of a main node. If a particular replica is busy serving a prior request, another server can happily respond to a new request. If your application is read heavy, replication like this might provide a good performance boost.

How the system replicates is a function of the data store. Some stores have all writes go to a single master. The master might

concurrently write to replicas. It might write to itself and then to the replicas. It might concurrently write to itself and the replicas. The exact whys and wherefores impact data consistency. We'll cover consistency later in the chapter. Just keep an eye out the particulars when evaluating your needs and what the specific data stores provide.

Election - I'm the President! No! I AM!

If something happens to the President of the United States, there is a law that defines who gets the job next. Presently (2014) there are 16 possible slots with 2 slots unfilled. As per the law, vacant spots are skipped. This law is rather algorithmic. Each case is defined with a codified response. How much more important is your data?

In the case of a master-slave system (even master-master where a master has slaves), a new master must be elected in the case of, say, a master dying from a blue tail fly. Each distributed system has its own means for doing this. Their documentation will describe the specific algorithm for election. Once a new master is elected, the clients to the system should redirect to the new master for all of the writes, if not all of the reads.

Dropping ACID to Free[3]-BASE

There are two terms here. The first is ACID. The second is BASE.

ACID stands for Atomicity, Consistency, Isolation, and Durability. Atomicity means that all of the database parts involved in a transaction are changed or none are. Consistency means that the database cannot get into an illegal state. Isolation means that the partial effects of a transaction can be visible or hidden depending

[3]This presumes that the data store doesn't have a single management node. If it does, then you're in as much trouble as in a Vertically Scalable system.

on what they user specifies. Finally Durability means that once the database said a change occurred, it sticks.

Most RDBMS' are thought to be ACID compliant [4]. For most people ACID is *thought* to be a requirement. Interestingly, most of the RDBMS do not actually provide ACID and the world still spins.

BASE stands for Basically Available, Soft-State, Eventually Consistent. Basically Available means that some version of the data is available when requested. It's not necessarily the true data, but it a version. Soft-State means that the data is not (necessarily) persisted to a permanent medium like a disk[5]. Eventually consistent means that the whole system will get into a good state given enough time (this could be a few milliseconds, vendor specific).

Let's say you've got a person adding a item to their shopping cart. When they pay, the system has to decrement the number of items in your system's inventory, create a purchase order, and possibly update the user's financial information like adding a new credit card to their list of cards.

In an ACID world, the number of available items is locked when the user added the item to his or her cart. The lock holds until the user checkouts or removes the item from the cart. Assuming that the purchase order goes through, *isolation* means presumes that only the whole order (header and lines) are visible to the larger system. Durable means that once you charge the credit card, the purchase is written to disk so you're legally on the hook to provide the buyer with the items. Consistent means that any rules in the system are not violated (for example causing the available inventory of an item to go below zero). Atomic means that all of the above occurs or none of it does. You cannot partially fulfill an order.

In a BASE-ic world you don't have these guarantees. For example, there is no item lock. So a person adds an item to their cart.

[4] http://www.bailis.org/blog/when-is-acid-acid-rarely/

[5] http://www.cs.berkeley.edu/~brewer/cs262b/TACC.pdf

You might decrement the number of items right then. That's fine but you're not consistent. It could be that the user abandons the cart. This means that you actually have one item available in the warehouse. The user may check out. You could have the purchase order visible to the system with only some of the lines added.

Now some people freak out about BASE. Truth is that it *might* not be that big of a deal. Many enterprise applications don't really leverage ACID because A) web servers make it hard to hold a transaction across page renders and B) developers are told to get and drop a database connection/transaction ASAP. So your applications in the field bringing in your monies might not really be locking and holding values as you might think. Another example is that banking systems aren't ACID even though they are the pedagogic example of ACID. If a banking system truly checked values in both accounts during a transaction, we'd never have overdrafts.

Picture what every database student first learns as the example of ACID. A person withdraws money from one account and deposits it into another. This is said to occur atomically, isolated, consistently and durably. In reality the account can go into an inconsistent state. This is called overdrawn.

When you and your team look at systems that implement BASE over ACID, ask yourself do you really need ACID. It seems comforting at first. It seems natural because we're all taught that it is right way. But then again, it's a tool. Do you need this tool?

And to CAP It All Off a Node Died!

Failure happens everyday. A powerful Oracle box suddenly goes offline due to a bad motherboard. Your web server, that faithful, old, beige box sitting in the closet, ground its last hard drive. Then there's the always humorous accident where a guy accidentally sends a picture of himself dressed as a White Castle Slider to

everyone in your multinational insurance company thereby bringing email down for all the agents and other company personnel including the VPs, VIPs and CEO because the picture was 2.58 MB and Exchange just couldn't handle that load. Yep, failure happens.

Failure happens even more when you're working in a shared/distributed system. Let's say you've got a great system that has a slim chance of failing which means that it's got a 99.9% chance of not failing. If you've got 40 nodes in a cluster you'll have 3.9% chance that something will fail[6]. Now you've got to figure out how you're going to react to failure.

Fortunately the Failure Reaction Triangle exists just like the Project Management Triangle[7]. This triangle is CAP. C stands for Consistency. A is Availability. P is Partition tolerance (T is not capitalized because it would be the CAPT theory and NoSQL folks tend to be pacifists; I'm making this obscure part up). Like the Project Management Triangle, you get to pick two. Unlike the Project Management Triangle, CA is not possible[8].

Consistency means that to an outside observer, like a database client, change events happen at single, logical point. This means that once a change is made to a record, all of the subsequent calls about that record reflect the change.

Availability means that every request to a working node must be satisfied. If a client asks a node for a record on patient A, it has to return the record. If a working node tosses some sort of error from its side, the record is not considered available. Note: if a client sends an invalid request and the server simple returns a bad request error, the system is still actually available. The proper response to getting garbage is to say, "That was crap."

Partition Tolerance deals with how the system works if one or more parts of the system can't talk to each other. Specifically it's

[6] http://codahale.com/you-cant-sacrifice-partition-tolerance/
[7] http://en.wikipedia.org/wiki/Project_management_triangle
[8] http://codahale.com/you-cant-sacrifice-partition-tolerance/

concerned about how the system handles losing messages. If you're looking at a system that says it doesn't have to work with Partition Tolerance, you've got a system that doesn't understand CAP or is one where there is no network. Anything else means the designers have bought one or more of the fallacies of distributed computing[9]. You should really look at another vendor.

Consistent systems react to partitioning different ways. Some might declare a snow day for the whole distributed system. It will reject all reads and writes just as if it were a VS system. It might allow only reads. Finally, it might allow updates based on the master data available in the currently "healthy" pool of nodes.

You'll need to have your team pay close attention to how the system figures out which are healthy and which are not. Let's say you've got 4 nodes in a cluster. Two nodes are in one rack. Two nodes in the other. The network connection between them dies, but both subsets are accessible to some of the clients. How does the logical system determine which nodes to consider healthy?

You might also hear the phrase "eventual consistency". In this model, a system will allow copies of a record to become outdated. A client might not get the latest update because the change may not have percolated out to all the copied nodes. Often times such systems have quorum settings in their drivers. If they do, the client to the datastore will poll multiple nodes (this is the quorum). If X nodes come back with the same answer, the client will take that.

[9] http://www.rgoarchitects.com/Files/fallacies.pdf

Key-Value Stores

Key-Value stores (K-V) are perhaps the simplest of the NoSQL solutions, at least logically. Most developers use maps or dictionaries in their everyday coding. Key-Value stores are an expansion of this idea. Another term for Key-Value is a Distributed Hash Table. As we'll see, a K-V is a lot like an old Rolodex. The key is the person's last name. The value is their contact information. Knowing the last name allows you to quickly jump to the proper location in the Rolodex.

K-Vs are everywhere. You can get them for mobile, for the server side and even in HTML5. What they lack in complexity, K-Vs make up for it with their ubiquity. Depending on your application or system needs using K-Vs through the stack can lead to less mental overhead since developers won't have to jump between different persistence models.

Architecture

At the core is an incredibly simple, yet powerful abstraction: keys and their values. A key is a set of bits that uniquely identifies a thing. That thing is called the value. It too is simply a set of bits. The most rudimentary key-value stores don't attempt to know what's in the key or in the value. Both are opaque to the storage mechanism. A key might be the string "name" with a value of "Patrick". Another key might be the hex number 0xBEEF with the value of "It's what's for dinner". The data store doesn't force any representational semantics on the developer.

The key's content is hashed. This means that it's turned into a new, fixed size data used to pick a slot in a table for the value. Production

hash functions are fast. Looking up a value via the hash is fast. Two fasts equal a fast system.

Given the logical simplicity of the paradigm, the simple K-V provides the operations of `GET`, `PUT`, and `DELETE`. `GET` retrieves a value from the store. `PUT` inserts a value at a key if it doesn't exists or overwrites the value if does. `DELETE` removes the value from the data store.

As stated earlier, most programming languages have the concept of a K-V as a map or dictionary. For example, Java looks like this.

```
1     HashMap<String, Integer> cache = new HashMap<String\
2  , Integer>();
3     cache.put("user.age", 21);
4     System.out.println("User age is " + cache.get("user\
5  .age")); // Prints User age is 21
6     cache.remove("user.age"); // Removes the value 21, \
7  and returns it.
```

From the example above one thing should be obvious, key uniqueness matters. If we used "user.age" for all of our users, we would overwrite the values and get it wrong. Instead we could model it with "<username>.user.age" where <username> is the id for the present user.

While basic CRUD could be enough, some implementations NoSQL variants farther. Many remove the opacity of the values. Rather than treating them as blobs, the K-V knows they are a set of items or a list of items or a counter. As a result the developer/modeler can safely add to a list in a concurrent environment. They can universally and atomically get the next value of an ever increasing integer.

So far we've really just talked about a single node system. In the early days, K-Vs largely were a single node system. Time has marched forward. Many K-Vs support clustering.

Even in a clustered environment, the development team is still using a logical hash table. That table is spread across multiple nodes. The hash function often performs double duty in a distributed system. Not only does it pick the slot for a record, but it also picks the node. All of this occurs quietly in the background. The mental model remains simple.

As we'll see in the Product Overviews section, a K-V can store its information in memory or persistently to a disk. There are tradeoffs and design considerations with either mechanism. Pure memory options are useful for pure speed. They are also purely transient. If the box suffers a power outage, everything in memory will probably be lost. This might not be a bad thing™. When the K-V persists it must slow down some in order write. How much of a slow down is product specific. The benefit is that while slower, you're system can survive a reboot.

K-Vs are transactional at the key level. This means that when you put a value for a key, you either put the whole value or you won't. It is not possible for a partial write to occur. The same is true for a read. You cannot read a partially added value. For example, say you wanted to `put` the value `Bob` in the key "<usersession>.firstname". It is not possible even during failure to write `Bo` or `B`. When a client of a K-V requests "<usersession>.firstname" the system will either return an empty response (if it hasn't got a value or if the value is removed) or it will return `Bob`. It can't return `B`.

Getting to Know the Players

Berkeley DB

The home page is http://www.oracle.com/technetwork/database/database-technologies/berkeleydb

This is the granddaddy of the NoSQL systems. It is so old as to not even qualify as NoSQL according to some members of the

community [10]. To others it is a NoSQL database [11]. I think it is too, so I included it.

It is an embeddable K-V. It runs as part of the memory space of the host applications. The host application accesses it via a C-lang binding. The C-lang binding allows many languages to use it outside of just C.

It is an impressive little system. Each release brings a new feature built on the previous layers. As a result there is a SQLite facade. This allows developers used to working with SQL to continue using SQL over BDB. Behind the scenes the SQL concepts are mapped to simple K-V entries.

Recent improvements allow Berkeley DB to scale beyond a single host. Know as the Berkely DB HA, you get all of the distributed goodness on top of a simple programming API. It is a master-slave setup with automatic master selection.

Depending on how you use it, Berkeley DB is immediately consistent since it is not, by default distributed. When running in HA, you can flip many switches to adjust consistency and read throughput.

Oracle provides many white papers and blog posts on BDB. Here a just a few.

- SQLite API - A Technical Evalutaiton http://www.oracle.com/technetwork/dat wp-186779.pdf
- Using Oracle Berkeley DB Java Edition as a Persistence Manager for the Google Web Toolkit http://www.oracle.com/technetwork/articles/ bdb-gwt-096313.html
- Berkeley DB Java Edition on Android http://www.oracle.com/technetwork/dat je-android-160932.pdf

[10] http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

[11] http://www.oracle.com/technetwork/articles/cloudcomp/berkeleydb-nosql-323570.html

Amazon DynamoDB

The home page is http://aws.amazon.com/dynamodb/

This is the fount of all modern NoSQL K-Vs. When the paper describing it came out in late 2007, it was not available to public. Since then it was promoted to public service, but it is still proprietary.

What's special about Dynamo is its management. Since it is tightly coupled with the AWS infrastructure, a database owner can automate processes like adding a new node to the cluster on demand through a nice web interface. It is also inexpensive to spin up a cluster to kick the tires then kill it off.

The API is more complex than the logical K-Vs we've discussed. Developers can query by id. They can also scan over ids for searches.

There are multiple implementations of the Amazon's paper that provide similar features, but within your data center like Riak and Voldemort.

Redis

The home page is http://redis.io/.

Redis is a K-V in the traditional sense. It is also horizontally scales with failover. What makes Redis stand out is its speed and advanced features.

Values are not just transparent bits. They can be strings, hashes, lists, sets, sorted sets, bitmaps and hyperloglogs. Changes to any of the data structures are atomic. If you add an item to a list, it either commits or it doesn't. Such types allow for a more expressive data model than in traditional K-Vs. Some indie developers use Redis for all of their data needs.

Redis takes types to the extreme by providing a fast pub/sub fire hose message queueing system. Clients can listen on a channel.

Publishers can publish on the channel. It even provides simple routing on channel names.

Other Players

- Riak - http://basho.com/riak/ A replicated K-V with a MapReduce chaser.
- Memcached - http://www.memcached.org/ Old standby for caching services. Large knowledge base and community support.
- Project Voldemort - http://www.project-voldemort.com/voldemort/ Apache implementation of the DynamoDB concepts.

So How Would We Use This?

Generally Anything Transient Looked Up By Key

This is something that can get lost in the simplicity of it all. If you have information, text, pictures, serialized objects, that has a unique key and doesn't require complex queries, a K-V is probably a good bet. A RDBMS is a great tool, but heavy when it comes to simple queries like `SELECT VALUE FROM TABLE WHERE ID= ?`. Using a K-V in the stack will reduce database load which frees it up for more complex service elsewhere. This can include saving database results into a K-V. Your application would check the K-V for an answer. Only if it doesn't exist will it go to the database.

Session Management

Perhaps the most quintessential application of K-Vs is session management. User session information is a natural fit here. There is a session id. That is the key or part of the key. The value is whatever

is needed. For example, a Java developer could use the session id as the key and store a complex Java object in the value.

The lifecycle of a session starts when a user logs in (or perhaps when they simply connect to your site). This creates an initial value or set of values on the application server. The value is pushed to the K-V. Whenever the session information is accessed, the K-V is read. User actions cause various changes to the session, like updating a shopping cart or increasing analytical values. When the session ends on the app server, it is possible to trigger an auto delete or session ETL into another data store like MySQL or MongoDB and then delete the session from the K-V cache.

Multiple open-source projects provide a means of integrating web session storage into Riak or Memcached near the beginning of the request pipeline. As a result, most developers will not have to worry about where their session is stored. This frees them up to focus on the important things: business problems.

Fast Paced Data Landing

Some applications generate high volumes of data. It could be user clicks. It could be requests for ads. K-Vs provide fast lookups and inserts. Distributed solutions provide the ability to grow your memory space to whatever size you require. Once the data is in the cluster a background service can ETL it into a permanent storage if necessary.

Some of the K-Vs support MapReduce to perform (relatively simple) analytics in the K-V itself. As a result it's possible track real-time or near real-time information like leader boards and dashboards.

Oddly, Messaging

Since Redis is a super K-V, we'll look at one of its standout features: fast messaging between components using the Pub/Sub paradigm.

Normally when one thinks messaging they think RabbitMQ, IBM MQ or MSMQ. Most of these are a complex protocol, often times binary. Redis' protocol is fairly straight forward and text based.

Clients register to a channel to publish. Other clients register on that channel to listen. The system is a firehose. If a client disconnects from the channel, it loses all of its messages. It doesn't guarantee delivery either. If you're willing to live within these parameters, you can create chat clients for your customer facing web sites with ease. Internally you can communicate anything with any component.

Sizing and Cost Considerations

When considering sizing and cost one must, now a days, look to Cloud vs Local. We'll first look at sizing locally within a company. Then we'll look at what major companies provide by way of cloud hosting.

On Premises

Many companies find that a single K-V or K-V cluster can provide caching value to multiple applications. Sharing the K-V can reduce cost per application.

K-Vs are memory centric. The more memory you give them, the better they run. They are rarely CPU or local IO centric. As a result, you should put your money into RAM.

After that, sizing varies by need. Essentially, you should get a server with 8 GB of RAM and a 100BASE-T network card. Depending on your level of failure response, you might consider adding a second NIC. Fortunately RAM is fairly cheap. Getting a single server with 16 GB should be cost effective.

If the K-V you're looking at supports sharding or replication, you might want to use it. You'll get better read throughput and possibly

redundancy for fail over. Both are a good thing to have. If you go down the replication path, multiply your base server cost by the number of nodes.

In the Cloud

Presently memory on AWS and Google hosts is rather expensive. If you want to standup your own Redis, Memcached, etc, you'll want to pick a configuration that supports high RAM, but doesn't cost you too much. An AWS `r3.large` presently offers 15 GB of RAM at $0.175 per hour used. Assuming your instance is on 24/7 with 30 average days per month, you'll spend $126/month to host the server. Costs go up as with storage fees. Fortunately most of the IO will be within AWS so you probably won't have to pay transfer fees. Google's `n1-highmem-2` offers 13 GB of RAM at $0.164/hr. So the average monthly cost is $118.08 with a bit less head room. Keep in mind these numbers are per instance.

Now the question is *should you stand up your K-V?* On AWS you can use DynamoDB. Their pricing model shows that a small to mid-size site should probably cost around $7.50 [12]/month. Since DynamoDB is persistent with more advanced modeling features, it could be your only datastore. Google offers a similar service partially for free using Memcache in the AppEngine space. The free version gives you unlimited storage in a Memcache pool. Sadly, nothing is ever free-free. Your items in the Memcache pool may be evicted at anytime depending on Google's needs. Might not sound useful, but every cache hit is a performance gain. You can also pay $0.06 per GB per hour for a dedicated Memcache pool. So a 24/7 30 days/month instance runs about $43.20.

[12] http://aws.amazon.com/dynamodb/pricing/

Further Resources

- *The Architecture of Open Source Applications* has a chapter on the Berkeley DB found at http://aosabook.org/en/bdb.html.
- Data Modeling with Key Value NoSQL Data Stores - Interview with Casey Rosenthal found at http://www.infoq.com/articles/data-modeling-with-key-value-nosql-data-stores
- Dynamo: Amazon's Highly Available Key-value Store found at http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- Comparison of ZeroMQ and Redis by Stephen McDonald http://blog.jupo.org/2013/02/23/a-tale-of-two-queues/