The

NoSQL Ecosystem

Adam Marcus

Unlike most of the other projects in this book, NoSQL is not a tool, but an ecosystem composed of several complimentary and competing tools. The tools branded with the NoSQL monicker provide an alternative to SQL-based relational database systems for storing data. To understand NoSQL, we have to understand the space of available tools, and see how the design of each one explores the space of data storage possibilities.

If you are considering using a NoSQL storage system, you should first understand the wide space of options that NoSQL systems span. NoSQL systems do away with many of the traditional comforts of relational database systems, and operations which were typically encapsulated behind the system boundary of a database are now left to application designers. This requires you to take on the hat of a systems architect, which requires a more in-depth understanding of how such systems are built.

13.1. What's in a Name?

In defining the space of NoSQL, let's first take a stab at defining the name. Taken literally, a NoSQL system presents a query interface to the user that is not SQL. The NoSQL community generally takes a more inclusive view, suggesting that NoSQL systems provide alternatives to traditional relational databases, and allow developers to design projects which use *Not Only* a SQL interface. In some cases, you might replace a relational database with a NoSQL alternative, and in others you will employ a mix-and-match approach to different problems you encounter in application development.

Before diving into the world of NoSQL, let's explore the cases where SQL and the relational model suit your needs, and others where a NoSQL system might be a better fit.

13.1.1. SQL and the Relational Model

SQL is a declarative language for querying data. A declarative language is one in which a programmer specifies *what* they want the system to do, rather than procedurally defining *how* the system should do it. A few examples include: find the record for employee 39, project out only the employee name and phone number from their entire record, filter employee records to those that work in accounting, count the employees in each department, or join the data from the employees table with the managers table.

To a first approximation, SQL allows you to ask these questions without thinking about how the data is laid out on disk, which indices to use to access the data, or what algorithms to use to process the data. A significant architectural component of most relational databases is a *query optimizer*, which decides which of the many logically equivalent query plans to execute to most quickly answer a query. These optimizers are often better than the average database user, but sometimes they do not have enough information or have too simple a model of the system in order to generate the most efficient execution.

Relational databases, which are the most common databases used in practice, follow the *relational data model*. In this

model, different real-world entities are stored in different tables. For example, all employees might be stored in an Employees table, and all departments might be stored in a Departments table. Each row of a table has various properties stored in columns. For example, employees might have an employee id, salary, birth date, and first/last names. Each of these properties will be stored in a column of the Employees table.

The relational model goes hand-in-hand with SQL. Simple SQL queries, such as filters, retrieve all records whose field matches some test (e.g., employeeid = 3, or salary > $20000). More complex constructs cause the database to do some extra work, such as joining data from multiple tables (e.g., what is the name of the department in which employee 3 works?). Other complex constructs such as aggregates (e.g., what is the average salary of my employees?) can lead to full-table scans.

The relational data model defines highly structured entities with strict relationships between them. Querying this model with SQL allows complex data traversals without too much custom development. The complexity of such modeling and querying has its limits, though:

- Complexity leads to unpredictability. SQL's expressiveness makes it challenging to reason about the cost of each query, and thus the cost of a workload. While simpler query languages might complicate application logic, they make it easier to provision data storage systems, which only respond to simple requests.
- There are many ways to model a problem. The relational data model is strict: the schema assigned to each table specifies the data in each row. If we are storing less structured data, or rows with more variance in the columns they store, the relational model may be needlessly restrictive. Similarly, application developers might not find the relational model perfect for modeling every kind of data. For example, a lot of application logic is written in object-oriented languages and includes high-level concepts such as lists, queues, and sets, and some programmers would like their persistence layer to model this.
- If the data grows past the capacity of one server, then the tables in the database will have to be partitioned across computers. To avoid JOINs having to cross the network in order to get data in different tables, we will have to denormalize it. Denormalization stores all of the data from different tables that one might want to look up at once in a single place. This makes our database look like a key-lookup storage system, leaving us wondering what other data models might better suit the data.

It's generally not wise to discard many years of design considerations arbitrarily. When you consider storing your data in a database, consider SQL and the relational model, which are backed by decades of research and development, offer rich modeling capabilities, and provide easy-to-understand guarantees about complex operations. NoSQL is a good option when you have a specific problem, such as large amounts of data, a massive workload, or a difficult data modeling decision for which SQL and relational databases might not have been optimized.

13.1.2. NoSQL Inspirations

The NoSQL movement finds much of its inspiration in papers from the research community. While many papers are at the core of design decisions in NoSQL systems, two stand out in particular.

Google's BigTable [CDG+06] presents an interesting data model, which facilitates sorted storage of multi-column historical data. Data is distributed to multiple servers using a hierarchical range-based partitioning scheme, and data is updated with strict consistency (a concept that we will eventually define in Section 13.5).

Amazon's Dynamo [DHJ+07] uses a different key-oriented distributed datastore. Dynamo's data model is simpler, mapping keys to application-specific blobs of data. The partitioning model is more resilient to failure, but accomplishes that goal through a looser data consistency approach called eventual consistency.

We will dig into each of these concepts in more detail, but it is important to understand that many of them can be mixed and matched. Some NoSQL systems such as HBase[1] sticks closely to the BigTable design. Another NoSQL system named Voldemort[2] replicates many of Dynamo's features. Still other NoSQL projects such as Cassandra[3] have taken some features from BigTable (its data model) and others from Dynamo (its partitioning and consistency schemes).

13.1.3. Characteristics and Considerations

NoSQL systems part ways with the hefty SQL standard and offer simpler but piecemeal solutions for architecting storage solutions. These systems were built with the belief that in simplifying how a database operates over data, an architect can better predict the performance of a query. In many NoSQL systems, complex query logic is left to the application, resulting in a data store with more predictable query performance because of the lack of variability in queries

NoSQL systems part with more than just declarative queries over the relational data. Transactional semantics, consistency, and durability are guarantees that organizations such as banks demand of databases. *Transactions* provide an all-or-nothing guarantee when combining several potentially complex operations into one, such as deducting money from one account and adding the money to another. *Consistency* ensures that when a value is updated, subsequent queries will see the updated value. *Durability* guarantees that once a value is updated, it will be written to stable storage (such as a hard drive) and recoverable if the database crashes.

NoSQL systems relax some of these guarantees, a decision which, for many non-banking applications, can provide acceptable and predictable behavior in exchange for improved performance. These relaxations, combined with data model and query language changes, often make it easier to safely partition a database across multiple machines when the data grows beyond a single machine's capability.

NoSQL systems are still very much in their infancy. The architectural decisions that go into the systems described in this chapter are a testament to the requirements of various users. The biggest challenge in summarizing the architectural features of several open source projects is that each one is a moving target. Keep in mind that the details of individual systems will change. When you pick between NoSQL systems, you can use this chapter to guide your thought process, but not your feature-by-feature product selection.

As you think about NoSQL systems, here is a roadmap of considerations:

- *Data and query model*: Is your data represented as rows, objects, data structures, or documents? Can you ask the database to calculate aggregates over multiple records?
- *Durability*: When you change a value, does it immediately go to stable storage? Does it get stored on multiple machines in case one crashes?
- *Scalability*: Does your data fit on a single server? Do the amount of reads and writes require multiple disks to handle the workload?
- *Partitioning*: For scalability, availability, or durability reasons, does the data need to live on multiple servers? How do you know which record is on which server?
- *Consistency*: If you've partitioned and replicated your records across multiple servers, how do the servers coordinate when a record changes?
- *Transactional semantics*: When you run a series of operations, some databases allow you to wrap them in a transaction, which provides some subset of ACID (Atomicity, Consistency, Isolation, and Durability) guarantees on the transaction and all others currently running. Does your business logic require these guarantees, which often come with performance tradeoffs?
- *Single-server performance*: If you want to safely store data on disk, what on-disk data structures are best-geared toward read-heavy or write-heavy workloads? Is writing to disk your bottleneck?
- *Analytical workloads*: We're going to pay a lot of attention to lookup-heavy workloads of the kind you need to run a responsive user-focused web application. In many cases, you will want to build dataset-sized reports, aggregating statistics across multiple users for example. Does your use-case and toolchain require such functionality?

While we will touch on all of these consideration, the last three, while equally important, see the least attention in this chapter.

13.2. NoSQL Data and Query Models

The *data model* of a database specifies how data is logically organized. Its *query model* dictates how the data can be retrieved and updated. Common data models are the relational model, key-oriented storage model, or various graph models. Query languages you might have heard of include SQL, key lookups, and MapReduce. NoSQL systems combine different data and query models, resulting in different architectural considerations.

13.2.1. Key-based NoSQL Data Models

NoSQL systems often part with the relational model and the full expressivity of SQL by restricting lookups on a dataset to a single field. For example, even if an employee has many properties, you might only be able to retrieve an employee by her ID. As a result, most queries in NoSQL systems are key lookup-based. The programmer selects a key to identify each data item, and can, for the most part, only retrieve items by performing a lookup for their key in the database.

In key lookup-based systems, complex join operations or multiple-key retrieval of the same data might require creative uses of key names. A programmer wishing to look up an employee by his employee ID and to look up all employees in a

department might create two key types. For example, the key `employee:30` would point to an employee record for employee ID 30, and `employee_departments:20` might contain a list of all employees in department 20. A join operation gets pushed into application logic: to retrieve employees in department 20, an application first retrieves a list of employee IDs from key `employee_departments:20`, and then loops over key lookups for each `employee:ID` in the employee list.

The key lookup model is beneficial because it means that the database has a consistent query pattern—the entire workload consists of key lookups whose performance is relatively uniform and predictable. Profiling to find the slow parts of an application is simpler, since all complex operations reside in the application code. On the flip side, the data model logic and business logic are now more closely intertwined, which muddles abstraction.

Let's quickly touch on the data associated with each key. Various NoSQL systems offer different solutions in this space.

Key-Value Stores

The simplest form of NoSQL store is a *key-value* store. Each key is mapped to a value containing arbitrary data. The NoSQL store has no knowledge of the contents of its payload, and simply delivers the data to the application. In our Employee database example, one might map the key `employee:30` to a blob containing JSON or a binary format such as Protocol Buffers[4], Thrift[5], or Avro[6] in order to encapsulate the information about employee 30.

If a developer uses structured formats to store complex data for a key, she must operate against the data in application space: a key-value data store generally offers no mechanisms for querying for keys based on some property of their values. Key-value stores shine in the simplicity of their query model, usually consisting of `set`, `get`, and `delete` primitives, but discard the ability to add simple in-database filtering capabilities due to the opacity of their values. Voldemort, which is based on Amazon's Dynamo, provides a distributed key-value store. BDB[7] offers a persistence library that has a key-value interface.

Key-Data Structure Stores

Key-data structure stores, made popular by Redis[8], assign each value a type. In Redis, the available types a value can take on are integer, string, list, set, and sorted set. In addition to `set` / `get` / `delete`, type-specific commands, such as increment/decrement for integers, or push/pop for lists, add functionality to the query model without drastically affecting performance characteristics of requests. By providing simple type-specific functionality while avoiding multi-key operations such as aggregation or joins, Redis balances functionality and performance.

Key-Document Stores

Key-document stores, such as CouchDB[9], MongoDB[10], and Riak[11], map a key to some document that contains structured information. These systems store documents in a JSON or JSON-like format. They store lists and dictionaries, which can be embedded recursively inside one-another.

MongoDB separates the keyspace into collections, so that keys for Employees and Department, for example, do not collide. CouchDB and Riak leave type-tracking to the developer. The freedom and complexity of document stores is a double-edged sword: application developers have a lot of freedom in modeling their documents, but application-based query logic can become exceedingly complex.

BigTable Column Family Stores

HBase and Cassandra base their data model on the one used by Google's BigTable. In this model, a key identifies a row, which contains data stored in one or more Column Families (CFs). Within a CF, each row can contain multiple columns. The values within each column are timestamped, so that several versions of a row-column mapping can live within a CF.

Conceptually, one can think of Column Families as storing complex keys of the form (row ID, CF, column, timestamp), mapping to values which are sorted by their keys. This design results in data modeling decisions which push a lot of functionality into the keyspace. It is particularly good at modeling historical data with timestamps. The model naturally supports sparse column placement since row IDs that do not have certain columns do not need an explicit NULL value for those columns. On the flip side, columns which have few or no NULL values must still store the column identifier with each row, which leads to greater space consumption.

Each project data model differs from the original BigTable model in various ways, but Cassandra's changes are most notable. Cassandra introduces the notion of a supercolumn within each CF to allow for another level of mapping, modeling, and indexing. It also does away with a notion of locality groups, which can physically store multiple column families together for performance reasons.

13.2.2. Graph Storage

One class of NoSQL stores are graph stores. Not all data is created equal, and the relational and key-oriented data models of storing and querying data are not the best for all data. Graphs are a fundamental data structure in computer science, and systems such as HyperGraphDB[12] and Neo4J[13] are two popular NoSQL storage systems for storing graph-structured data. Graph stores differ from the other stores we have discussed thus far in almost every way: data models, data traversal and querying patterns, physical layout of data on disk, distribution to multiple machines, and the transactional semantics of queries. We can not do these stark differences justice given space limitations, but you should be aware that certain classes of data may be better stored and queried as a graph.

13.2.3. Complex Queries

There are notable exceptions to key-only lookups in NoSQL systems. MongoDB allows you to index your data based on any number of properties and has a relatively high-level language for specifying which data you want to retrieve. BigTable-based systems support scanners to iterate over a column family and select particular items by a filter on a column. CouchDB allows you to create different views of the data, and to run MapReduce tasks across your table to facilitate more complex lookups and updates. Most of the systems have bindings to Hadoop or another MapReduce framework to perform dataset-scale analytical queries.

13.2.4. Transactions

NoSQL systems generally prioritize performance over *transactional semantics*. Other SQL-based systems allow any set of statements—from a simple primary key row retrieval, to a complicated join between several tables which is then subsequently averaged across several fields—to be placed in a transaction.

These SQL databases will offer ACID guarantees between transactions. Running multiple operations in a transaction is Atomic (the A in ACID), meaning all or none of the operations happen. Consistency (the C) ensures that the transaction leaves the database in a consistent, uncorrupted state. Isolation (the I) makes sure that if two transactions touch the same record, they will do without stepping on each other's feet. Durability (the D, covered extensively in the next section), ensures that once a transaction is committed, it's stored in a safe place.

ACID-compliant transactions keep developers sane by making it easy to reason about the state of their data. Imagine multiple transactions, each of which has multiple steps (e.g., first check the value of a bank account, then subtract $60, then update the value). ACID-compliant databases often are limited in how they can interleave these steps while still providing a correct result across all transactions. This push for correctness results in often-unexpected performance characteristics, where a slow transaction might cause an otherwise quick one to wait in line.

Most NoSQL systems pick performance over full ACID guarantees, but do provide guarantees at the key level: two operations on the same key will be serialized, avoiding serious corruption to key-value pairs. For many applications, this decision will not pose noticeable correctness issues, and will allow quick operations to execute with more regularity. It does, however, leave more considerations for application design and correctness in the hands of the developer.

Redis is the notable exception to the no-transaction trend. On a single server, it provides a `MULTI` command to combine multiple operations atomically and consistently, and a `WATCH` command to allow isolation. Other systems provide lower-level *test-and-set* functionality which provides some isolation guarantees.

13.2.5. Schema-free Storage

A cross-cutting property of many NoSQL systems is the lack of schema enforcement in the database. Even in document stores and column family-oriented stores, properties across similar entities are not required to be the same. This has the benefit of supporting less structured data requirements and requiring less performance expense when modifying schemas on-the-fly. The decision leaves more responsibility to the application developer, who now has to program more defensively. For example, is the lack of a `lastname` property on an employee record an error to be rectified, or a schema update which is currently propagating through the system? Data and schema versioning is common in application-level code after a few iterations of a project which relies on *sloppy-schema* NoSQL systems.

13.3. Data Durability

Ideally, all data modifications on a storage system would immediately be safely persisted and replicated to multiple locations to avoid data loss. However, ensuring data safety is in tension with performance, and different NoSQL systems make

different *data durability* guarantees in order to improve performance. Failure scenarios are varied and numerous, and not all NoSQL systems protect you against these issues.

A simple and common failure scenario is a server restart or power loss. Data durability in this case involves having moved the data from memory to a hard disk, which does not require power to store data. Hard disk failure is handled by copying the data to secondary devices, be they other hard drives in the same machine (RAID mirroring) or other machines on the network. However, a data center might not survive an event which causes correlated failure (a tornado, for example), and some organizations go so far as to copy data to backups in data centers several hurricane widths apart. Writing to hard drives and copying data to multiple servers or data centers is expensive, so different NoSQL systems trade off durability guarantees for performance.

13.3.1. Single-server Durability

The simplest form of durability is a *single-server durability*, which ensures that any data modification will survive a server restart or power loss. This usually means writing the changed data to disk, which often bottlenecks your workload. Even if you order your operating system to write data to an on-disk file, the operating system may buffer the write, avoiding an immediate modification on disk so that it can group several writes together into a single operation. Only when the `fsync` system call is issued does the operating system make a best-effort attempt to ensure that buffered updates are persisted to disk.

Typical hard drives can perform 100-200 random accesses (seeks) per second, and are limited to 30-100 MB/sec of sequential writes. Memory can be orders of magnitudes faster in both scenarios. Ensuring efficient single-server durability means limiting the number of random writes your system incurs, and increasing the number of sequential writes per hard drive. Ideally, you want a system to minimize the number of writes between `fsync` calls, maximizing the number of those writes that are sequential, all the while never telling the user their data has been successfully written to disk until that write has been `fsync`ed. Let's cover a few techniques for improving performance of single-server durability guarantees.

Control `fsync` Frequency

Memcached[14] is an example of a system which offers no on-disk durability in exchange for extremely fast in-memory operations. When a server restarts, the data on that server is gone: this makes for a good cache and a poor durable data store.

Redis offers developers several options for when to call `fsync`. Developers can force an `fsync` call after every update, which is the slow and safe choice. For better performance, Redis can `fsync` its writes every N seconds. In a worst-case scenario, the you will lose last N seconds worth of operations, which may be acceptable for certain uses. Finally, for use cases where durability is not important (maintaining coarse-grained statistics, or using Redis as a cache), the developer can turn off `fsync` calls entirely: the operating system will eventually flush the data to disk, but without guarantees of when this will happen.

Increase Sequential Writes by Logging

Several data structures, such as B+Trees, help NoSQL systems quickly retrieve data from disk. Updates to those structures result in updates in random locations in the data structures' files, resulting in several random writes per update if you `fsync` after each update. To reduce random writes, systems such as Cassandra, HBase, Redis, and Riak append update operations to a sequentially-written file called a *log*. While other data structures used by the system are only periodically `fsync`ed, the log is frequently `fsync`ed. By treating the log as the ground-truth state of the database after a crash, these storage engines are able to turn random updates into sequential ones.

While NoSQL systems such as MongoDB perform writes in-place in their data structures, others take logging even further. Cassandra and HBase use a technique borrowed from BigTable of combining their logs and lookup data structures into one *log-structured merge tree*. Riak provides similar functionality with a *log-structured hash table*. CouchDB has modified the traditional B+Tree so that all changes to the data structure are appended to the structure on physical storage. These techniques result in improved write throughput, but require a periodic log compaction to keep the log from growing unbounded.

Increase Throughput by Grouping Writes

Cassandra groups multiple concurrent updates within a short window into a single `fsync` call. This design, called *group commit*, results in higher latency per update, as users have to wait on several concurrent updates to have their own update be acknowledged. The latency bump comes at an increase in throughput, as multiple log appends can happen with a single

`fsync`. As of this writing, every HBase update is persisted to the underlying storage provided by the Hadoop Distributed File System (HDFS)[15], which has recently seen patches to allow support of appends that respect `fsync` and group commit.

13.3.2. Multi-server Durability

Because hard drives and machines often irreparably fail, copying important data across machines is necessary. Many NoSQL systems offer multi-server durability for data.

Redis takes a traditional master-slave approach to replicating data. All operations executed against a master are communicated in a log-like fashion to slave machines, which replicate the operations on their own hardware. If a master fails, a slave can step in and serve the data from the state of the operation log that it received from the master. This configuration might result in some data loss, as the master does not confirm that the slave has persisted an operation in its log before acknowledging the operation to the user. CouchDB facilitates a similar form of directional replication, where servers can be configured to replicate changes to documents on other stores.

MongoDB provides the notion of replica sets, where some number of servers are responsible for storing each document. MongoDB gives developers the option of ensuring that all replicas have received updates, or to proceed without ensuring that replicas have the most recent data. Many of the other distributed NoSQL storage systems support multi-server replication of data. HBase, which is built on top of HDFS, receives multi-server durability through HDFS. All writes are replicated to two or more HDFS nodes before returning control to the user, ensuring multi-server durability.

Riak, Cassandra, and Voldemort support more configurable forms of replication. With subtle differences, all three systems allow the user to specify `N`, the number of machines which should ultimately have a copy of the data, and `W` < `N`, the number of machines that should confirm the data has been written before returning control to the user.

To handle cases where an entire data center goes out of service, multi-server replication across data centers is required. Cassandra, HBase, and Voldemort have *rack-aware* configurations, which specify the rack or data center in which various machines are located. In general, blocking the user's request until a remote server has acknowledged an update incurs too much latency. Updates are streamed without confirmation when performed across wide area networks to backup data centers.

13.4. Scaling for Performance

Having just spoken about handling failure, let's imagine a rosier situation: success! If the system you build reaches success, your data store will be one of the components to feel stress under load. A cheap and dirty solution to such problems is to *scale up* your existing machinery: invest in more RAM and disks to handle the workload on one machine. With more success, pouring money into more expensive hardware will become infeasible. At this point, you will have to replicate data and spread requests across multiple machines to distribute load. This approach is called *scale out*, and is measured by the *horizontal scalability* of your system.

The ideal horizontal scalability goal is *linear scalability*, in which doubling the number of machines in your storage system doubles the query capacity of the system. The key to such scalability is in how the data is spread across machines. Sharding is the act of splitting your read and write workload across multiple machines to scale out your storage system. Sharding is fundamental to the design of many systems, namely Cassandra, HBase, Voldemort, and Riak, and more recently MongoDB and Redis. Some projects such as CouchDB focus on single-server performance and do not provide an in-system solution to sharding, but secondary projects provide coordinators to partition the workload across independent installations on multiple machines.

Let's cover a few interchangeable terms you might encounter. We will use the terms *sharding* and *partitioning* interchangeably. The terms *machine*, *server*, or *node* refer to some physical computer which stores part of the partitioned data. Finally, a *cluster* or *ring* refers to the set of machines which participate in your storage system.

Sharding means that no one machine has to handle the write workload on the entire dataset, but no one machine can answer queries about the entire dataset. Most NoSQL systems are key-oriented in both their data and query models, and few queries touch the entire dataset anyway. Because the primary access method for data in these systems is key-based, sharding is typically key-based as well: some function of the key determines the machine on which a key-value pair is stored. We'll cover two methods of defining the key-machine mapping: hash partitioning and range partitioning.

13.4.1. Do Not Shard Until You Have To

Sharding adds system complexity, and where possible, you should avoid it. Let's cover two ways to scale without sharding: read replicas and caching.

Read Replicas

Many storage systems see more read requests than write requests. A simple solution in these cases is to make copies of the data on multiple machines. All write requests still go to a master node. Read requests go to machines which replicate the data, and are often slightly stale with respect to the data on the write master.

If you are already replicating your data for multi-server durability in a master-slave configuration, as is common in Redis, CouchDB, or MongoDB, the read slaves can shed some load from the write master. Some queries, such as aggregate summaries of your dataset, which might be expensive and often do not require up-to-the-second freshness, can be executed against the slave replicas. Generally, the less stringent your demands for freshness of content, the more you can lean on read slaves to improve read-only query performance.

Caching

Caching the most popular content in your system often works surprisingly well. Memcached dedicates blocks of memory on multiple servers to cache data from your data store. Memcached clients take advantage of several horizontal scalability tricks to distribute load across Memcached installations on different servers. To add memory to the cache pool, just add another Memcached host.

Because Memcached is designed for caching, it does not have as much architectural complexity as the persistent solutions for scaling workloads. Before considering more complicated solutions, think about whether caching can solve your scalability woes. Caching is not solely a temporary band-aid: Facebook has Memcached installations in the range of tens of terabytes of memory!

Read replicas and caching allow you to scale up your read-heavy workloads. When you start to increase the frequency of writes and updates to your data, however, you will also increase the load on the master server that contains all of your up-to-date data. For the rest of this section, we will cover techniques for sharding your write workload across multiple servers.

13.4.2. Sharding Through Coordinators

The CouchDB project focuses on the single-server experience. Two projects, Lounge and BigCouch, facilitate sharding CouchDB workloads through an external proxy, which acts as a front end to standalone CouchDB instances. In this design, the standalone installations are not aware of each other. The coordinator distributes requests to individual CouchDB instances based on the key of the document being requested.

Twitter has built the notions of sharding and replication into a coordinating framework called Gizzard[16]. Gizzard takes standalone data stores of any type—you can build wrappers for SQL or NoSQL storage systems—and arranges them in trees of any depth to partition keys by key range. For fault tolerance, Gizzard can be configured to replicate data to multiple physical machines for the same key range.

13.4.3. Consistent Hash Rings

Good hash functions distribute a set of keys in a uniform manner. This makes them a powerful tool for distributing key-value pairs among multiple servers. The academic literature on a technique called *consistent hashing* is extensive, and the first applications of the technique to data stores was in systems called *distributed hash tables* (*DHTs*). NoSQL systems built around the principles of Amazon's Dynamo adopted this distribution technique, and it appears in Cassandra, Voldemort, and Riak.

Hash Rings by Example

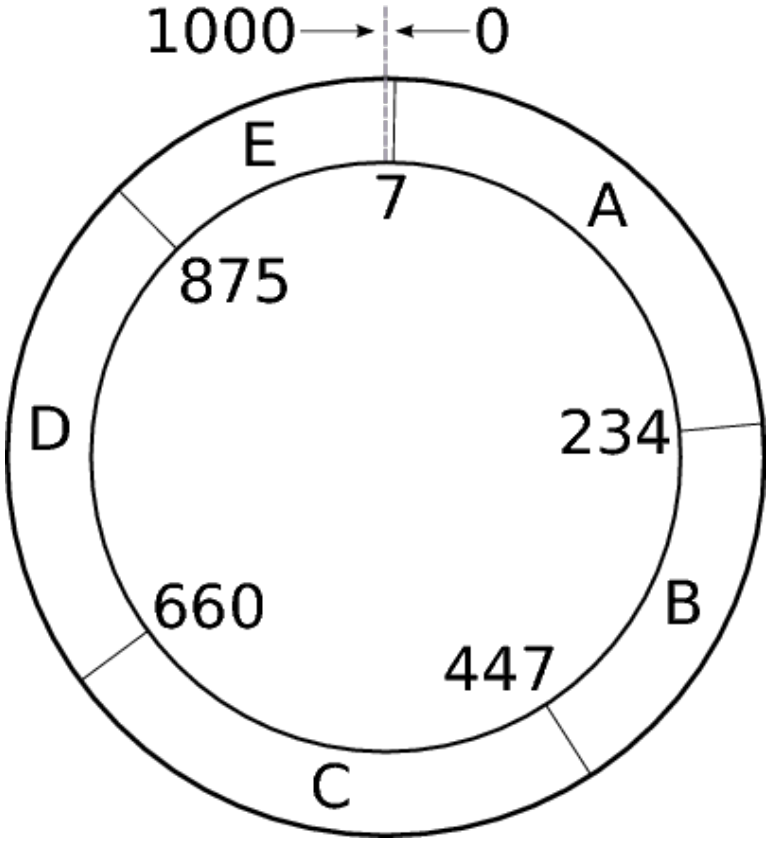

Figure 13.1: A Distributed Hash Table Ring

Consistent hash rings work as follows. Say we have a hash function `H` that maps keys to uniformly distributed large integer values. We can form a ring of numbers in the range [1, L] that wraps around itself with these values by taking H(key) mod L for some relatively large integer L. This will map each key into the range [1,L]. A consistent hash ring of servers is formed by taking each server's unique identifier (say its IP address), and applying H to it. You can get an intuition for how this works by looking at the hash ring formed by five servers (`A` - `E`) in Figure 13.1.

There, we picked `L = 1000` . Let's say that `H(A) mod L = 7` , `H(B) mod L = 234` , `H(C) mod L = 447` , `H(D) mod L = 660` , and `H(E) mod L = 875` . We can now tell which server a key should live on. To do this, we map all keys to a server by seeing if it falls in the range between that server and the next one in the ring. For example, `A` is responsible for keys whose hash value falls in the range [7,233], and `E` is responsible for keys in the range [875, 6] (this range wraps around on itself at 1000). So if `H('employee30') mod L = 899` , it will be stored by server `E` , and if `H('employee31') mod L = 234` , it will be stored on server `B` .

Replicating Data

Replication for multi-server durability is achieved by passing the keys and values in one server's assigned range to the servers following it in the ring. For example, with a replication factor of 3, keys mapped to the range [7,233] will be stored on servers `A` , `B` , and `C` . If `A` were to fail, its neighbors `B` and `C` would take over its workload. In some designs, `E` would replicate and take over `A` 's workload temporarily, since its range would expand to include `A` 's.

Achieving Better Distribution

While hashing is statistically effective at uniformly distributing a keyspace, it usually requires many servers before it distributes evenly. Unfortunately, we often start with a small number of servers that are not perfectly spaced apart from one-another by the hash function. In our example, `A` 's key range is of length 227, whereas `E` 's range is 132. This leads to uneven load on different servers. It also makes it difficult for servers to take over for one-another when they fail, since a neighbor suddenly has to take control of the entire range of the failed server.

To solve the problem of uneven large key ranges, many DHTs including Riak create several `virtual' nodes per physical machine. For example, with 4 virtual nodes, server `A` will act as server `A_1` , `A_2` , `A_3` , and `A_4` . Each virtual node hashes to a different value, giving it more opportunity to manage keys distributed to different parts of the keyspace.

Voldemort takes a similar approach, in which the number of partitions is manually configured and usually larger than the number of servers, resulting in each server receiving a number of smaller partitions.

Cassandra does not assign multiple small partitions to each server, resulting in sometimes uneven key range distributions. For load-balancing, Cassandra has an asynchronous process which adjusts the location of servers on the ring depending on their historic load.

13.4.4. Range Partitioning

In the range partitioning approach to sharding, some machines in your system keep metadata about which servers contain which key ranges. This metadata is consulted to route key and range lookups to the appropriate servers. Like the consistent hash ring approach, this range partitioning splits the keyspace into ranges, with each key range being managed by one machine and potentially replicated to others. Unlike the consistent hashing approach, two keys that are next to each other in the key's sort order are likely to appear in the same partition. This reduces the size of the routing metadata, as large ranges are compressed to [start, end] markers.

In adding active record-keeping of the *range-to-server* mapping, the range partitioning approach allows for more fine-grained control of load-shedding from heavily loaded servers. If a specific key range sees higher traffic than other ranges, a load manager can reduce the size of the range on that server, or reduce the number of shards that this server serves. The added freedom to actively manage load comes at the expense of extra architectural components which monitor and route shards.

The BigTable Way

Google's BigTable paper describes a range-partitioning hierarchical technique for sharding data into tablets. A tablet stores a range of row keys and values within a column family. It maintains all of the necessary logs and data structures to answer queries about the keys in its assigned range. Tablet servers serve multiple tablets depending on the load each tablet is experiencing.

Each tablet is kept at a size of 100-200 MB. As tablets change in size, two small tablets with adjoining key ranges might be combined, or a large tablet might be split in two. A master server analyzes tablet size, load, and tablet server availability. The master adjusts which tablet server serves which tablets at any time.

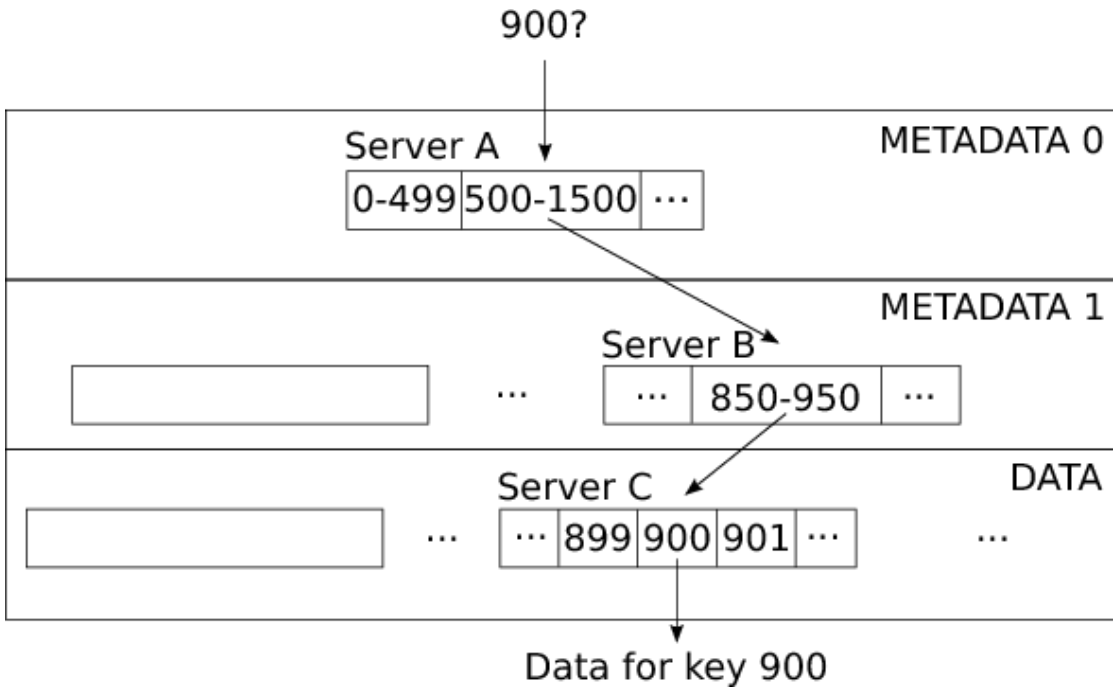

Figure 13.2: BigTable-based Range Partitioning

The master server maintains the tablet assignment in a metadata table. Because this metadata can get large, the metadata table is also sharded into tablets that map key ranges to tablets and tablet servers responsible for those ranges. This results in a three-layer hierarchy traversal for clients to find a key on its hosting tablet server, as depicted in Figure 13.2.

Let's look at an example. A client searching for key `900` will query server `A`, which stores the tablet for metadata level 0.

This tablet identifies the metadata level 1 tablet on server 6 containing key ranges 500-1500. The client sends a request to server `B` with this key, which responds that the tablet containing keys 850-950 is found on a tablet on server C. Finally, the client sends the key request to server `C`, and gets the row data back for its query. Metadata tablets at level 0 and 1 may be cached by the client, which avoids putting undue load on their tablet servers from repeat queries. The BigTable paper explains that this 3-level hierarchy can accommodate 2^{61} bytes worth of storage using 128MB tablets.

Handling Failures

The master is a single point of failure in the BigTable design, but can go down temporarily without affecting requests to tablet servers. If a tablet server fails while serving tablet requests, it is up to the master to recognize this and re-assign its tablets while requests temporarily fail.

In order to recognize and handle machine failures, the BigTable paper describes the use of Chubby, a distributed locking system for managing server membership and liveness. ZooKeeper[17] is the open source implementation of Chubby, and several Hadoop-based projects utilize it to manage secondary master servers and tablet server reassignment.

Range Partitioning-based NoSQL Projects

HBase employs BigTable's hierarchical approach to range-partitioning. Underlying tablet data is stored in Hadoop's distributed filesystem (HDFS). HDFS handles data replication and consistency among replicas, leaving tablet servers to handle requests, update storage structures, and initiate tablet splits and compactions.

MongoDB handles range partitioning in a manner similar to that of BigTable. Several configuration nodes store and manage the routing tables that specify which storage node is responsible for which key ranges. These configuration nodes stay in sync through a protocol called *two-phase commit*, and serve as a hybrid of BigTable's master for specifying ranges and Chubby for highly available configuration management. Separate routing processes, which are stateless, keep track of the most recent routing configuration and route key requests to the appropriate storage nodes. Storage nodes are arranged in replica sets to handle replication.

Cassandra provides an order-preserving partitioner if you wish to allow fast range scans over your data. Cassandra nodes are still arranged in a ring using consistent hashing, but rather than hashing a key-value pair onto the ring to determine the server to which it should be assigned, the key is simply mapped onto the server which controls the range in which the key naturally fits. For example, keys 20 and 21 would both be mapped to server A in our consistent hash ring in Figure 13.1, rather than being hashed and randomly distributed in the ring.

Twitter's Gizzard framework for managing partitioned and replicated data across many back ends uses range partitioning to shard data. Routing servers form hierarchies of any depth, assigning ranges of keys to servers below them in the hierarchy. These servers either store data for keys in their assigned range, or route to yet another layer of routing servers. Replication in this model is achieved by sending updates to multiple machines for a key range. Gizzard routing nodes manage failed writes in different manner than other NoSQL systems. Gizzard requires that system designers make all updates idempotent (they can be run twice). When a storage node fails, routing nodes cache and repeatedly send updates to the node until the update is confirmed.

13.4.5. Which Partitioning Scheme to Use

Given the hash- and range-based approaches to sharding, which is preferable? It depends. Range partitioning is the obvious choice to use when you will frequently be performing range scans over the keys of your data. As you read values in order by key, you will not jump to random nodes in the network, which would incur heavy network overhead. But if you do not require range scans, which sharding scheme should you use?

Hash partitioning gives reasonable distribution of data across nodes, and random skew can be reduced with virtual nodes. Routing is simple in the hash partitioning scheme: for the most part, the hash function can be executed by clients to find the appropriate server. With more complicated rebalancing schemes, finding the right node for a key becomes more difficult.

Range partitioning requires the upfront cost of maintaining routing and configuration nodes, which can see heavy load and become central points of failure in the absence of relatively complex fault tolerance schemes. Done well, however, range-partitioned data can be load-balanced in small chunks which can be reassigned in high-load situations. If a server goes down, its assigned ranges can be distributed to many servers, rather than loading the server's immediate neighbors during downtime.

13.5. Consistency

Having spoken about the virtues of replicating data to multiple machines for durability and spreading load, it's time to let you in on a secret: keeping replicas of your data on multiple machines consistent with one-another is hard. In practice, replicas will crash and get out of sync, replicas will crash and never come back, networks will partition two sets of replicas, and messages between machines will get delayed or lost. There are two major approaches to data consistency in the NoSQL ecosystem. The first is strong consistency, where all replicas remain in sync. The second is eventual consistency, where replicas are allowed to get out of sync, but eventually catch up with one-another. Let's first get into why the second option is an appropriate consideration by understanding a fundamental property of distributed computing. After that, we'll jump into the details of each approach.

13.5.1. A Little Bit About CAP

Why are we considering anything short of strong consistency guarantees over our data? It all comes down to a property of distributed systems architected for modern networking equipment. The idea was first proposed by Eric Brewer as the *CAP Theorem*, and later proved by Gilbert and Lynch [GL02]. The theorem first presents three properties of distributed systems which make up the acronym CAP:

- *Consistency*: do all replicas of a piece of data always logically agree on the same version of that data by the time you read it? (This concept of consistency is different than the C in ACID.)
- *Availability*: Do replicas respond to read and write requests regardless of how many replicas are inaccessible?
- *Partition tolerance*: Can the system continue to operate even if some replicas temporarily lose the ability to communicate with each other over the network?

The theorem then goes on to say that a storage system which operates on multiple computers can only achieve two of these properties at the expense of a third. Also, we are forced to implement partition-tolerant systems. On current networking hardware using current messaging protocols, packets can be lost, switches can fail, and there is no way to know whether the network is down or the server you are trying to send a message to is unavailable. All NoSQL systems should be partition-tolerant. The remaining choice is between consistency and availability. No NoSQL system can provide both at the same time.

Opting for consistency means that your replicated data will not be out of sync across replicas. An easy way to achieve consistency is to require that all replicas acknowledge updates. If a replica goes down and you can not confirm data updates on it, then you degrade availability on its keys. This means that until all replicas recover and respond, the user can not receive successful acknowledgment of their update operation. Thus, opting for consistency is opting for a lack of round-the-clock availability for each data item.

Opting for availability means that when a user issues an operation, replicas should act on the data they have, regardless of the state of other replicas. This may lead to diverging consistency of data across replicas, since they weren't required to acknowledge all updates, and some replicas may have not noted all updates.

The implications of the CAP theorem lead to the strong consistency and eventual consistency approaches to building NoSQL data stores. Other approaches exist, such as the relaxed consistency and relaxed availability approach presented in Yahoo!'s PNUTS [CRS+08] system. None of the open source NoSQL systems we discuss has adopted this technique yet, so we will not discuss it further.

13.5.2. Strong Consistency

Systems which promote strong consistency ensure that the replicas of a data item will always be able to come to consensus on the value of a key. Some replicas may be out of sync with one-another, but when the user asks for the value of `employee30:salary`, the machines have a way to consistently agree on the value the user sees. How this works is best explained with numbers.

Say we replicate a key on N machines. Some machine, perhaps one of the `N`, serves as a coordinator for each user request. The coordinator ensures that a certain number of the `N` machines has received and acknowledged each request. When a write or update occurs to a key, the coordinator does not confirm with the user that the write occurred until `W` replicas confirm that they have received the update. When a user wants to read the value for some key, the coordinator responds when at least R have responded with the same value. We say that the system exemplifies strong consistency if `R+W>N`.

Putting some numbers to this idea, let's say that we're replicating each key across N=3 machines (call them `A`, `B`, and `C`). Say that the key `employee30:salary` is initially set to the value $20,000, but we want to give `employee30` a raise to $30,000. Let's require that at least `W=2` of `A`, `B`, or `C` acknowledge each write request for a key. When `A` and `B` confirm the write request for `(employee30:salary, $30,000)`, the coordinator lets the user know that `employee30:salary` is safely updated. Let's assume that machine C never received the write request for `employee30:salary`, so it still has the value $20,000. When a coordinator gets a read request for key `employee30:salary`, it will send that request to all 3 machines:

- If we set `R=1`, and machine `C` responds first with $20,000, our employee will not be very happy.
- However, if we set `R=2`, the coordinator will see the value from `C`, wait for a second response from `A` or `B`, which will conflict with `C`'s outdated value, and finally receive a response from the third machine, which will confirm that $30,000 is the majority opinion.

So in order to achieve strong consistency in this case, we need to set `R`=2} so that `R+W`3}.

What happens when `W` replicas do not respond to a write request, or `R` replicas do not respond to a read request with a consistent response? The coordinator can timeout eventually and send the user an error, or wait until the situation corrects itself. Either way, the system is considered unavailable for that request for at least some time.

Your choice of `R` and `W` affect how many machines can act strangely before your system becomes unavailable for different actions on a key. If you force all of your replicas to acknowledge writes, for example, then `W=N`, and write operations will hang or fail on any replica failure. A common choice is `R + W = N + 1`, the minimum required for strong consistency while still allowing for temporary disagreement between replicas. Many strong consistency systems opt for `W=N` and `R=1`, since they then do not have to design for nodes going out of sync.

HBase bases its replicated storage on HDFS, a distributed storage layer. HDFS provides strong consistency guarantees. In HDFS, a write cannot succeed until it has been replicated to all `N` (usually 2 or 3) replicas, so `W = N`. A read will be satisfied by a single replica, so `R = 1`. To avoid bogging down write-intensive workloads, data is transferred from the user to the replicas asynchronously in parallel. Once all replicas acknowledge that they have received copies of the data, the final step of swapping the new data in to the system is performed atomically and consistently across all replicas.

13.5.3. Eventual Consistency

Dynamo-based systems, which include Voldemort, Cassandra, and Riak, allow the user to specify `N`, `R`, and `W` to their needs, even if `R + W <= N`. This means that the user can achieve either strong or eventual consistency. When a user picks eventual consistency, and even when the programmer opts for strong consistency but `W` is less than `N`, there are periods in which replicas might not see eye-to-eye. To provide eventual consistency among replicas, these systems employ various tools to catch stale replicas up to speed. Let's first cover how various systems determine that data has gotten out of sync, then discuss how they synchronize replicas, and finally bring in a few dynamo-inspired methods for speeding up the synchronization process.

Versioning and Conflicts

Because two replicas might see two different versions of a value for some key, data versioning and conflict detection is important. The dynamo-based systems use a type of versioning called *vector clocks*. A vector clock is a vector assigned to each key which contains a counter for each replica. For example, if servers `A`, `B`, and `C` are the three replicas of some key, the vector clock will have three entries, `(N_A, N_B, N_C)`, initialized to `(0,0,0)`.

Each time a replica modifies a key, it increments its counter in the vector. If B modifies a key that previously had version `(39, 1, 5)`, it will change the vector clock to `(39, 2, 5)`. When another replica, say `C`, receives an update from B about the key's data, it will compare the vector clock from `B` to its own. As long as its own vector clock counters are all less than the ones delivered from `B`, then it has a stale version and can overwrite its own copy with `B`'s. If `B` and `C` have clocks in which some counters are greater than others in both clocks, say `(39, 2, 5)` and `(39, 1, 6)`, then the servers recognize that they received different, potentially unreconcilable updates over time, and identify a conflict.

Conflict Resolution

Conflict resolution varies across the different systems. The Dynamo paper leaves conflict resolution to the application using the storage system. Two versions of a shopping cart can be merged into one without significant loss of data, but two versions of a collaboratively edited document might require human reviewer to resolve conflict. Voldemort follows this model,

returning multiple copies of a key to the requesting client application upon conflict.

Cassandra, which stores a timestamp on each key, uses the most recently timestamped version of a key when two versions are in conflict. This removes the need for a round-trip to the client and simplifies the API. This design makes it difficult to handle situations where conflicted data can be intelligently merged, as in our shopping cart example, or when implementing distributed counters. Riak allows both of the approaches offered by Voldemort and Cassandra. CouchDB provides a hybrid: it identifies a conflict and allows users to query for conflicted keys for manual repair, but deterministically picks a version to return to users until conflicts are repaired.

Read Repair
If R replicas return non-conflicting data to a coordinator, the coordinator can safely return the non-conflicting data to the application. The coordinator may still notice that some of the replicas are out of sync. The Dynamo paper suggests, and Cassandra, Riak, and Voldemort implement, a technique called *read repair* for handling such situations. When a coordinator identifies a conflict on read, even if a consistent value has been returned to the user, the coordinator starts conflict-resolution protocols between conflicted replicas. This proactively fixes conflicts with little additional work. Replicas have already sent their version of the data to the coordinator, and faster conflict resolution will result in less divergence in the system.

Hinted Handoff
Cassandra, Riak, and Voldemort all employ a technique called *hinted handoff* to improve write performance for situations where a node temporarily becomes unavailable. If one of the replicas for a key does not respond to a write request, another node is selected to temporarily take over its write workload. Writes for the unavailable node are kept separately, and when the backup node notices the previously unavailable node become available, it forwards all of the writes to the newly available replica. The Dynamo paper utilizes a 'sloppy quorum' approach and allows the writes accomplished through hinted handoff to count toward the W required write acknowledgments. Cassandra and Voldemort will not count a hinted handoff against W, and will fail a write which does not have W confirmations from the originally assigned replicas. Hinted handoff is still useful in these systems, as it speeds up recovery when an unavailable node returns.

Anti-Entropy
When a replica is down for an extended period of time, or the machine storing hinted handoffs for an unavailable replica goes down as well, replicas must synchronize from one-another. In this case, Cassandra and Riak implement a Dynamo-inspired process called *anti-entropy*. In anti-entropy, replicas exchange *Merkle Trees* to identify parts of their replicated key ranges which are out of sync. A Merkle tree is a hierarchical hash verification: if the hash over the entire keyspace is not the same between two replicas, they will exchange hashes of smaller and smaller portions of the replicated keyspace until the out-of-sync keys are identified. This approach reduces unnecessary data transfer between replicas which contain mostly similar data.

Gossip
Finally, as distributed systems grow, it is hard to keep track of how each node in the system is doing. The three Dynamo-based systems employ an age-old high school technique known as *gossip* to keep track of other nodes. Periodically (every second or so), a node will pick a random node it once communicated with to exchange knowledge of the health of the other nodes in the system. In providing this exchange, nodes learn which other nodes are down, and know where to route clients in search of a key.

13.6. A Final Word

The NoSQL ecosystem is still in its infancy, and many of the systems we've discussed will change architectures, designs, and interfaces. The important takeaways in this chapter are not what each NoSQL system currently does, but rather the design decisions that led to a combination of features that make up these systems. NoSQL leaves a lot of design work in the hands of the application designer. Understanding the architectural components of these systems will not only help you build the next great NoSQL amalgamation, but also allow you to use current versions responsibly.

13.7. Acknowledgments

I am grateful to Jackie Carter, Mihir Kedia, and the anonymous reviewers for their comments and suggestions to improve the chapter. This chapter would also not be possible without the years of dedicated work of the NoSQL community. Keep building!

ootnotes

. http://hbase.apache.org/
. http://project-voldemort.com/
. http://cassandra.apache.org/
. http://code.google.com/p/protobuf/
. http://thrift.apache.org/
. http://avro.apache.org/
. http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html
. http://redis.io/
. http://couchdb.apache.org/
. http://www.mongodb.org/
. http://www.basho.com/products_riak_overview.php
. http://www.hypergraphdb.org/index
. http://neo4j.org/
. http://memcached.org/
. http://hadoop.apache.org/hdfs/
. http://github.com/twitter/gizzard
. http://hadoop.apache.org/zookeeper/