High Performance Scalable Data Stores

Rick Cattell[1] April 27, 2010

1. Overview

In recent years, a number of new systems, sometimes called "NoSQL" systems, have been introduced to provide indexed data storage that is much higher performance than existing relational database products like MySQL, Oracle, DB2, and SQL Server. These data storage systems have a number of features in common:

- a simple call level interface or protocol (in contrast to a SQL binding)
- ability to horizontally scale throughput over many servers,
- efficient use of distributed indexes and RAM for data storage, and
- the ability to dynamically define new attributes or data schema.

The systems differ in other ways, and in this paper I contrast those differences. They range in functionality from the simplest distributed hashing (analogous to memcached) to highly scalable partitioned tables, as supported by Google's BigTable. In fact, BigTable, memcached, and Amazon's Dynamo provided a "proof of concept" that inspired many of the data stores we describe here:

- Memcached demonstrated that in-memory indexes can be scalable, distributing and replicating objects over multiple nodes.
- Dynamo pioneered the concept of "eventual consistency" as a way to achieve higher availability and scalability, and
- BigTable demonstrated that persistent record storage could be scaled to thousands of nodes, a feat that all of the other systems aspire to.

Good performance on a single multicore node is important, but I believe the most important feature of most NoSQL systems is "shared nothing" horizontal scaling, sharding and replicating over many servers. This allows them to support a large number of simple read/write operations per second, as in Web 2.0 applications, for example. This simple operation load is traditionally called OLTP (online transaction processing). To date, RDBMSs have not provided good horizontal scaling for OLTP, but at the end of this paper we will look at RDBMSs making progress in that direction. Note that data

---

[1] Caveats: This document is still in progress, I hope to publish it as a paper at some point. Input is solicited: rick@cattell.net. Be forewarned that many of the statements in this document are based on sources and documentation that might not be reliable. I haven't yet tested the facts first-hand on data store implementations.

Acknowledgements: I'd like to thank Jonathan Ellis, Dan DeMaggio, Kyle Banker, John Busch, Darpan Dinker, David Van Couvering, Peter Zaitsev, Steve Yen, and Scott Jarr for their input on earlier drafts. Any errors are my own, however! I'd also like to thank Schooner Technologies for their support on this paper.

warehousing RDBMSs provide horizontal scaling of complex joins and queries when the database is read-only or read-mostly, but that is not the focus of these new systems.

In this document, I will simply refer to the new systems as "data stores", because "NoSQL" does not seem an accurate description, and the term "database system" is widely used to refer to traditional DBMSs. However, I will still use the term "database" to refer to the stored data in these systems. All of the data stores have some administrative unit that you would call a database: data may be stored in one file, or in a directory, or via some other mechanism that defines the scope of data used by a group of applications. Each database is an island unto itself, even if the database is partitioned and distributed over multiple machines: there is no "federated database" concept in these systems (as with some relational and object-oriented databases), allowing multiple separately-administered databases to appear as one.

In addition to the "NoSQL" datastores, I will examine some new scalable relational database systems. These systems may overcome the scalability shortcomings of traditional RDBMSs while retaining the simplicity of SQL and transactions. If they succeed, the need for NoSQL systems would be much reduced.

Data Models

Unlike traditional DBMSs, the terminology used by the various data stores is often inconsistent. For the purposes of this paper, we need a consistent way to compare the data models and functionality.

All of the systems provide a way to store scalar values, like numbers and strings, as well as BLOBs. Almost all of them provide a way to store compound or "complex" values as well, but they differ in their structure:

- A "tuple" is a set of attribute-value pairs, analogous to a row in a relational table, where attribute names are pre-defined in a schema, and the values must be scalar. The values are referenced by attribute name, as opposed to an array or list, where they are referenced by ordinal position.
- A "document" is a set of attribute-value pairs, where the values may be complex, and the attribute names are dynamically defined for each document at runtime. A document differs from a tuple in that the attributes are not defined in a global schema, and complex or nested values are permitted.
- An "extensible record" is a hybrid between a tuple and a document, where families of attributes are defined in a schema, but new attributes can be defined (within an attribute family) on a per-record basis.
- An "object" is a set of attribute-value pairs, where the values may be complex values or pointers to other objects. This is analogous to objects in programming languages, but without the procedural methods.

Data Stores

In this paper, I'm grouping the data stores according to their data model:

1. *Key-value Stores*: These systems store values and an index to find them, based on a programmer-defined key.
2. *Document Stores*: These systems store documents (as just defined). The documents are indexed and a simple query mechanism may be provided.
3. *Extensible Record Stores*: These systems store extensible records that can be partitioned across nodes. Some papers call these "column-oriented databases".
4. *Relational Databases*: These systems store (and index and query) tuples. Some new RDBMSs provide horizontal scaling, so they are included in this paper.

Data stores in these four categories are covered in the next four sections, respectively. Web sites for more information on these systems are listed in the tables at the end of the paper.

2. Key-Value Stores

The simplest data stores use a data model similar to the popular memcached distributed in-memory cache, with a single key-value index for all the data. We'll call these systems *key-value stores*. Unlike memcached, these systems generally provide some persistence mechanism and additional functionality as well: replication, versioning, locking, transactions, sorting, and/or other features. The client interface provides inserts, deletes, and index lookups. Like memcached, none of these systems offer secondary indices or keys.

Voldemort

Voldemort is an advanced key-value store, written in Java. It is open source, with substantial contributions from LinkedIn. The code is mature enough that LinkedIn is using it internally. Voldemort provides multi-version concurrency control for updates. It updates replicas asynchronously, so it does not guarantee a consistent data. However, it supports optimistic locking for consistent multi-record updates, and can give an up-to-date view if you read a majority of copies. For the put and delete operations you can specify which version you are updating. Vector clocks (from Dynamo) provide an ordering on versions.

Voldemort supports automatic partitioning with consistent hashing. Data partitioning is transparent. Nodes can be added or removed from a database cluster, and Voldemort adapts automatically. Voldemort automatically detects and recovers failed nodes.

Voldemort can store data in RAM, but it also permits plugging in a different storage engine. In particular, it supports a BerkeleyDB and Random Access File storage engine. Voldemort supports lists and records in addition to simple scalar values.

Riak

Riak is written in Erlang. It was open-sourced by Basho in mid-2009. Basho alternately describes Riak as a "key-value store" and "document store". I categorize it as an advanced key-value store, because it lacks key features of document stores, but it (and Voldemort, I believe) have more functionality than the other key-value stores:

- Riak objects can be fetched and stored in JSON format, and thus can have multiple fields (like documents), and objects can be grouped into buckets, like the collections supported by document stores, with allowed and required fields defined on a per-bucket basis.
- Riak does not support indices on any fields except the primary key, the only thing you can do with the non-primary fields is fetch and store them as part of a JSON object. Riak lacks the query mechanisms of the document stores; the only lookup you can do is on primary key.

Riak supports replication of objects and distribution (sharding) by hashing on the primary key. It allows replica values to be temporarily inconsistent. Consistency is tunable by specifying how many replicas (on different nodes) must respond for a successful read and how many must respond for a successful write. This is per-read and per-write, so different parts of an applicatioon can choose different trade-offs.

Like Voldemort, Riak uses a derivative of MVCC where vector clocks are assigned when values are updated. Vector clocks can be used to determine when objects are direct descendents of each other or a common parent, so the Riak can often self-repair data that it discovers to be out of sync.

The Riak architecture is symmetric and simple. Consistent hashing is used to distribute data around a ring of nodes. Using good sharding technique, there should be many more "virtual" nodes (vnodes) than physical nodes (machines). Data hashed to vnode K is replicated on vnode K+1 … K+n where n is the desired number of extra copies (often n=1). There is no distinguished node to track status of the system: the nodes use a gossip protocol to track who is alive and who has which data, and any node may service a client request. Riak also includes a map/reduce mechanism to split work over all the nodes in a cluster.

The storage part of Riak is "pluggable": the key-value pairs may be in memory, in ETS tables, in DETS tables, or in Osmos tables. ETS, DETS, and Osmos tables are all implemented in Erlang, with different performance and properties.

One unique feature of Riak is that it can store "links" between objects (documents), for example to link objects for authors to the objects for the books they wrote. Links reduce the need for secondary indices, but there is still no way to do range queries.

The client interface to Riak is REST-based. There is also a programmatic interface for Erlang.

Here's an example of a Riak object described in JSON:

```
{
 "bucket":"customers",
 "key":"12345",
 "object":{
          "name":"Mr. Smith",
          "phone":"415-555-6524"
         }
 "links":[
         ["sales","Mr. Salesguy","salesrep"],
         ["cust-orders","12345","orders"]
```

```
        ]
 "vclock":"opaque-riak-vclock",
 "lastmod":"Mon, 03 Aug 2009 18:49:42 GMT"
}
```

Note that the primary key is distinguished, while other fields are part of an "object" portion. Also note that the bucket, vector clock, and modification date is specified as part of the object, and links to other objects are supported.

Redis

The Redis key-value data store is in an early stage of development. It was mostly written by one person, but there are now a number of active contributors.

Redis is written in C. A Redis server is accessed by a wire protocol implemented in various client libraries (which must be updated when the protocol changes). The client side does the distributed hashing over servers. The servers store data in RAM, but data can copied to disk for backup or system shutdown. System shutdown may be needed to add more nodes.

Like the other key-value stores, Redis implements insert, delete and lookup operations. Like Voldemort, it allows lists and sets to be associated with a key, not just a blob or string. It also includes list and set operations.

Redis does atomic updates by locking, and does asynchronous replication. It is reported to support 100K gets/sets per second. I believe this result is based on "batched" gets and sets (i.e., Redis's set or list operations).

Scalaris

Scalaris is functionally similar to Redis. It is written in Erlang at the Zuse Institute in Berlin, but is open source (like most of the systems we discussed). In distributing data over nodes, it allows key ranges to be assigned to nodes, rather than simply hashing to nodes. This means that a query on a range of values does not need to go to every node, and it also may allow better load balancing.

Like the other key-value stores, it supports insert, delete, and lookup. It does replication synchronously (both copies updated before completion) so data is guaranteed to be consistent. Scalaris also supports transactions with ACID properties on multiple objects. Data is stored in RAM, but replication and recovery from node failures provides durability of the updates. Nevertheless, a multi-node power failure would cause disastrous loss of data, and the virtual memory limit sets a maximum database size.

I'm also concerned about the scalability of Scalaris. Reads and writes must go to a majority of the replicas, and Solaris uses a ring of nodes, an unusual distribution and replication strategy that requires log(N) hops to read/write a key-value pair. Scalaris contributors report a favorable benchmark comparison in a wikipedia implementation, but it seemed an apples-to-oranges comparison to me.

Tokyo Tyrant

Tokyo Tyrant seems to be a small-scale operation. It was a sourceforge.net project, but the web site has recently been moved to one individual's web site. It is written in C.

There are actually two parts: a front end called Tokyo Tyrant, and a multi-threaded back-end server called Tokyo Cabinet. There are six different variations for the Tokyo Cabinet server: hash indexes in memory or on disk, B-trees in memory or on disk, fixed-size record tables, and variable-length record tables. The engines obviously differ in their performance characteristics, e.g. the fixed-length records allow quick lookups. There are slight variations on the API supported by these engines, but they all support common get/set/update operations. The documentation is a bit confusing on what functionality is supported by what server, but they claim to support locking, ACID transactions, a binary array data type, and more complex update operations to atomically update a number of concatenate to a string.

They also claim to support asynchronous replication with dual master or master/slave; I think this is implemented by the Tokyo Tyrant front end. There is mention of failover, but it appears that recovery of the failed node is manual. There does not appear to be automatic sharding; that could be done through modifications to Tokyo Tyrant to distribute over multiple servers.

They claim 2M gets/puts per second; probably this was using the in-memory hash engine.

Enhanced memcached

The memcached open-source system has been enhanced by several companies, including Schooner and Northscale, to include features analogous to the other key-value stores: replication, high availability, dynamic growth, persistence, backup, and so on. I may write more about these in a future version of this paper. Gear6 was also developing an enhanced memcached, but I'm not sure of its status. All of these systems are designed to be compatible with existing memcached clients; this is an attractive feature, given that memcached is many times more popular than any of the key-value stores I describe here.

Summary

- All these systems support insert, delete, and lookup operations.
- All of these systems provide scalability through key distribution over nodes.
- Voldemort, Riak, Tokyo Cabinet, and enhanced memcached can store data in RAM or on disk, with pluggable storage add-ons. The others store data in RAM, and provide disk as backup, or rely on replication and recovery so that a backup is not needed.
- Scalaris and enhanced memcached use synchronous replication, the rest use asynchronous.
- Scalaris and Tokyo Cabinet implement transactions, while the others do not.
- Voldemort uses multi-version concurrency control (MVCC), the others use locks.
- Voldemort and Riak currently seem to be the most popular of the key-value stores, but memcached extensions might prove most popular over time, because of their backward compatibility.

3. Document Stores

As discussed in the first section, document stores support more complex data than the key-value stores. The term "document store" is not ideal, because these systems store objects (generally objects without pointers, described in JSON notation), not necessarily documents. Unlike the key-value stores, they generally support multiple indexes and multiple types of documents (objects) per database, and they support complex values.

The document stores described here do not support locking nor synchronous copies to replicas, so there are no ACID transactional properties. Some authors suggest a "BASE" acronym in contrast to the "ACID" acronym:

- BASE = Basically Available, Soft state, Eventually consistent
- ACID = Atomicity, Consistency, Isolation, and Durability

The idea is that by giving up ACID constraints, you can achieve much higher performance and scalability. However, the systems differ in how much they give up.

SimpleDB

SimpleDB is part of Amazon's proprietary cloud computing offering, along with their Elastic Compute Cloud (EC2) and their Simple Storage Service (S3) on which SimpleDB is based. SimpleDB has been around since 2007. As the name suggests, its model is simple: SimpleDB has Select, Delete, GetAttributes, and PutAttributes operations on documents. SimpleDB is simpler than the other document stores, as it does not allow complex values, e.g. nested documents.

SimpleDB supports "eventual consistency" rather than transactions or synchronous replication. Like some of the earlier systems discussed, SimpleDB does asynchronous replication, and some horizontal scaling can be achieve by reading any of the replicas, if you don't care about having the latest version. Writes do not scale, because they must go asynchronously to all copies of a domain. If customers want better scaling, they must do so manually by sharding themselves.[2]

Unlike key-value datastores, and like the other document stores, SimpleDB supports more than one grouping in one database: documents are put into domains, which support multiple indexes. You can enumerate domains and their metadata. Select operations are on one domain, and specify constraints on one attribute at a time, basically in the form:

select <attributes> from <domain> where <list of attribute value contraints>

Different domains may be stored on different Amazon nodes.

Domain indexes are automatically updated when any document's attributes are modified. It is unclear from the documentation whether SimpleDB automatically selects which attributes to index, or if it indexes everything. In either case, the user has no choice, and the use of the indexes is automatic in SimpleDB query processing.

---

[2] SimpleDB does provide a BatchPutAttributes operation to update attributes of up to 25 documents in a single call.

SimpleDB is a “pay as you go” proprietary solution from Amazon. There are some built-in constraints, some of which are quite limiting: a 10 GB maximum domain size, a limit of 100 active domains, a 5 second limit on queries, and so on. Amazon doesn’t license SimpleDB source or binary code to run on your own servers. SimpleDB does have the advantage of Amazon support and documentation.

CouchDB

CouchDB has been an Apache project since early 2008. It is written in Erlang. It seems to be the most popular of the open source data stores; there is even a book available on CouchDB (see http://books.couchdb.org/relax/).

A CouchDB “collection” is similar to a SimpleDB domain, although the CouchDB data model is richer. Collections comprise the only schema in CouchDB, and secondary indexes must be explicitly created on fields in collections (and are automatically maintained by CouchDB). Any document may have any fields, and the field values can be a scalar (text, numeric, or boolean), or complex (a document or list).

Queries are done with what CouchDB calls “views”, which are defined with Javascript to specify field constraints. The indexes are B-trees, so the results of queries can be ordered or value ranges. Queries can be distributed in parallel over multiple nodes using a map-reduce mechanism. However, CouchDB’s view mechanism puts more burden on programmers than a declarative query language.

Like SimpleDB, CouchDB achieves scalability through asynchronous replication, not through partitioning (sharding). Reads can go to any server, if you don’t care about having the latest values, and updates must be propagated to all the servers. However, that could change with a new project called CouchDB Lounge that provides partitioning on top of CouchDB, see:

http:/tilgovi.github.com/couchdb-lounge/

Like SimpleDB, CouchDB does not guarantee consistency. Unlike SimpleDB, each client does see a self-consistent view of the database because CouchDB implements multi-version concurrency control on individual documents, with a Sequence ID that is automatically created for each version of a document. CouchDB will notify an application if someone else has updated the document since it was fetched. The application can then try to combine the updates, or can just retry its update and overwrite.

CouchDB also provides durability on system crash. All updates (documents and indexes) are flushed to disk on commit, by writing to the end of a file. (This means that periodic compaction is needed.) By default, it flushes to disk after every document update. Together with the MVCC mechanism, CouchDB’s durability thus provides ACID semantics at the document level.

Clients call CouchDB through a RESTful interface. There are libraries for various languages (Java, C, PHP, Python, LISP, etc) that convert native API calls into the RESTful calls for you.

CouchDB looks reasonable and mature. It even has some primitive database adminstration functionality, with an admin user. CouchDB’s asynchronous replication is

useful. However it lacks partitioning, and therefore lack of scalability for writes. I am further investigating CouchDB Lounge, to see if that is a viable.

MongoDB

MongoDB is a GPL open source document store written in C++ and sponsred by 10gen. It has some similarities to CouchDB: it provides indexes on collections, it is lockless, and it provides a document query mechanism. However, there are important differences:

- MongoDB supports automatic sharding.
- Replication in MongoDB is mostly used for failover, not for (dirty read) scalability. MongoDB is thus somewhere in between CouchDB's "eventual consistency" and the global consistency of a traditional DBMS: you can get local consistency on the up-to-date primary copy of a document.
- MongoDB supports dynamic queries with automatic use of indices, like RDBMSs. In CouchDB, data is indexed and searched by writing map-reduce views.
- CouchDB provides MVCC on documents, while MongoDB provides atomic operations on fields.

Atomic operations on fields are provided as follows:

- The update command supports "modifiers" that facilitate atomic changes to individual values: $set sets a value, $inc increments a value, $push appends a value to an array, $pushAll appends several values to an array, $pull removes a value from an array, and $pullAll removes several values from an array. Since these updates normally occur "in place", they avoid the overhead of a return trip to the server.
- There is an "update if current" convention for changing an object only if its value matches a given previous value.
- MongoDB supports a findAndModify command to perform an atomic update and immediately return the updated document. This is useful for impelmenting queues and other data structures requiring atomicity.

MongoDB indices are explicitly defined using an ensureIndex call, and any existing indices are automatically used for query processing. To find all products released last year costing under $100 you could write:

```
db.products.find({released: {$gte: new Date(2009, 1, 1,)}, price {'$lte': 100},})
```

If indexes are defined on the queried fields, MongoDB will automatically use them. MongoDB also supports map-reduce, which allows for complex aggregations across documents.

MongoDB stores data in a binary JSON-like format called BSON. BSON supports boolean, integer, float, date, string and binary types. Client drivers encode the local language's document data structure (usually a dictionary or associative array) into BSON and send it over a socket connection to the MongoDB server (in contrast to CouchDB, which sends JSON as text over an HTTP REST interface). MongoDB also supports a

GridFS specification for large binary objects, eg. images and vidoes. These are stored in chunks that can be streamed back to the client for efficient delivery.

MongoDB supports master-slave replication with automatic failover and recovery. Replication (and recovery) is done at the level of shards. Collections are automatically partitioned via a user-defined shard key. Replication is asynchronous for higher performance, so some updates may be lost on a crash.

Comparison of Operations

	Terminology	Create objects	Query objects
SimpleDB	Domain Item Attribute	PutAttributes(list of attribute/value pairs)	Select from domain where <conjunction of conditions on attributes>
CouchDB	Database Document Field	db.getDocument(key) *Creates if doesn't exist*	if (conjunction of conditions on fields) emit(key, list of field values) *defines a "view"*
MongoDB	Collection Document Key	collection.save(object) *Object defined in BSON*	collection.find(JSON-style document selector with comparison comparisons)

Other recent document stores include Terrastore, which is built on the Terracotta distributed JVM clustering product, and OrientDB, an Apache-licensed project that supports ACID and SQL as well. I plan to add these to this report in the future.

Summary

- The document stores are schema-less, except for attributes (which are simply a name, and are not pre-specified), collections (which are simply a grouping of documents), and the indexes defined on collections (explicitly defined, except with SimpleDB).
- Unlike the key-value stores, the document stores provide a mechanism to query collections based on multiple attribute value constraints. However, CouchDB does not support a non-procedural query language: it puts more work on the programmer and requires explicit utilization of indices.
- The document stores do not use explicit locks, and have weaker concurrency and atomicity properties than traditional ACID-compliant databases. They differ in how much concurrency control they do provide.
- Riak and CouchDB are written in Erlang, with RESTfull call processing. I'm uncertain how performant Erlang and the REST protocol will be under heavy throughput.
- Documents can be distributed over nodes in all of the systems, but scalability differs. All three systems can achieve scalability by reading (potentially) out-of-

date replicas. MongoDB can obtain scalability without that compromise, and can scale writes as well, through automatic sharding and through atomic operations on documents. CouchDB might be able to achieve this write-scalability with the help of the new CouchDB Lounge code.

4. Extensible Record Stores

The extensible record stores all seem to have been motivated by Google's success with BigTable. Their basic data model is rows and columns, and their basic scalability model is splitting both rows and columns over multiple nodes:

- Rows are split across nodes through conventional sharding, on the primary key. They typically split by range rather than a hash function (this means that queries on ranges of values do not have to go to every node).
- Columns of a table are distributed over multiple nodes by using "column groups". These may seem like a new complexity, but column groups are simply a way for the customer to indicate which columns are best grouped together.

These two partitionings (horizontal and vertical) can be used simultaneously on the same table.

The column groups must be pre-defined with the extensible record stores. However, that is not a big constraint, as new attributes can be defined at any time. Rows are not that dissimilar from documents: they can have a variable number of attributes (fields), the attribute names must be unique, rows are grouped into collections (tables), and an individual row's attributes can be of any type.[3]

Although the systems were patterned after BigTable, none of the extensible records stores (indeed, none of the systems in this paper or perhaps the world) come anywhere near to BigTable's scalability. At Google, BigTable runs on tens of thousands of nodes with incredible transaction rates. The best I've seen on any of the other systems is hundreds of nodes at much lower transaction rates, especially for writes. Unfortunately, BigTable is not available for sale or as open source, it is only "for rent" (a subset as part of the Google App Engine), as far as I know. Google's sales model seems to be similar to Amazon's with SimpleDB.

It is worthwhile reading the BigTable paper for background on extensible record stores, and on the challenges with scaling to tens of thousands of nodes:

http://labs.google.com/papers/bigtable.html

HBase

HBase is an Apache project written in Java. It is patterned directly after BigTable:

- HBase uses the Hadoop distributed file system in place of the Google file system. It puts updates into memory and periodically writes them out to files on the disk.

---

[3] However, that CouchDB and MongoDB support complex types, such as nested objects, while the extensible record stores generally support only scalar types.

- The updates go to the end of a data file, to avoid seeks. The files are periodically compacted. Updates also go to the end of a write ahead log, to perform recovery if a server crashes.
- Row operations are atomic, with row-level locking and transactions. There is optional support for transactions with wider scope. These use optimistic concurrency control, aborting if there is a conflict with other updates.
- Partitioning and distribution are transparent; there is no client-side hashing or fixed keyspace as in some NoSQL systems. There is multiple master support, to avoid a single point of failure. MapReduce support allows operations to be distributed efficiently.
- HBase uses B-tree indices, so range queries and sorting are fast.
- There is a Java API, a Thrift API, and REST API. JDBC and ODBC support has recently been added (see hbql.com).

The initial prototype of HBase released in February 2007. The support for transactions is attractive, and unusual for a NoSQL system. I'm not sure of the performance of HBase, but with the right JVM, Java performance can be quite good.

HyperTable

HyperTable is written in C++. Its was open sourced by Zvents. It doesn't seem to have taken off in popularity yet, but Baidu recently became a project sponsor, that should help.

Hypertable is very similar to HBase and BigTable. It uses column families that can have any number of column "qualifiers". It includes timestamps on data, and uses MVCC. It requires an underyling distributed file system such as Hadoop, and a distributed lock manager. Tables are replicated and partitioned over servers by key ranges. Updates are done in memory and later flushed to disk.

Hypertable supports a number of programming language client interfaces. It uses a query language named HQL.

Cassandra

Cassandra is similar to the other extensible record stores in its data model and basic functionality. It has column groups, updates are cached in memory and then flushed to disk, and the disk representation is periodically compacted. It does partitioning and replication. Failure detection and recovery are fully automatic. However, Cassandra has a weaker concurrency model than some other systems: there is no locking mechanism, and replicas are "eventually consistent", being updated asynchronously.

Like HBase, Cassandra is written in Java, and used under Apache licensing. It was open sourced by Facebook in 2008. It was designed by a Facebook engineer and a Dynamo engineer, and is described as a marriage of Dynamo and BigTable. Cassandra is used by Facebook as well as other companies, so the code is reasonably mature.

Client interfaces are created using Facebook's Thrift framework:

http://incubator.apache.org/thrift/

Cassandra automatically brings new available nodes into a cluster, uses the phi accrual algorithm to detect node failure, and determines cluster membership in a distributed fashion with a gossip-style algorithm.

Cassandra adds the concept of a "supercolumn" that provides another level of grouping within column groups, I'm not sure how useful this is. Databases (called keyspaces) contain column families. A column family contains either supercolumns or columns (not a mix of both). Supercolunns contain columns. As with the other systems, any row can have any combination of column values (i.e., rows are variable length and are not constrained by a table schema).

Cassandra uses an ordered hash index, which should give most of the benefit of both has and B-tree indexes: you know which nodes could have a particular range of values instead of searching all nodes. However, sorting would still be slower than with B-trees.

Cassandra has reportedly scaled to about 100 machines in production at Facebook, perhaps more by now. Cassandra seems to be gaining a lot of momentum as an open source project, as well. However, I've had difficulty finding public documentation on Cassandra, except for some articles and presentations. I am researching further.

Cassandra's eventual-consistency model might not work for some applications. However, "quorum reads" do provide a way to get the latest data, and writes are atomic within a column family. There is also some support for versioning and conflict resolution.

Summary

- The extensible record stores are patterned after BigTable. They are all quite similar.
- Cassandra focuses on "weak" concurrency (via MVCC) and HBase and HyperTable on "strong" consistency alternative (via locks and logging).
- Cassandra's backing by Facebook and Hypertable's backing by Baidu might help them dominate, but it's too early to tell.

5. Scalable Relational Databases

Unlike the other datastores, relational databases have a complete pre-defined schema, a SQL interface, and ACID transactions. RDBMSs have not achieved the scalability of the some of the previously-described datastores. To date, MySQL Cluster seems most scalable, linear up to about 50 nodes depending on application. However, I believe it is *possible* for a relational database to provide more scalability, with certain provisos:

- *Small-scope operations:* As we've noted, operations that span many nodes, e.g. joins over many tables, will not scale well with sharding.
- *Small-scope transactions:* Likewise, transactions that span many nodes are going to be very inefficient, with the communication and two-phase commit overhead.

I include scalable RDBMSs as a viable alternative in this paper. A scalable RDBMS does not need to *preclude* larger-scope operations and transactions, but they must

penalize a customer for these operations *only if they use them.* In fact, scalable RDBMSs then have an advantage over the other datastores, because you have the convenience of the higher-level SQL language and ACID properties, but you only pay a price for those when they span nodes.[4]

MySQL Cluster

MySQL Cluster was developed about 10 years ago, and has been part of the MySQL release since 2004. MySQL cluster works by replacing InnoDB with a distributed layer called NDB. MySQL Cluster shards data over multiple database servers (this is a "shared nothing" architecture). Every shard is replicated on two machines, to support recovery.

Although MySQL Cluster seems to scale to more nodes than Oracle RAC, Parallel DB2, and other RDBMSs to date, it still runs into bottlenecks after a few dozen nodes. Work continues on MySQL Cluster, so this may improve. Much of the overhead may be in inter-node communication and synchronization.

ScaleDB

ScaleDB is a new derivative of MySQL underway. Like MySQL Cluster, it replaces the InnoDB engine, and uses clustering of multiple servers to achieve scalability. ScaleDB differs in that it requires disks shared across nodes. Every server must have access to every disk. This architecture has not worked well for Oracle RAC, so I am not optimistic.

ScaleDB's partitioning (sharding) over servers is automatic: more servers can be added at any time. Server failure handling is also automatic. ScaleDB redistributes the load over existing servers.

ScaleDB supports ACID transactions and row-level locking. It has multi-table indexing (which is possible due to the shared disk).

Drizzle

Another MySQL derivative underway is Drizzle. It is derived from code forked off of MySQL 6.0 by Brian Akers, but some MySQL features are not supported (stored procedures, prepared statements, views, triggers, grants).

Drizzle has been "stripped down" from full MySQL for performance, and shards data over multiple nodes, for scalability.

The source code is available on http://launchpad.net/drizzle, but Drizzle is not yet completed: there is not yet an official first release, and it is not yet supported.

NimbusDB

NimbusDB is another new relational system, from Jim Starkey. It uses MVCC and distributed object based storage. SQL is the access language, with a row-oriented query optimizer and AVL tree indexes.

---

[4] To be precise: in theory, it seems to me that RDBMS can provide SQL and transactions without penalty as long as they don't span nodes. However, this has not yet been proven in practice.

MVCC provides transaction isolation without the need for locks allowing large scale parallel processing. Data is horizontally segmented row-by-row into distributed objects, allowing multi-site, dynamic distribution.

There is not much information available on NimbusDB yet, but there are slides at

http://www.nimbusdb.com/SpecialRelativity.pdf

VoltDB

VoltDB is a new RDBMS designed for high performance via a number of means:

1. Tables are partitioned over multiple servers, and clients can call any server.
2. The database is stored in memory, along with an undo log. Disk waits are eliminated, because the data is always available.
3. Transactions are encapsulated in stored procedures, and are executed in the same order on a node and on replica node(s). Replicas provide protection from failure of an (in-memory) database node.
4. Lock overhead is avoided, because only one process executes the (serialized) transactions on each partition.

VoltDB is still under development. I will report more on it when information becomes available.

Summary

- MySQL Cluster uses a "shared nothing" architecture for scalability. It seems to be the most scalable solution at present.
- ScaleDB, NimbusDB and Drizzle are in an early stage of development, I can't say much about them yet.
- VoltDB looks promising because of its bottom-up redesign for performance, but I likewise more information. An early release is now available.
- In theory, RDBMSs should be able to deliver scalability, but they haven't gotten there yet.

6. Some Use Cases

No one of these data stores is best for all uses. Your prioritization of features will be different depending on the application, as will the type of scalability required. I assume you need some kind of scalability across machines: otherwise your best solution may be a traditional RDBMS running on a single server.

In this section, I'll give some examples of use cases that fit well with the different data store categories. At the moment, I've only listed a few examples. I'll elaborate the use cases over time.

Key-value Store Examples

Key-value stores are generally good solutions if you have only one kind of object, and you only need to look objects up based on one attribute. Key-value stores are generally the simplest to use, especially if you're already familiar with memcached.

For example, suppose you have a web application that does many RDBMS queries to create a tailored page when a user logs in. Suppose it takes several seconds execute those queries, and the user's data is rarely changed, or you know when it changes because updates go through the same interface. Then you might want to store the user's tailored page as a single object in a key-value store, represented in a manner that's efficient to send in response to browser requests, and index these objects by user ID. If you store these objects persistently, then you may be able to avoid RDBMS queries altogether, reconstructing the objects only when a user's data is updated.

Even in the case of an application like Facebook, where a user's home page changes based on updates made by the user as well as updates made by others, it may be possible to execute RDBMS queries just once when the user logs in, and for the rest of that session show only the changes made by that user (not by other users). Then, a simple key-value store could still be used.

You could use key-value stores to do lookups based on multiple attributes, by creating additional key-value indexes that you maintain yourself. However, at that point you probably want to move to a document store.

Document Store Examples

If you have multiple different kinds of objects (say, in a DMV application with vehicles and drivers), and you need to look up objects based on multiple fields (for example, a driver's name, license number, owned vehicle, or birth date), then you'll want a document store.

Another factor to consider is what level of concurrency guarantees you need. If you can tolerate the "eventually consistent" model and limited atomicity, the document stores should work well for you. That might be the case in the DMV application, e.g. you don't need to know if the driver has new traffic violations in the past minute, and it would be quite unlikely for two DMV offices to be updating the same driver's record at the same time. But if you require that data be accurate and atomically consistent, e.g. if you want to lock out logins after three incorrect attempts, then you can't afford to be even a few seconds out of date. In that case, you need to use document stores carefully, reading enough replicas to guarantee that you have the latest value.

Extensible Record Store Examples

The use cases for extensible record stores are similar to those for document stores: multiple kinds of objects, with lookups based on any field. However, the extensible record store projects generally seem to be aimed at higher throughput, and may provide stronger concurrency guarantees, at the cost of slightly more complexity than the document stores.

Suppose you are storing customer information for an eBay-style application, and you want to partition your data both horizontally and vertically:

- You might want to cluster customers by country, so that you can efficiently search all of the customers in one country.
- You might want to separate the rarely-changed "core" customer information such as customer addresses and email addresses in one place, and put certain frequently-updated customer information (such as current bids in progress) in a different place, to improve performance.

Although you could do this kind of horizontal/vertical partitioning yourself on top of a document store by creating multiple collections for multiple dimensions, the partitioning is most easily achieved with an extensible record store like HBase or HyperTable.

Scalable RDBMS Examples

If you need to query across types or if you sometimes need transactions that span multiple records, you'll probably want an RDBMS. If you need scalability across dozens of machines right now, your best solution would probably be MySQL Cluster.

As an example, suppose in the DMV application that you need to be able to find all of the red cars that are registered in San Francisco that are owned by someone aged 30 to 50 years who works in Santa Clara County. With queries like these, a scalable RDBMS is the most convenient solution.

As another example, you might require or want to use some existing tools based on SQL, such as Crystal Reports. Again, a scalable RDBMS is probably the best solution, assuming you can achieve the performance you require with a system like MySQL Cluster.

Note that if more scalable RDBMSs are successful, then they may provide a better solution for an even wider range of applications: the need for NoSQL solutions could be obviated.

7. Overall Summary

Here's a table listing storage models, concurrency models, and other aspects of these systems.

	Conc Control	Data Storage	Repli-cation[5]	Tx[6]	Comments
Redis	Locks	RAM	Async	N	
Scalaris	Locks	RAM	Sync	L	
Tokyo	Locks	RAM or	Async	L	B-trees and hash. Manual

[5] Replication indicates whether the mirror copies are always in sync, or are updated asynchronously. BigTable updates georgraphically remote copies asynchronously (which is really the only practical solution).

[6] An "L" indicates transactions limited to a single node or record.

		disk			sharding.
Voldemort	MVCC	RAM or BDB	Async	N	Open source Dynamo clone.
SimpleDB	None	S3	Async	N	No automated sharding, scales by replication.
Riak	MVCC	Pluggable	Async	N	No indices nor queries on secondary key fields.
MongoDB	Field-level	Disk	Async	N	JSON queries. Sharding new.
Couch DB	MVCC	Disk	Async	N	No sharding, possibly remedied by Lounge.
HBase	Locks	Hadoop	Async	L	B-trees. Based on Hadoop, logging. BigTable clone.
HyperTable	Locks	Files	Sync	L	Very similar to BigTable.
Cassandra	MVCC	Disk	Async	L	Facebook spinoff. BigTable clone except MVCC.
BigTable	Locks+ stamps	GFS	Sync+ Async	L	B-trees. Sync locally, async remotely.
MySQL Cluster	Locks	Disk	Sync	Y	Limited scalability. Replaces InnoDB with "NDB".
ScaleDB	Locks	Disk	Sync	Y	Shared disk. Also replaces InnoDB.
Drizzle	Locks	Disk	Sync	Y	Under development. Also replaces InnoDB.
NimbusDB	MVCC	Disk	Sync	Y	Under development.
VoltDB	Lock-equiv	RAM	Sync	Y	Under development.

Here is some more information on the systems and their sources, including a rough measure of popularity on the net (thousands of hits on Google as of January) and a measure of code maturity (my judgment based on developer comments):

	Mature	K-hits	License	Lang	Web site for more information
Redis	Low	200	BSD	C	code.google.com/p/redis
Scalaris	Med	200	Apache	Erlg	code.google.com/p/scalaris
Tokyo	Low	100	GPL	C	tokyocabinet.sourceforge.net
Voldemort	Med	100	None	Java	project-voldemort.com
SimpleDB	High	300	Prop	N/A	amazon.com/simpledb
Riak	Med	400	Apache	Erlg	riak.basho.com
MongoDB	Med	100	GPL	C++	mongodb.org
Couch DB	High	1M	Apache	Erlg	couchdb.apache.org
HBase	Med	500	Apache	Java	hadoop.apache.org/hbase
HyperTable	Med	200	GPL	C++	hypertable.org
Cassandra	Med	600	Apache	Java	incubator.apache.org/cassand

					ra
BigTable	High	1M+	Prop	C++	labs.google.com/ papers/bigtable.html
MySQL Cluster	High	1M+	GPL	C++	mysql.com/cluster
ScaleDB	Med	100	GPL	C++	scaledb.com
Drizzle	Low	200	GPL	C++	launchpad.net/drizzle
VoltDB	Low	20	GPL	Java+	voltdb.com

I believe that a few of these systems will gain critical mass and key players, and will pull away from the others by next year. At that point, users and open source contributors will likely migrate to those players.