2008 BBC Presentation Notes

Notes from Jan Lehnardt’s Presentation on CouchDB to the BBC, 2008-08-27

by Scott Beatty

September 6th, 2008

On August 27, 2008, Jan Lehnardt gave a presentation (video, slides) on CouchDB to BBC web developers in London. As I listened to the recorded video, I realized that there was a lot of good information. In an effort to learn more, I have played it again, taken some notes, and shared them here. Most of the time I have paraphrased, but sometimes I have quoted (without using quotation marks). At the end of each note I list the time at which the subject begins.

Total run time: (68:02)

Contents

State of CouchDB

There’s still a lot of work yet to be done (for CouchDB). (0:40)

CouchDB is deployable. (0:45)

Comparison to Relational Databases

It is not a good idea to use relational databases all the time. Use the right tool for the job. (1:44)

CouchDB is very easy to integrate. (2:25)

CouchDB is a database built for the future. (2:46)

Relational databases (RDBMS) were originally built for single users on single machines doing single operations. Data had to be prepared in such a way that execution was as fast as possible. (2:56)

Relational databases weren’t designed for today’s world with thousands of simultaneous users. (3:25) Today there is no fixed structure of data. (4:02) We used to have big computers. Now we have a lot of small computers. (4:19) Disk space is getting cheap. CouchDB trades off everything (other resources) for disk space. (4:47)

RDBMS: design and work with schemas. CouchDB: just store data. (5:32)

Object Relational Mappers (ORM) don’t handle lots of load well. (6:31)

Using a relational database creates a lossy translation between the data object that you want to store and how it is stored on the disk. (6:38)

Most data that occurs in the “real world” (quotes mine) is not inherently relational. (7:06)

Documents

CouchDB stores data in documents, which have no schema. One document can have a field that another one doesn’t have. (8:01)

The documents in CouchDB are actual representations of the data objects. There is no lossy translation. (8:22)

Documents can also be out of date. (9:12)

CouchDB stores data (encodes objects) in the JSON format. It is language-agnostic. Data is serialized and de-serialized to and from that format. It is lighter and easier to read (by humans) than XML. (10:25)

Each CouchDB document has an unique, constant identifier with a changing revision value. (12:09)

JSON can include all the native data types (e.g., numbers, strings, arrays, booleans) in a programming language. (14:07)

Documents can have attachments. (14:25)

HTTP REST API

You talk to CouchDB using an HTTP REST API. (14:56)

Four basic operations of working with a document:

- Create: HTTP PUT /db/docid
- Read: HTTP GET /db/docid
- Update: HTTP POST /db/docid
- Delete: HTTP DELETE /db/docid (15:56)

Every CouchDB document has an URI. (16:13)

HTTP HEAD is also available. (16:38)

Your programming language and application can handle the details of talking HTTP. (16:51)

There is no definite PHP library (to choose a particular language) for talking to CouchDB. (17:33)

Views

To make CouchDB a database it is more than a versioned object instance store. It has views. (17:56)

Views: filtering, collating, and aggregating the stored data. Filtering: subsets. Collating: ordering. Aggregating: calculations. (18:11)

Views are powered by MapReduce. It is a concept from the functional programming world. Google relies on it for search queries. It is a massively parallelizeable operation, which makes scaling possible. (18:38)

To use MapReduce you provide CouchDB code for a map function and code for a reduce function. CouchDB executes these functions on your data in a two-stage process. The reduce function works on the results of the map function. The default language for MapReduce in CouchDB is Javascript. (19:44)

MapReduce is good for handling huge amounts of data by breaking up the work into chunks and assigning it to different machines. (23:00)

Views are built incrementally, on demand. (23:52)

View computations on a single document are only done once, and the result is stored or cached in indexes, making subsequent computations much faster than the initial computations. Changes are integrated incrementally, on demand. The view module of CouchDB is a separate module from the data storage part. The internals of CouchDB are optimized to make view updating very fast. (24:03)

When you write to a CouchDB database it doesn’t touch the views. It doesn’t spend time updating indexes. There is lazy evaluation for the view. You decide when to spend the CPU time to do that. (25:47)

The second step of Reduce, for aggregation, is optional. Map does filtering and collation. Aggregation isn’t always needed, or it might also be done in Map. (26:31)

Both Map and Reduce can work on chunks of data on parallel machines. (27:03)

Replication

Replication is the answer to data synchronization between machines. (29:57)

Network connections might not always be available. (31:15)

Replication is good for data synchronization, load balancing, and failover. (31:37)

You can take your data with you. (31:48)

CouchDB replication is like rsync for databases. (31:56)

CouchDB replication is uni-directional. To get replication in the other direction just call for the reverse. A->B, then B->A. (32:43)

CouchDB can replicate data over any number of nodes. (33:44)

Unlike MySQL, CouchDB replication does not need to be a constant process. In CouchDB, replication is trigger-based. (34:19)

Problem: Changing same piece of data (e.g., same document) on more than one node before synchronization. This is called a conflict. CouchDB does automatic conflict detection and resolution. It will flag a document as conflicting (like in a version control system). It has an algorithm that selects one of the revisions as the latest, and the other conflicts are stored as previous revisions. Each node will have the same set of winning and losing revisions in the same order. This is a data consistency guarantee for replication. In any case of a conflict you then need to go in and resolve, and then replicate the resolution to all the other nodes in the cluster. (34:48)

Each node does not need to talk to any other nodes in order to come up with the same list of winning and losing revisions. (37:32)

Most conflicts can be resolved by the application. For example, if timestamps are used then they can be used for deciding, or if different attributes are changed then they can be merged. Application can not resolve a conflict in the same attribute without timestamps. These conflicts need to be presented to user for resolution. (38:09)

There is a view for showing all conflicts. You could then use a cronjob to attempt to resolve the conflicts. There is a notification system in CouchDB that might be useful for this subject, too. (38:48)

If you resolve a conflict it might create another conflict. There can also be multiple conflicts. You need to resolve conflicts from the bottom up. Eventually everything will be conflict-free. There is no deadlock. Guaranteed. (40:17)

Replication scales very well. Like the view server, CouchDB maintains a sequence number for the database, so it only has to work on the latest differences. (40:55)

Built for the Future

CouchDB is written in Erlang. Erlang is a functional programming language, a virtual machine, and a set of standard libraries. It provides very fault-tolerant concurrent operation. The language is set up that you create modules, and you send messages to modules, and the modules live in processes. From the language level you don’t care where the processes live; you just send messages to them. The virtual machines can live on different hardware (nodes). If a node dies then the virtual machine just sends messages to other nodes. (41:21)

CouchDB uses Erlang because the problems that it solved for telco are the same for the Web today. (44:39)

ACID compliant data store - means data is safely stored in the database. CouchDB has it. (44:59)

Non-locking multi-version concurrency control (MVCC). Reads are not corrupted by writes during the read. Concurrent requests can continue to be served. CouchDB does this with versioning, and it never overwrites data. No reads are blocked. Writes are put into a buffer pool so that they happen serially. Changes are appended to the end of the database file. Only one write can happen at one time. So, to delete you add data. Compaction can be used to get rid of previous revisions, and to reclaim disk space. (So you can’t use CouchDB as a version control system.) CouchDB makes sure that the data is actually stored on disk “on filesystems that implement fsync correctly”. If there is a crash CouchDB does not have to do a consistency check when restarted. Crash-resistant. Append-only. Need compaction. (45:40)

CouchDB can serve many read requests per second using very little memory. (50:09)

History

Damien Katz started CouchDB. Previously, he worked on the core database in Lotus Notes. He quit his job to work on CouchDB full-time, originally in C++ and then Erlang, and then released it as an open source project. After more than two years he was hired by IBM to work full-time on CouchDB, with the code within the Apache Software Foundation, under the Apache 2.0 License. (50:51)

Resources

- http://incubator.apache.org/couchdb/
- http://wiki.apache.org/couchdb/
- http://damienkatz.net/
- http://jan.prima.de/plok/
- http://blog.racklabs.com/?p=74

(53:37)

CouchDB is looking for Erlang hackers. (54:02)

Answers to Audience Questions

You can do full-text search with a Lucene reference implementation. You can use any indexing technology, e.g., Sphinx, geo-spatial indexing. There is a plug-in API. You can't do everything in views, though you can do a lot. (54:41)

Concurrent connections: You can have as many as the number of sockets that your operating system can handle, but latency will go up, so requests will take a little longer if there is a lot of load. There is no point where it crashes and burns. This is a property from Erlang. (55:55)

Constraints and transactions: Constraints: Not yet. You will be able to provide a document validation function that will be evaluated on write, with the ability to stop the write. Transactions: CouchDB has single-node transactions for single HTTP requests. Either the request will succeed or not. Beyond the single-node you won't be able to ensure that the request is replicated to all other nodes, so there are no multi-node transactions. They are hard to do in a distributed environment. CouchDB has optimistic locking. (57:07)

Scaling: Currently, the primary means is replication. Eventually, you will be able to do database partitioning. If you have a lot of data that won't fit on a single box then CouchDB will be able to transparently partition it out. Also with regards to scaling, they have not yet tried running the Erlang virtual machine in different places, e.g., some machines dedicated to reads, some to writes, and some to views. There are only four and half developers working on CouchDB. Only one who is full-time, for the rest it is hobby-work. Developer time is greatly appreciated. (58:58)

Map/Reduce on a multi-core machine: It is not yet spread to more than one core, but it will be. (60:08)

When do you trigger compaction? Does database performance slow down? Not that we've seen. CouchDB might not be the best choice for highly volatile data. Maybe use memcache, then periodically synchronize. (62:10)

Does compaction lock anything? No, it is live compaction. It tries to make a running copy of the latest revisions of your documents. This can take some time if there is a high write load, so it is possible it might time out. You need to work out the metrics for your hardware and your dataset. Once all the current revisions are copied then CouchDB switches to the new database and then throws away the old one. This is done live, with no downtime. (62:49)

Stacking views? No, single. It is not a priority for CouchDB at this time to add the functionality to support views on views. CouchDB is not always the right tool. Revisions are used for optimistic locking, not for revision control. There are edge cases in distributed replication, and no elegant solution has yet been found. Documents are replicated, but views are not. Views need to be recreated on each node. (63:51)

Can you safely backup a live database? Yes, you can just make a filesystem copy of the database. (66:22)

Are there tradeoffs to having some documents without the same fields? There is complexity (and flexibility) in no schema stores, and you have to deal with it at some point. In CouchDB you don't have to deal with that up front. It is usually handled in the view functions. We can recommend best practices to deal with that. (67:06)