Couchbase

Making the Shift from Relational to NoSQL

How to change the way you think about data modeling

While the hype surrounding "NoSQL" (non-relational) database technology has become deafening, there is real substance beneath the often exaggerated claims. But like most things in life, the benefits come at a cost. Developers accustomed to data modeling and application development against relational database technology will need to approach things differently. This white paper highlights the differences between a relational database and a distributed document-oriented database, the implications for application development, and guidance that can ease the transition from relational to NoSQL database technology.

Why make the shift?

Change is hard and rarely undertaken unless it alleviates significant pain. The white paper Why NoSQL provides a comprehensive look at the motivating challenges that have led to the emergence and rapid adoption of NoSQL database technology. In a nutshell, the transition has been spurred by the need for flexibility – both in the scaling model and the data model.

Scaling model

Relational database technology is a “scale up” technology – to add capacity (whether data storage or I/O capacity) one gets a bigger server. The modern approach to application architecture is to scale out, rather than scale up. Instead of buying a bigger server, you add more commodity servers, virtual machines or cloud instances behind a load balancer. Conversely, capacity can be easily removed when no longer required. While scaling out is already common at the application logic tier, database technology is only now catching up.

Data model

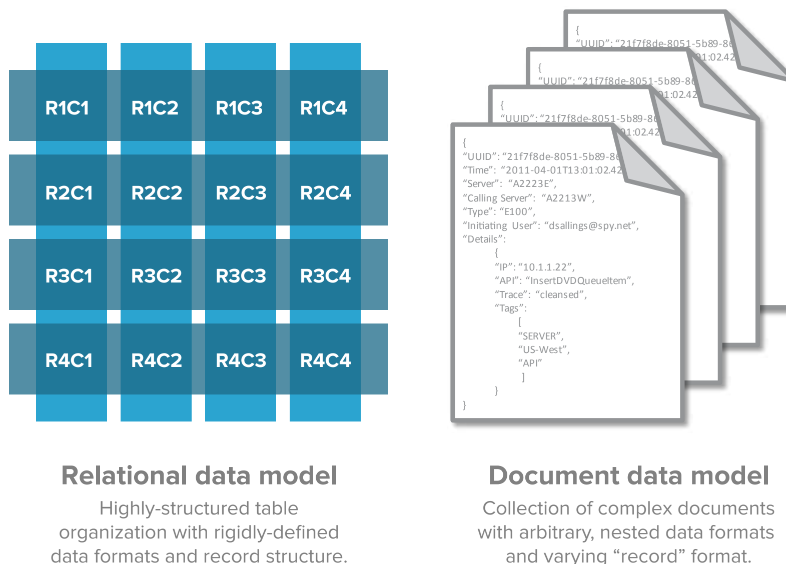
The scale-out deployment benefits of NoSQL technology frequently get the most attention, but equally important are the benefits afforded by a schemaless approach to data management. With a relational database, you must define a schema before adding records to the database. Each record added to the database must adhere strictly to this schema with its fixed columns and data types. Changing the database schema, particularly when dealing with a partitioned relational database spread across many servers, is difficult. If your data capture and management needs are constantly evolving, a rigid schema quickly becomes blocker to change.

NoSQL databases (whether key-value, document, column-oriented or otherwise) scale out, and they don’t require schema definition prior to inserting data nor a schema change when data capture and management needs evolve.

The rest of this paper will focus on distributed *document-oriented* NoSQL database technology – with Couchbase and MongoDB being the two most visible and widely adopted examples.

The relational vs. document-oriented data model

The figure below compares four records from a relational database with four from a document-oriented database.



Relational data model

Highly-structured table organization with rigidly-defined data formats and record structure.

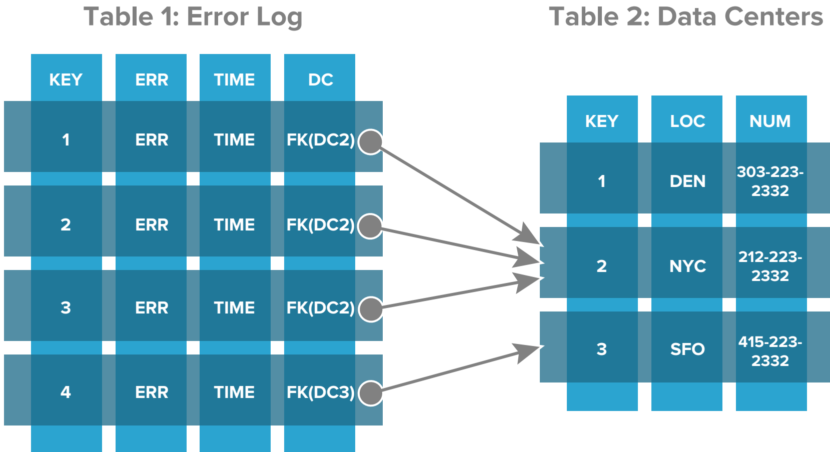
Document data model

Collection of complex documents with arbitrary, nested data formats and varying “record” format.

Relational data model

As shown above, each record in a relational database conforms to a schema – with a fixed number of fields (columns) each having a specified purpose and data type. Every record is the same. If you wish to capture different data in the future, the database schema must be revised.

Additionally, the relational model is characterized by database normalization, where large tables are decomposed into smaller, interrelated tables. The figure below illustrates the concept:



In the above example, the database is used to store error log information. Each error record (each row in Table 1) consists of an error number (ERR), the time the error occurred (TIME) and the datacenter (DC) in which the error occurred. Instead of repeating all the datacenter information in each error record (location, phone number), each error record points to a row in the in the Data Centers Table (Table 2) which includes the location of the datacenter (LOC) and the phone number (NUM).

In the relational model, records are "striped" across multiple tables, with some data shared by multiple records (multiple error records share the same data center information). The upside is that there is less duplicated data in the database. The downside is that a change in a single record can mean locking down many tables simultaneously to ensure a change doesn't leave the database in an inconsistent state. ACID transactions can be complex on a relational database because the data, even of a single record, is spread about. This complex web of interrelationships between shared data items is what makes it so difficult to distribute relational data across multiple servers and can lead to performance challenges both reading and writing data.

When storage resources were expensive and scarce, the tradeoffs made sense. But the price of storage has dropped precipitously over the last 40 years, to say the least. For many, the tradeoff calculus no longer makes sense. Using more storage in exchange for better application performance and the ability to easily distribute workloads across machines is now the best choice for many applications.

Document data model

Use of the term “document” is a bit confusing. A document-oriented database really has nothing to do with “documents” in the classical sense of the word. It doesn’t mean books, letters or articles. Rather, a document in this case refers to a data record that is self-describing as to the data elements it contains. XML documents, HTML documents and JSON documents are examples of “documents” in this context. Couchbase Server is a document-oriented database that uses JSON as the document format. In Couchbase, error records would look like this:

```
{
    “ID”: 1,
    “ERR”: “Out of Memory”,
    “TIME”: “2004-09-16T23:59:58.75”,
    “DC”: “NYC”,
    “NUM”: “212-223-2332”
}
{
    “ID”: 2,
    “ERR”: “ECC Error”,
    “TIME”: “2004-09-16T23:59:59.00”,
    “DC”: “NYC”,
    “NUM”: “212-223-2332”
}
```

As can be seen, the data is denormalized. Each record contains a complete set of information related to the error without external reference. The records are self-contained. This makes it very easy to move the entire record to another server – all the information simply comes along with it. There is no concern about having parts of the record in other tables left behind. And because only the self-contained record (document) needs to be updated when changes are made (versus changing entries in multiple tables simultaneously), ensuring ACID compliance is far easier, at record boundaries. Performance is also increased on reads.

But complete data denormalization is not required in a document database, as highlighted in the next section. In fact, in this particular example, maintaining documents representing each datacenter and simply referencing those from each error record would probably be the right decision. Separation would eliminate duplication and allow quick changes to information shared across many records (e.g. if the telephone number for the datacenter changed, there would be no need to go update it in every error record). Ultimately, however, data modeling decisions are dependent on the use case and planned update patterns.

Document modeling: rules of thumb

It takes a while to unlearn habits. But do not fear: by understanding alternatives you will be able to make more efficient use of your trusted knowledge as well. After all, the tool best suited for the job will leave you with the least headache. If you know more tools, you can choose more wisely.

Models

In an application, data objects are a central construct – the model layer in Model-View-Controller (MVC). These are the documents that hold your data and let you manipulate it. If a blog has posts and comments, these are likely two different models. Ideally, you should have a separate document for every post and every comment.

When looking at an existing application, stop at the Object-Relational Mapping (ORM) layer. Instead of splitting your models up into tables and rows, turn them into JSON and make them a document. Each document gets a unique ID by which you can find it later. Done.

(Primary) Keys In the NoSQL world the document ID is the one and only key to a document. They are roughly equivalent to primary keys in a relational database. Usually an ID can only appear once in a database (different NoSQL solutions have different names for these: buckets, collections, tables etc. The idea is roughly similar to a table in an RDBMS. You can have many per server).

Some NoSQL database systems sort data by ID. Data with nearby IDs can be accessed more efficiently than IDs that are all over the place. Keeping data that you tend to access at the same time closer together makes your application faster.

The larger point here though is that an ID-lookup is extremely fast, and by selecting clever IDs you can make your life a lot easier. One example is the use of prefixes (user:com.example:123) to group your documents.

Multiple places and editability

Suppose you have a piece of data that shows up all over your application but you still want to be able to edit that data. For example, a photo on flickr has a title. The photo can show up in your photo stream, in sets, collections, groups on your flickr front page and in many more places.

Usually, a photo's title is shown with the photo. You could create a document for each occurrence of the photo in each of the places. But then, if you change the title of your picture, you need to update a bunch of documents. If you know this is a bounded number (no more than 10-100 e.g.) and the renaming doesn't have to happen simultaneously in all places (which means an asynchronous background task could do the renames), using separate docs for each occurrence can work fine.

However, if the number of copies is unbounded and could potentially lead you to update thousands of documents, that approach probably won't work. Instead, you would want to keep the title and perhaps other identifying data in a single "photo information" document and create a separate "photo placement" document for each place the photo appears (these "photo placement" documents would each point to the photo's information document). Now when you display a photo you will make two lookups: one for the placement document and then another for the photo information document. If you want to change the title of a photo, you just edit the single photo information document and it will be changed everywhere on your site.

There is a special trick you can use in Couchbase: Using view collation you can have a single query answering for all the data you need. Christopher Lenz wrote an outstanding blog post on this topic, highlighting three approaches to modeling using the blog example.

With views, Couchbase Server allows one to keep a single canonical source of a piece of data while having it show up in many different places.

In the RDBMS world you are taught to normalize your data as much as possible; in the NoSQL world you are taught to denormalize as much as possible. In both cases, the truth is somewhere in the middle.

Concurrency

Let's stay with the blog example. There are multiple authors, maybe an editor, and each of them is looking at a single article at any given time. No two people work on the same article, usually. If you have data that you know is only edited by a single person at any given time, it's a good idea to place it into a single document.

Comments are different. Many people can write comments and they can do so independently and simultaneously. Once the post is published comments can be added immediately. To avoid write contention – that is, concurrent writes happening to the same document – you can store comments in separate documents, thus ensuring that again, only one author is editing a single document at any given time.

To avoid serializing and locking each comment author out or accidentally overwriting any data, just store the posts ID with the comment to be able to fetch them back in one request for displaying. (Note: Couchbase won't allow overwriting data, but it requires more complex code to handle that case, so it's best to just avoid it if possible.)

Conclusion

The relational data model relies on rigid adherence to a database schema, normalization of data and joins to store data and perform complex queries. Over the last 40 years, relational modeling and query techniques have been well established and are familiar to most application developers.

But changes in application, user and infrastructure characteristics have led application developers and architects to seek alternative "NoSQL" (non-relational) database technologies. Many view distributed document database technology as a natural successor to relational database technology:

- It effortlessly scales across commodity servers, virtual machines or cloud instances.
- It doesn't require a rigid schema before inserting data, nor does it require a schema change when different data must be captured and processed.
- Its rich data model and view technology allows for complex data modeling, capture and queries.

It's important to note that in some cases, additional storage may be required given the preference for data denormalization. But the overall benefits in performance, scalability and flexibility are usually, and increasingly, a more than fair trade.

About Couchbase

We're the company behind the Couchbase open source project, a vibrant community of developers and users of Couchbase document-oriented database technology. Our flagship product, Couchbase Server, is a packaged version of Couchbase technology that's available in Community and Enterprise Editions. We're known for our easy scalability, consistent high performance, 24x365 availability, and a flexible data model. Companies like AOL, Cisco, Concur, LinkedIn, Orbitz, Salesforce.com, Shuffle Master, Zynga and hundreds of others around the world use Couchbase Server for their interactive web and mobile applications. www.couchbase.com

© 2013 Couchbase all rights reserved | www.couchbase.com