Interactive Mediums – Migrating to CouchDB with a Focus on Views

By John Wood

Interactive Mediums

May 2010

(Original Blogposts were written July – August 2009)

Content

Couchio

The folks here at Couchio would like to thank John for documenting his migration to CouchDB so well. We hope that providing this documentation will help others that are looking to use CouchDB. John's original blogposts are at http://johnpwood.net/2009/06/15/couchdb-a-case-study/ .

This work is licensed under the Creative Commons Attribution 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Part 1: Introduction

The wall was quickly approaching. After only a few short years, several of our database tables had over a million rows, a handful had over 10 million, and a few had over 30 million. Our queries were taking longer and longer to execute, and our migrations were taking longer and longer to run. We even had to disable a few customer facing features because the database queries required to support them were too expensive to run, and were causing other issues in the application.

The nature of our business requires us to keep most, if not all, of this data around and easily accessible in order to provide the level of customer support that we strive for. But, it was becoming very clear that a single database to hold all of this information was not going to scale. Besides, it is common practice to have a separate, reporting database that frees the application database from having to handle these expensive data queries, so we knew that we'd have to segregate the data at some point.

Being a young company with limited resources, scaling up to some super-powered server, or running the leading commercial relational database was not an option. So, we started to look into other solutions. We tried offloading certain expensive queries onto the backup database. That helped a little, but the server hosting the backup database simply didn't have enough juice to keep up with the load. We also considered rolling up key statistics into summary tables to save us from calculating those stats over and over. However, we realized that this was only solving part of the problem. The tables would still be huge, and summary tables would only replace some of the expensive queries.

It was about this time that my colleague Dave started looking into CouchDB as a possible solution to our issues. Up until this point, I had never heard of CouchDB. CouchDB is document oriented, schema-free database similar to Amazon's SimpleDB and Google's BigTable. It stores data as JSON documents and provides a powerful view engine that lets you write Javascript code to select documents from the database, and perform calculations. A RESTful HTTP/JSON API is used to access the database. The database boasts other features as well, such as robust replication, and bi-directional conflict detection and resolution.

The view engine is what peeked our interest. Views can be "rebuilt" whenever we determine it is necessary, and can be configured to return stale data. "Stale data? Why would I want stale data?", you may be asking yourself. Well, one big reason comes to mind. Returning stale data is fast. When configured to return stale data, the database doesn't have to calculate anything on the fly. It simply returns what it calculated the last time the view was built, making the query as fast as the HTTP request required to get the data. The CouchDB view engine is also very powerful. CouchDB views use a map/reduce approach to selecting documents from the database (map), and performing aggregate calculations on that data (reduce). The reduce function is optional. CouchDB supports Javascript as the default language for the map and reduce functions. However, this is extensible, and there is support out there for writing views in several other languages.

In our case, we are planning to use CouchDB as an archive database that we can move "old" data to once a night. Once the data is moved to the CouchDB database, it would no longer be updated, and would only be used for calculating statistics in the application. Since we would only be moving data into the database once a day, we only need to rebuild the views once a day. Therefore, all queries

2010 Couchio — www.couch.io

could simply ask for (and get) stale data, even when the views were in the process of being rebuilt. Also, moving all of the old data out of the relational database would dramatically reduce the size of the specific tables, improving the performance of the queries that hit those tables.

I'm really looking forward to this partial migration to CouchDB. The ability to add new views to the database without affecting existing views gives us the flexibility we need to grow our application to provide better, more specific, and more relevant statistics. In marketing, statistics are king. Since our Application is a mobile marketing tool, we want it to be able to provide all of the data that our customers are looking for, and more. I feel that by moving to CouchDB, we will not only be able to re-activate those features that we had to disable due to database performance, but also add more features and gather more statistics that would have otherwise been impossible with our previous infrastructure.

We faced several challenges migrating to CouchDB and learned some important lessons along the way. All of those challenges will be addressed here.

2010 Couchio — www.couch.io

Part 2: Databases and Documents

CouchDB is a document oriented database. A document oriented database stores information as documents of related data. All of the data within a document is self contained, and does not rely on data in other documents within the database. This can be quite a shift if you're used to working with a relational database, where data is broken up in to multiple rows existing in multiple tables, limiting (or eliminating) the duplication of data. Although radically different, the document oriented approach is a very good fit for many applications. For some applications, data integrity is not the primary concern. Such applications can work just fine without the restrictions provided by a relational database, which were designed to preserve data integrity. Instead, giving up these restrictions lets document oriented databases provide functionality that is difficult, if not impossible to provide with a relational database. For example, it is trivial to setup a cluster of document oriented databases, making it easier to deal with certain scalability and fault tolerance issues. Such clusters can theoretically provide you with limitless disk space and processing power. This is the primary reason why document oriented databases (or key/value pair databases) are becoming the standard for data storage in the cloud.

There are plenty of articles on the web describing the benefits of using a document oriented or key/value pair database, so I won't re-hash any of that information here.

Databases

Creating a new database in CouchDB is a simple process, with no overhead. In fact, it's as simple as issuing a single HTTP request.

```
curl -X PUT http://127.0.0.1:5984/my_database
```

There appears to be no "penalty" for hosting many databases within the same CouchDB server, as opposed to storing all of the documents within a single database. We took advantage of this when migrating data into CouchDB. Three very large tables were the focus of this migration, each containing between 3 to 50 million rows. We decided to store the data from each table in its own database. The data within these tables are completely unrelated, so we would never need to view data from one database combined with another. If they were related, we would have combined the tables into a single database, as CouchDB cannot create views across multiple databases. Also, storing each set of data in its own database provides additional flexibility. During the migration process, there were several points where we changed the structure of the documents. Having the ability to easily delete the affected database and re-populate it, without affecting any other document types, came in handy. With multiple databases, we have the flexibility to change the replication schedule for one database to be different from the others. It also makes it easy to move one or more databases to another server, should we ever choose to do so.

My colleague Dave made a few changes to CouchRest Rails to support connecting to multiple CouchDB databases within a single rails application. You simply specify the database server location in the configuration files, and then each model object can specify which database it is using.

Documents

CouchDB documents are very flexible. Documents are stored in JSON format, allowing you to take advantage of JSON arrays and dictionaries to represent collections of data. There is no external force dictating how a document should be structured, or what it should contain (as long as the document is valid JSON). Below is an example of what a document may look like for a blog post.

```
01    {
02       “_id”: “CouchDB: Databases and Documents”,
03       “_rev”: “1-704787893”,
04       “author”: “John Wood”,
05       “email”: “john_p_wood”,
06       “post”: “CouchDB is a documented oriented database.  A   document...”,
07       “tags”: [“couchdb”, “couchdb case study”, “json”],
08       “comments”: [
09          {
10             “email”: “joe@somewhere.com”,
11             “comment”: “Thanks for the information”
12          },
13          {
14             “email”: “kevin@xyz.com”,
15             “comment”: “CouchDB sounds pretty interesting”
16          }
17       ]
18    }
```

Schema-less

Probably the best part about the document oriented approach is the ability to make each document different from the next. There is no schema to enforce that a document contains specific information. This makes CouchDB a great fit if your application needs to store data that can be wildly different between objects of the same type. In a relational database, this is usually handled by serializing the data in some format, writing the serialized data to the database, and de-serializing the data when it is read by the application. However, this is really nothing more than a hack. Querying the data in such columns can be a nightmare. And, the serialization/de-serialization process is just one more thing that can go wrong. In a document oriented database, there is no need for such a hack. You simply code your CouchDB views to account for the fact that certain fields may not be in the document, and act accordingly (either defaulting to some value, or simply move onto the next document in the database).

Self contained

The most important thing to remember about documents is that they are self contained. All of the data representing a particular concept is right there in the document. (This is a bit of a fabrication, as it is completely possible to establish relationships between documents by having one document store the unique id of another document. However, these links are not directly supported by CouchDB, and can

be easily broken.) So if you are moving from a relational database to CouchDB, you should de-normalize your data as much as possible while defining the structure of your document. JSON arrays and dictionaries can help tremendously when de-normalizing relationships. If there is only one piece of information from the relationship worth storing in the document, then an array works great (see the "tags" property above). For relationships with more complex data structures, an array of dictionaries fits the bill quite nicely (see the "comments" property above).

The document id

Another important point to consider when designing your document structure is defining what you will use as the id of the document. The id must be unique not only in that database, but all instances of the same database if you happen to be running inside a database cluster. CouchDB uses document ids to replicate changes between servers. Auto-generated sequential keys are a poor fit for this. While wildly popular in relational databases, auto-generated sequential keys throw a wrench into the gears of the replication process. If each database in the cluster was responsible for generating its own sequential ids, it is highly likely that different documents on different servers could be assigned the same id, which would make CouchDB think that two distinct documents are the same document. Badness would surely ensue.

Instead, it is recommended that you use the data's "natural key" as the id of your document. The natural key is some field, or combination of fields, in your document that uniquely identifies that document. In the example above, the title of the blog post is a good fit for a natural key. It is not very likely that I will be writing posts with the same title. If you happen to enjoy writing about the same stuff over and over, perhaps the title of the post combined with the date and time it was created would be a better fit. Either way, the id should be composed from data within the document.

If you do not provide an id, CouchDB will provide one for you. CouchDB uses an algorithm that makes it virtually impossible for multiple CouchDB instances to generate the same id. However, I have read articles on the web indicating that this is a very slow operation, so you may want to avoid letting CouchDB generate an id for you. Regardless, natural keys pulled straight from data within your document always make better ids, as they are easier to read, and more identifiable.

For a few of our documents, we used the sequential key generated by MySQL as the id :) I know how stupid this sounds, given the last few paragraphs. However, I think this was the best choice for an id in our case. The data contained in these documents are basically a collection of ids to rows that exist, and will remain in MySQL. None of the data within the document would be any more readable than the MySQL id. Also, since all of the keys were originally generated in a single MySQL database, they are guaranteed to be unique. As of right now, we always plan on creating the data in MySQL, and then "archiving" it to CouchDB at a later date, so this approach should continue to work just fine.

Supporting existing functionality

If you are migrating data from a relational database to CouchDB, there is another important item to think about. If your application needs to interact with CouchDB in the same way that it did with the relational database, then you need to make sure that the CouchDB views you build will be able to

replace any SQL queries that are done against that data in the relational database. In order to make this happen, the CouchDB document will need to contain all of the necessary information for you to build views to replace those queries, if you intend on supporting the same functionality. Remember, there are no JOINs in CouchDB.

Summary

I think the way CouchDB handles databases and documents is very straight forward. Once you get used to the idea that there could be multiple instances of databases in a cluster, and that documents should be self contained, the rest is cake. The schema-less approach has the potential to open a lot of doors. I know that we're already making plans to take advantage of it.

2010 Couchio — www.couch.io

Part 3: Views – The Advantages

Views are what attracted us to CouchDB. If you've made it this far you already know that CouchDB is a document oriented database, and that documents themselves don't have any official structure beyond the structure enforced by JSON. Views are what provide the necessary structure so that you can run queries on your data.

CouchDB has several strong points, including its efficient B-Tree data store implementation and replication/synchronization support. These strong points already set it apart from other, more traditional databases. However, we came for the views, because we saw views as the potential answer to our database performance woes.

CouchDB builds views using a map/reduce algorithm. When building a view, CouchDB will feed all documents that are new or have changed since the last time the view was built through a map function. The map function selects the documents of interest for that particular view. Then, optionally, a reduce function is run to calculate some aggregate statistics on the documents that have been selected (counts, sums, etc). There are several places you can go on the web for more information about how CouchDB views work.

A large part of the performance issues we are trying to address are being caused by repeatedly running the same database queries against very large tables, where the vast majority of the data in those tables has not changed since it was inserted. The last part of that statement is very important. **The data has not changed since it was inserted**, and due to the nature of these tables, it probably never will. It was very wasteful for us to keep running the same calculations on that old data.

This is where CouchDB views come in. When CouchDB generates a view, it stores the result of the view on disk in a B-Tree data structure, which is very efficient to access. CouchDB will only re-generate that view when documents that match the criteria specified in the map function are changed or added. And, CouchDB will only need to update the view for the changed/added documents. It will incrementally update the view's index, so it doesn't have to start from scratch every time. This makes views especially ideal for large data sets.

Using views, we can replace all of the queries we were performing on these tables, and the calculations would be performed once, and then stored. Accessing that data would be as simple as issuing a single HTTP request, which would efficiently pull the data from the view's B-Tree. In other words, it would be fast, and very efficient.

CouchDB views are also very flexible. The output of the map function is a key/value pair. That key/value pair can be anything…data from the document, hard coded values, whatever. This flexibility allows you to create complex keys, such as a JSON array of values. Using the view API, you can specify ranges of keys when executing your query, fetching only the data that you want. You also have the ability to group complex keys by the first *n* elements of the key, and run the reduce function on those groups of data. This enables you fetch aggregate data on multiple levels, and allows you to support multiple queries with a single view. For example, we need to calculate the number of SMS messages sent by a particular account by minute, hour, day, month, year, etc. Using CouchDB's view engine, we can have our map function emit a key of `[account_id, year, month, day,`

2010 Couchio — www.couch.io

`hour, minute]` and a value of 1 for each document in our messages database. Our reduce function simply sums all of the values for a matching key, using the provided sum function. Here is how simple the map/reduce code is for this view:

Map

```
1    function(doc) {
2        datetime = doc.created_at_utc;
3        year = parseInt(datetime.substr(0, 4));
4        month = parseInt(datetime.substr(5, 2), 10);
5        day = parseInt(datetime.substr(8, 2), 10);
6        hour = parseInt(datetime.substr(11, 2), 10);
7        minute = parseInt(datetime.substr(14, 2), 10);
8        emit([doc.account_id, year, month, day, hour, minute], 1);
9    }
```

Reduce

```
1    function(keys, values, rereduce) {
2        return sum(values);
3    }
```

Using the grouping feature of the view API, we can easily fetch message counts for this account by year, month, day, hour, or minute, by simply specifying how many levels of the key we would like to group together. For example, to get a breakdown of messages sent for a particular account on each day in May of 2009, we would simply include the following parameters in our URL when accessing the view: `startkey=[1,2009,5]&endkey=[1,2009,5,{}]&group_level=4`. These parameters tell the view that we only want to consider messages for account number 1 that were sent or received in May of 2009, and that we'd like the reduce results grouped by the 4th parameter in the key, which is the day of the month. This would return something like:


```
{"rows":[
{"key":[1,2009,5,1],"value":13},
{"key":[1,2009,5,2],"value":9},
{"key":[1,2009,5,3],"value":10},
{"key":[1,2009,5,4],"value":9},
{"key":[1,2009,5,5],"value":11},
{"key":[1,2009,5,6],"value":17},
{"key":[1,2009,5,7],"value":12},
{"key":[1,2009,5,8],"value":12},
{"key":[1,2009,5,9],"value":14},
{"key":[1,2009,5,10],"value":8},
{"key":[1,2009,5,11],"value":12},
{"key":[1,2009,5,12],"value":11},
{"key":[1,2009,5,13],"value":9},
{"key":[1,2009,5,14],"value":20},
{"key":[1,2009,5,15],"value":7},
{"key":[1,2009,5,16],"value":15},
```


2010 Couchio — www.couch.io

```
{"key":[1,2009,5,17],"value":8},
{"key":[1,2009,5,18],"value":8},
{"key":[1,2009,5,19],"value":13},
{"key":[1,2009,5,20],"value":7},
{"key":[1,2009,5,21],"value":12},
{"key":[1,2009,5,22],"value":28},
{"key":[1,2009,5,23],"value":8},
{"key":[1,2009,5,24],"value":4},
{"key":[1,2009,5,25],"value":2},
{"key":[1,2009,5,26],"value":16},
{"key":[1,2009,5,27],"value":15},
{"key":[1,2009,5,28],"value":12},
{"key":[1,2009,5,29],"value":7},
{"key":[1,2009,5,30],"value":5},
{"key":[1,2009,5,31],"value":6}
]}
```

Views are re-built when they are accessed, and not when new documents are added to the database or existing documents are changed. However, you **do** have control over when views are built. If you specify `stale="ok"` when accessing your view, CouchDB will not check to see if the view needs to be re-built. It will simply return results from the last time the view was built. We took advantage of this feature when writing the application code to access the views. In our case, data is only added to the database once a day, and it is added by a background job. When the job is finished inserting data into the CouchDB database, it triggers the views to re-build themselves by accessing all of the views in the database (a few at a time), without specifying the `stale="ok"` flag. Since this background job takes on the responsibility of updating the views after it inserts new data, the rest of our application can always specify `stale="ok"` when accessing the views. This keeps the queries executed by the application fast, even when views are in the process of being re-built.

Views are powerful, and offer a tremendous amount of flexibility. However, they come with their own set of challenges. Next, I will talk about some of the challenges we faced when attempting to replace our SQL queries against a MySQL database with CouchDB views.

2010 Couchio — www.couch.io

Part 4: Views – The Challenges

Earlier I wrote about the many features of CouchDB views. Now, I will describe the challenges we faced when replacing MySQL queries with CouchDB views.

Map/Reduce

One of the largest challenges with CouchDB views is simply wrapping your brain around the map/reduce model. If you've spent any significant amount of time in the relational model, this can be quite a task. Do not underestimate it. Give yourself plenty of time to make this adjustment. It really is a whole new world. After several weeks, I'm still not 100% sure how to use the map/reduce model to its fullest potential. In fact, there were a few queries that I could not figure out how to implement as views. Because of that, I had to keep around an archive table in MySQL, and I run those few queries against that table.

Javascript

If you don't know Javascript, you may want to tack on another week to the learning curve. Javascript is an incredibly powerful language. However, in its raw form it is fairly basic and can take some getting used to. It does help that you don't need to write too much Javacript to implement most map and reduce functions. However, if you're new to Javascript, get ready to do some research.

There are view servers available in other programming languages, and it is easy to configure CouchDB to use them. But, CouchDB is still young and under development, and these alternative view servers are not supported by the CouchDB team. So, use them at your own risk.

SQL

As I mentioned earlier, views are powerful and flexible. But, views are not nearly as flexible as SQL. SQL has been in development for **decades**. Even today, it continues to evolve as a language. You can do a ton with SQL. As of right now, views simply cannot rival this flexibility. The CouchDB team continues to add built-in Javascript functions to help write map/reduce code, and there is even talk about supporting map/reduce/merge. But as of right now, the feature sets are not even close. It is very difficult for any new technology to enter the game with the same, or even a comparable feature set to such a battle-hardened veteran. And to be honest, I highly doubt that the CouchDB team is even trying to match SQLs feature set. After all, CouchDB is not meant to be a replacement for the relational database. However, this is an important point to consider if you think your current relational database backed application might be a good fit for CouchDB.

Multiple views, one design document

Views live in documents called "design documents". Views within the same design document share a B-Tree data structure. This means that when one view in the design document is built, they all are built. So, careful planning is required to make sure unrelated views do not live in the same design

2010 Couchio — www.couch.io

document. You would not want the re-building of one view to delay the accessibility of another, totally unrelated view.

Building/Indexing views

Views can take a **L…O…N…G** time to build from scratch. The view building process is also very resource intensive. This becomes less of an issue once the views have been built, as views are updated incrementally. It really only comes into play when you are adding many, many documents to a CouchDB database between view builds. However, one place where this is an issue is ad-hoc queries. Every week or two, I'll get a request from a customer for data that is not available via our web application. While we'll throw that request onto the product backlog so it is eventually available via the application, it doesn't change the fact that our customer needs that information now. We usually satisfy such requests by firing up the MySQL client, and running one or more ad-hoc queries. This simply is not feasible using CouchDB views, especially if you are working with a large database containing millions of documents. CouchDB does support "temporary views", which are ad-hoc views that you can build and execute on the fly. However, temporary views are not recommended for production use, because they need to be built before they can get you the data you need. This could take hours, or days depending on the size of you database and the processing power of your database server. Temporary views are meant as a way to test new views in development which will eventually be saved into a design document, and not for running ad-hoc queries.

View sizes on disk

I've already mentioned that each design document is stored in its own B-Tree, completely separate from the B-Tree that holds the documents in the database. These data structures can become quite large, especially if you have a ton of documents in your database. A large database combined with several design documents can take up quite a bit of disk space. For example, our main messaging database consisting of around 30 million documents is 20GB on disk. The 8 design documents for that database combine for a total size of 35GB. This brings the grand total, for the documents and the views, to 55GB. That's a whole lot of disk space. CouchDB sacrifices disk space for performance, which **is** a good tradeoff, as disk space is cheap and virtually limitless these days. But sadly, this is not the case for everyone. To add enough storage capacity to handle this database, the other large databases, a mirror/backup of these databases, and still have room to grow, we were looking at an additional several hundred bucks a month in charges from our hosting company for the additional disk space. That can be a lot to handle for a small company. Some larger companies use SANs or similar storage devices that offer unmatched redundancy and reliability. However, these devices will often have a fixed maximum amount of storage, and can be very expensive to upgrade one the storage capacity is maxed out. Justifying the use of so much disk space on such an expensive resource could prove difficult.

2010 Couchio — www.couch.io

Summary

Views were without a doubt the hardest part of this project to get right. My colleagues Dave and Jerry are still hard at work trying to find creative ways to reduce the size of the views on disk. We're very happy with the performance boots that we've seen using views in our testing environment. But unless we can find a way to reduce the disk usage, or find more affordable storage options, we may never see these performance boosts in production. (See Part 6: The Last Mile for the solution.)

2010 Couchio — www.couch.io

Part 5: Application Changes

Compared to the challenges we faced with views, modifying our Application to interact with CouchDB was very straight forward. This section describes how we changed the Application code in order to use CouchDB as an archive database. Since Our Application is a Ruby on Rails application, much of the content in this post references Ruby/Rails specific libraries and frameworks. However, I feel the general concepts could be applied to any development platform.

Configure the application to access CouchDB

Before our application can talk to CouchDB, we need to tell it a little bit about our CouchDB installation. The CouchRest-Rails plugin aims to make this as easy as possible for Rails applications. CouchRest-Rails provides the necessary hooks that allow you to specify your CouchDB configuration in a `couchdb.yml` file, which serves the same purpose as the default `database.yml` file used by Rails. Simply update this file with your CouchDB connection information, and you'll be able to easily connect to CouchDB from within your application.

CouchRest-Rails also provides a series of Rake tasks that help you manage your databases and views.

Define the documents

The very first thing you need to do when moving data to CouchDB is to figure out what your documents will look like. I talked about this earlier in Part 2, so I won't cover it again here.

Write code to create documents from relational database backed data objects

Once you know what the documents are going to look like, you need to write some code that will convert your RDBMS backed objects into a document, and store it in CouchDB.

We decided to use CouchRest to help us interact with CouchDB. CouchRest consists of two main parts: code to interact directly with CouchDB via a set of APIs just above CouchDB's HTTP API (known as CouchRest Core), and code that allows you to create an object model backed by CouchDB. The `ExtendedDocument` class is the cornerstone of the object model code. `ExtendedDocument` is like `ActiveRecord::Base` in Rails. It serves as the base class for CouchDB backed objects. It provides convenient ways to define document properties, access views, define life cycle callbacks, create documents, save documents, destroy documents, paginate view results, and more.

A class extending `ExtendedDocument` simply needs to define the properties that make up its document.

```
1    class ArchivedContestCampaignEntry < ExtendedDocument
2      use_database :contest_campaign_entry_archive
3
4      property :campaign_id
5      property :user_id
6      property :entry_number
7      property :winner
8    end
```

Then, all it takes to save a document in CouchDB is to create an instance of this class, set its properties, and call the object's `create` method.

Determine how data will be moved to CouchDB

Now that we have code that can convert RDBMS objects into documents, we need to figure out how to actually get those documents into CouchDB. This step will likely be dependent on how you plan on using CouchDB. For us, we decided it would be best to "archive" records after their corresponding campaigns have been over for 48 hours or more. So, we created a nightly cron job to fetch all non-archived campaigns that have been over for 48 hours or more, and move their corresponding entries to CouchDB. When a campaign's entries have been moved, an "archive" flag is set on the campaign itself, so the application knows to fetch its entries from CouchDB instead of MySQL.

One important item to point out is that CouchDB supports a bulk save operation. This operation allows you to save a batch of documents with a single HTTP request. This is a big time saver, as executing one HTTP request is obviously much quicker than executing several thousand. Our archive cron job takes advantage of this. When archiving entries for a particular campaign, we will build one document for each entry record, and then toss that document into an array. When that array exceeds a certain size, 2,500 in our case, a single bulk save request is sent to CouchDB with the array of documents. This dramatically decreases the number of HTTP requests sent to CouchDB, and the amount of time required to add data to CouchDB.

In addition, our archive job will pause to rebuild all of the views in the database after 100,000 new documents have been inserted, as well as at the end of the job. CouchDB will handle smaller, incremental view updates more efficiently than very large view updates. The final view rebuild is necessary since all of the view queries done from within the application ask for stale data, which will not trigger a view update. (Update: Version 0.11 has greatly improved this functionality and can now handle large view updates much more efficiently.)

Replacing SQL queries with CouchDB views

Next, we changed the application to support the substitution of SQL queries with CouchDB views. We did this in several steps.

Identify queries being performed on the data you want to move

The first step in replacing SQL queries with CouchDB views is identifying all of the queries being performed on the data you plan on moving to CouchDB. This took a few hours to do, but was not difficult by any means. We simply searched the code for all instances of the `ActiveRecord` class name and the MySQL table name for the tables with data being moved. We also tracked down all `ActiveRecord` associations that were made to that particular table. We then made a note of what the queries did, and how they were used.

Abstract away the query

After the queries had been identified, we moved the execution of all queries to a new class. This freed the rest of the application from having to know if the data being fetched lived in MySQL or CouchDB. The new class would make that decision, delegating to the `ActiveRecord` class if the data was in MySQL, or the `ExtendedDocumenet` class if the data was in CouchDB. To start off, we simply delegate to the `ActiveRecord` class since we have not yet implemented the CouchDB views.

```
1     class ContestCampaignEntryDelegate
2       def self.find_all_by_campaign_id_and_winner(campaign_id, winner)
3         # Delegate to the ActiveRecord object
4         ContestCampaignEntry.find_all_by_campaign_id_and_winner
      (campaign_id, winner)
5       end
6     end
```

Build views to replace the queries

Now that we have the complete list of queries performed on the data that we wish to archive, we can begin building the necessary CouchDB views to support those queries for archived campaigns.

Add methods to the `ExtendedDocument` class to query the views

CouchRest gives you a few options when it comes to creating and accessing your views.

One option is to use the `view_by` method available on all classes that extend ExtendedDocument. `view_by` not only makes the views easily accessible via the code, but it will also take care of creating and storing the view in the database.

In its simplest form, `view_by` will generate the necessary map function based on the parameters you specify. This example from the CouchRest documentation shows the map function that will be generated when `view_by :date` is called inside a class named `Post`:

2010 Couchio — www.couch.io

```
function(doc) {
  if (doc['couchrest-type'] == 'Post' && doc.date) {
    emit(doc.date, null);
  }
}
```

`view_by` also lets you specify compound keys (`view_by :user_id, :date`) and any parameters that you wish to be used when you query your view (`:descending => true`).

If you need to do something a little more complicated, `view_by` will also let you specify the map and reduce functions to use. Here's another example from the CouchRest documentation:

```
# view with custom map/reduce functions
 # query with Post.by_tags :reduce => true
view_by :tags,
  :map =>
    "function(doc) {
      if (doc['couchrest-type'] == 'Post' && doc.tags) {
        doc.tags.forEach(function(tag){
          emit(doc.tag, 1);
        });
      }
    }",
  :reduce =>
    "function(keys, values, rereduce) {
      return sum(values);
    }"
```

Another option for creating and accessing views is to use CouchRest Core. CouchRest Core, as described above, is a raw, close to the metal set of APIs that let you interact with CouchDB. These APIs let you do basically anything, including creating and accessing views. This example from the CouchRest documentation shows how you can create and query a view using CouchRest Core:

```
@db = CouchRest.database!("http://127.0.0.1:5984/couchrest-test")
@db.save_doc({
  "_id" => "_design/first",
  :views => {
    :test => {
      :map =>
        "function(doc) {
          for (var w in doc) {
            if (!w.match(/^_/)) emit(w,doc[w])
          }
        }"
      }
    }
```


2010 Couchio — www.couch.io

```
14      })
15    puts @db.view('first/test')['rows'].inspect
```

For accessing our views, we decided to go with a hybrid approach. We didn't really feel comfortable storing our map and reduce functions inside the application code. Doing so made it less clear on how we could gracefully introduce new views or update existing views in production, keeping in mind that some of these views could take hours or days to be built for the first time. Instead, we stored our map and reduce code outside of the application, and used CouchRest-Rails to help us get those views into the database. This allows us to push new or updated views independent of the application, giving us time to build the views before anything tries to access them.

Since the views are already in the database, we decided to use CouchRest Core to access them. We created a class called `ArchivedRecord` to make working with CouchRest Core a little easier. `ArchivedRecord` contains methods that do type conversions, manage bulk save operations, incrementally regenerate the views, and more. It also contains a series of methods that help with executing views with similar behavior. For example, there are methods that will simply return the number of rows returned by a view, execute a view for a specific timeframe using the dates stored in the documents, etc. These abstractions also handle any errors that could pop up when accessing a view. Our application code uses the abstractions provided by `ArchivedRecord` to access the views.

Change the delegate class to call the `ExtendedDocument` class for archived data

Now that our views can be accessed via the application code, we can modify the delegate class to call the `ExtendedDocument` object's query method to fetch data for campaigns that have been archived.

```
01    class ContestCampaignEntryDelegate
02      def self.find_all_by_campaign_id_and_winner(campaign_id, winner)
03        campaign = ContestCampaign.find_by_id(campaign_id)
04        if campaign.archived?
05          ArchivedContestCampaignEntry.find_all_by_campaign_id_ and_winner
      (campaign_id, winner)
06        else
07          ContestCampaignEntry.find_all_by_campaign_id_and_winner
      (campaign_id, winner)
08        end
09      end
10    end
```

Deal with the `ActiveRecord` associations

The last piece of the puzzle is to deal with the `ActiveRecord` associations. `ActiveRecord` associations are magic little things that make a record's associated data accessible via methods on an instance of the `ActiveRecord` class. For example, if I wanted to declare an association between a contest and its entries, I would simply declare the following at the top of my `ContestCampaign` class:

2010 Couchio — www.couch.io

```
1  has_many :contest_campaign_entries
```

`ActiveRecord` takes care of joining the `contest_campaign_entries` table with the `contest_campaigns` table, and making all of the related campaign entries available via a call to `some_contest_instance.contest_campaign_entries`.

This will not work for us, as the `contest_campaign_entries` table will not contain any data for archived contests. So, we need to handle associations differently.

Instead of using the above code to create the association, we use the following:

```
1     has_many :active_contest_campaign_entries,
2       :foreign_key => 'contest_campaign_id',
3       :class_name => 'ContestCampaignEntry'
```

This more verbose version tells `ActiveRecord` that we want to setup an association, named `active_contest_campaign_entries`, on the class. Since we're circumventing the convention of naming the association after the foreign key to the associated data (which is in turned named after the associated table), we need to specify the foreign key to use, and the name of the class that backs that table.

To keep from breaking the existing code that uses the `contest_campaign_entries` method to obtain related entry data for a contest, we define a new method on the class with that name to fetch the associated data. The new method simply calls the corresponding method on the delegate class, which will pull the associated entries from MySQL or CouchDB, depending on if the campaign has been archived.

```
01    class ContestCampaign
02    …
03      def contest_campaign_entries
04        ContestCampaignEntryDelegate.contest_campaign_entries(self.id)
05      end
06    …
07    end
08
09    class ContestCampaignEntryDelegate
10    …
11      def self.contest_campaign_entries(campaign_id)
12        campaign = ContestCampaign.find(campaign_id)
13        if campaign.archived?
14          ArchivedContestCampaignEntry.find_all_entries(campaign_id)
15        else
16          campaign.active_contest_campaign_entries
17        end
18      end
19    …
20    end
```


2010 Couchio — www.couch.io

`ActiveRecord` supports other associations besides `has_many`. These other associations also add methods to the class that will fetch associated data from the database. In our case, some of this associated data is going to remain in MySQL. CouchRest will not (and should not) automatically fetch the corresponding data from MySQL, so we needed to handle this ourselves.

In our documents, we store the ids of the associated data still in MySQL (see `campaign_id` and `user_id` in the document snippet below). Because we have associations setup between the `ContestCampaignEntry` class and the `ContestCampaign` and `User` classes, `ActiveRecord` adds methods named `campaign` and `user` to `ContestCampaignEntry` that will fetch the associated objects. We need to do the same in our `ExtendedDocument` class.

```
01   class ArchivedContestCampaignEntry < ExtendedDocument
02     use_database :contest_campaign_entry_archive
03
04     property :campaign_id
05     property :user_id
06     property :entry_number
07     property :winner
08
09     def user
10       @user ||= User.find_by_id(user_id)
11     end
12
13     def campaign
14       @campaign ||= ContestCampaign.find_by_id(campaign_id)
15     end
16   end
```

The `user` and `campaign` methods in the class above will take the ids stored in the document and fetch their corresponding objects from MySQL. In our case, these values will never change for an archived entry, so we hold on to the objects as instance variables to avoid doing additional queries when they are referenced again.

Make the `ExtendedDocument` **class** *act like the* `ActiveRecord` **class**

As I stated above, one of the goals was to make it so the application code does not need to know which database the data is coming from. Since the data can be returned as instances of two different classes, `ContestCampaignEntry` or `ArchivedContestCampaignEntry`, we need to make sure that both of these classes implement the same methods, and behave the same way. Failing to do so could result in hard to find bugs, or straight up exceptions.

One group of methods to pay extra attention to are the convenience methods that `ActiveRecord` adds to the class based on the attribute types in the database. An example of this is the `attribute?` accessor method that is added for boolean attributes. All attributes get an accessor named after the

2010 Couchio — www.couch.io

column in the database, but boolean attributes get an additional accessor, containing a "?" at the end. I personally use the "?" variation of the accessor method all of the time, as I feel it makes the code easier to read and understand.

CouchRest on the other hand is not able to determine in advance the data types of the properties you have stored, since CouchDB is a schema-less database. So, it is not able to do anything special for properties of a given type unless you specifically tell it what the type is. CouchRest does allow you to specify a type when you declare the property, but the current release (version 0.32) only uses this to cast property values into their proper type after they are fetched from the database. I've submitted a patch that will generate "?" accessor methods for properties with a type specified as `:boolean`. However, this is just one example of how your `ExtendedDocument` class could be subtlety different from the corresponding `ActiveRecord` class.

Summary

As I stated at the beginning of this section, changing the application to work with CouchDB was much more straightforward than getting the views to work as expected. Perhaps this is because I'm a developer, and not a DBA. But, great libraries like CouchRest and CouchRest-Rails certainly go a long way in helping to write clear and concise code that interacts with CouchDB. I can only hope that other programming languages have, or soon will have, libraries like these. The fact that CouchDB has a great API built on a protocol that everybody can talk, HTTP, certainly makes it possible.

2010 Couchio — www.couch.io

Part 6: The Last Mile

Addressing the remaining issues

We were almost there. After modifying the code to talk to CouchDB, our application was successfully pulling data from CouchDB in our development environments. There were just a few remaining issues that needed to be addressed before we could deploy CouchDB to production.

Reducing the view sizes on disk

As I mentioned previously, the amount of disk space consumed by the views was a big problem. If we didn't do something, we were sure to run out of disk space when migrating our 30 million row messages table to CouchDB.

We determined that it was not **what** we were emitting from our map functions that was killing us, but **how many times** we were emitting it. Each of the views emitted a key/value pair for every document in the database. At 30 million documents and 8 views, that ends up being a crap load of key/value pairs.

My colleagues Dave and Jerry took a detailed look at the problem, and came up with a solution. They determined that there was simply no need to be emitting data for each document in the database. While this would give us views that could report statistics by the second, our application only supported presenting statistics by the minute. Even if we were able to support statistics at this level of detail, we doubted our customers would even need it. It was simply not worth the disk space.

So, Dave and Jerry modified the import job described earlier to roll up several key statistics by the minute as it was building the documents. When the job finishes processing all of the documents for that minute, it creates a summary document containing all of the rolled up statistics, and adds it to the database. Then, they changed the map functions to only consider these summary documents.

This solution was able to **dramatically** reduce the sizes of the views on disk, while still supporting the current application functionality. Since we are still persisting all of the original documents to CouchDB, it is possible to add a new statistic to the summary documents should we ever need to.

Oh, and we also picked up two new terabyte database servers, just in case :)

Paginating records in CouchDB

Like many Rails applications, we were using the popular will_paginate gem to paginate results from the database. Given the size of our data sets, pagination was an absolute necessity to keep from using up every last bit of memory.

CouchRest has a `Pager` class that paginates over view results, but it is in the CouchRest Core part of the library and doesn't integrate too well with the object model part of the library. It simply returns the view results as an array of hashes. We were hoping to see a solution that would give us back an array of the corresponding `ExtendedDocument` objects. We were also trying to keep our

2010 Couchio — www.couch.io

application from having to know about CouchDB outside of the classes described earlier in Part 5: Application Changes. Having completely different pagination strategies for the two databases would make that more difficult.

So, I decided to write some new pagination code that supported the will_paginate interface and integrated a little better with the object model part of CouchRest. I had a quick solution that same day which fetched view results and handed back an array of the corresponding `ExtendedDocument` objects. I then spent some time over the next two weeks modifying the code to integrate a little better with CouchRest and add support for CouchRest views, which we weren't using.

With the new code in place, we can now paginate over a set of contest entries without having to know what database they are coming from.

```
1     ContestCampaignEntryDelegate.contest_campaign_entries.paginate(
2       :page => 1, :per_page => 50)
```

This pagination code eventually made it into CouchRest.

Going live

With the remaining issues addressed, it was time to start the production migration. One at a time, we manually started the jobs to move the data from MySQL to CouchDB. When one job completed, we would start the next. As I mentioned earlier in Part 4: Views - The Challenges, building the views is very resource intensive. We didn't want to completely bog down the production machine we were using to do the migration by running multiple jobs at once.

Moving the archived data from MySQL to CouchDB and building all of the views took about a week (a day for this table, a couple of days for that table, etc). Overall, it was a fairly smooth process.

For the initial import, we did not purge any of the data from MySQL. Since we needed to wait until our CouchDB databases were fully populated with all views built before we could start using them, the application needed to continue working with the data in MySQL while the migration was in progress. In anticipation of the eventual switch from MySQL to CouchDB, I added a flag in the application configuration that told the application if it should pull archived data from CouchDB. Once all of the data had been imported and all of the views had been built, we flipped the switch.

With the pouring of a celebratory beer, we watched as our application began pulling data from CouchDB in production. It was time to relax :)

The results

I **really** wish we had taken the time to record how long our "troublesome" pages were taking to load before the move to CouchDB. Sadly, we did not. All I can say is that **pages that used to occasionally time out were now loading in a few seconds**. Since the migration, we have also implemented a few new features that would simply not have been possible without CouchDB due to database performance issues.

2010 Couchio — www.couch.io

The database performance issues we set out to address seem to be a thing of the past. If new ones pop up, I'm confident that we could once again utilize CouchDB to address them.

What's next

This project was focused on addressing database related performance issues that we were facing in production. With these issues out of the way, and our CouchDB infrastructure built-out and proven, we will soon be building even more reporting capabilities that would have simply killed our old database. Interactive Mediums customers will soon be able to view their data in more ways than they could have imagined.

I am also working on a project that takes advantage of CouchDB's schema-less nature to let our customers store and utilize data they collect from their customers. Such a feature, which essentially lets customers define their own schema, would have been a challenge to implement in a relational database. With CouchDB, its just a document.

Thoughts about this project, and CouchDB

I learned a **ton** while working on this project. While vaguely familiar with "NoSQL" databases before this project, I have just recently become aware of all of the alternatives available. With the enormous amount of data companies are beginning to collect and process, I'm sure that CouchDB and its NoSQL friends will soon become a common component in the operational environments of most companies.

The CouchDB community has been great. The CouchDB and CouchRest mailing lists are extremely active, and have been very helpful. The committers on both of these projects are active, and always eager to help. I'd specifically like to call out Jan Lehnardt and Chris Anderson from the CouchDB project/Couchio. Jan has commented on a few of these posts, encouraging me to keep writing. He also suggested a more efficient implementation of the CouchRest pagination code I wrote, which I quickly implemented. Chris left a comment on the first post in this series thanking me for writing about CouchDB, and offering his assistance if I needed it. I actually took Chris up on that offer when we were running into issues regarding the sizes of the views on disk. He was quick to reply, offering several suggestions. I'd like to thank Jan and Chris for their support and encouragement.

NoSQL databases are here to stay, and CouchDB is truly unique in this area. The way it handles views, and its support for replication/synchronization set it apart from the others. There are already several large projects, like Ubuntu One, that are relying on CouchDB to deliver what nobody else can. Because of this, I'm sure CouchDB has a very bright future ahead of it.

2010 Couchio — www.couch.io