Culvert – A secondary indexing framework for BigTable

*Culvert is a secondary indexing framework for BigTable. Currently, it integrates with HBase and Accumulo. It is designed not only for using the built in indexes by for easy extention to existing, hand-rolled secondary indexes. It is available at:* [*https://github.com/booz-allen-hamilton/culvert*](https://github.com/booz-allen-hamilton/culvert) *If you are just interested in how it works, skip down a few paragraphs.*

Big Data is big innovation, big innovation and, in the end, big money. The only problem is, is it can be a huge pain to get running….and then to get running correctly. Jump back 5 years – Google releases BigTable (labs.**google**.com/papers/**bigtable**-osdi06.pdf) and the open source world jumped at the idea, quickly spinning up HBase (org.apache.hbase) under the Hadoop umbrella. And for a while it seemed great! I can store petabytes of data – awesome. I can access it in real time – even better. And do appends, updates and deletes over a write-once file system? Fantastic. It was even so great the US Government put their heads together and came up with their own version of BigTable, Accumulo (<http://incubator.apache.org/accumulo/>), optimized for high throughput, though still faithful to many of the aspects of the original BigTable.

So great, we have this massively scalable database. Well, turns out that BigTable doesn’t cover everything we want to do with the database, particularly if you want to do fast lookups or scale out even farther or do traditional RDMS operations. So along comes Megastore (research.google.com/pubs/archive/36971.pdf). Now, there are a lot of things going on in Megastore that most of the companies outside of Google don’t need or are covered via alternative means (see Hive (hive.apache.org) or Pig (pig.apache.org)). However, one of the things that isn’t really covered by external tools is indexing.

Now you are probably, “Woah, hold on! What about Lily (<http://www.lilyproject.org/lily/index.html>)? Or Solr (<http://lucene.apache.org/solr/>)? Or etc….???” Well these things are good if you are doing indexing on just one thing – unstructured text in a given field. And a lot of times, that’s all you need. This is especially true as these tools integrate with search tools. However, what about the case where you need to index across multiple fields? Or build your own special indexes to make it go fast? What about trying out new index schemes? Then you are going to be out of luck and hand rolling your own.

To make sure these indexes scale you then have to store them in a cloud (probably the same database as the one hosting your data). Okay, doable but that can get a little tricky to make sure it scales well. Then you have to make sure that when you update your database that the indexes also get updated. And then you have to build a tool to use those indexes. What about using something SQL-like? Then you are writing your own SQL parser and then pipe that into your indexing and *then* use that to pull out and combine data. Now consider that you have to make that performant on a cloud scale. Ouch.

Clearly this is a hard problem. And every organization doesn’t want to solve this problem from the ground up, scalably every time. You just want to write indexes and have it integrate with all your tools right away. Easy. Simple. Done.

**Enter Culvert.**

Culvert is a secondary indexing *platform*, which means it provides everything you need to write indexes and use them to access and well, index your data.

Lets start with how you can actually get a Culvert client up and running. Turns out its pretty simple. We are going to use an example of connecting to an Accumulo Database:

<code>

//start configuring how to connect to the instance

Configuration conf = **new** Configuration();

conf.set(AccumuloConstants.*INSTANCE\_CLASS\_KEY*, ZooKeeperInstance.**class**.getName());

conf.set(AccumuloConstants.*INSTANCE\_NAME\_KEY*, INSTANCE\_NAME);

…//set all your other configuration values

// create the database adapter with a configuration

DatabaseAdapter database = **new** AccumuloDatabaseAdapter();

database.setConf(conf);

// create a client to configure

Client client = **new** Client(CConfiguration.*getDefault*());

//setup the client to talk to your database

Client.*setDatabase*(database, client.getConf());

</code>

That wasn’t too bad, right? At this point we’ve got a client to talk to the database. Since you are using Culvert is for indexing, the next thing you would want to do is add an index. Its actually pretty simple programmatically:

<code>

// create term-based index : index each of the words in the value, where the

// row key is the word and the row id is stored in the rest of the key

Index index = **new** TermBasedIndex(INDEX\_NAME, database, PRIMARY\_TABLE\_NAME,

INDEX\_TABLE\_NAME, COLUMN\_FAMILY\_TO\_INDEX, COLUMN\_QUALIFIER\_TO\_INDEX);

// other index definitions could also be loaded from the configuration

// and programmatically add the index to the client's configuration

client.addIndex(index);

</code>

Its important to note that each index needs to be given a unique name, otherwise namespace conflicts will occur. But generally this is not a problem and it useful when you want to have more than one index of the same type (eg. You want to do a TermBasedIndex on two different tables, two different fields, two different whatever).

You can also save yourself some effort by setuping your indexes in the configuration – the client will pick these up when it starts and automatically make sure the indexes you specified are used.

Once you have the client setup and all the indexes specified, the next step is to put data in the table. All data is wrapped as the high level Culvert type key and value - a CKeyValue. A CKeyValue is then transformed into the correct key and value for the underlying database. This makes doing an insert very similar to how inserts are done already in a BigTable system:

<code>

//doing an insert

List<CKeyValue> valuesToPut = Lists.*newArrayList*(**new** CKeyValue("foo"

.getBytes(), "bar".getBytes(), "baz".getBytes(), "value".getBytes()));

Put put = **new** Put(valuesToPut);

client.put(PRIMARY\_TABLE, put);

</code>

Pretty simple, right? Not only are these items being inserted into the database, Culvert also takes care of all the heavy lifting for you of make sure those values get indexed by *all* the indexes you have added to the client.

Secondary indexes are only useful if you can actually access the data. Culvert also handles doing this via “Constraints”. A constraint is the way you query the index, it’s the way you get the columns associated with row ids that the index stores and its also the way you can do efficient SQL-like queries.

*For those interested, we used the decorator design pattern here to make it really easy to that nesting. Every constraint takes another constraint and some parameters.*

Querying your data back out using the indexes is a little bit more complex as you have to build up your constraints but once you pick up the general strategy, it isn’t too bad. Lets start with just doing a simple query of the index looking for any records that have the word “value” in them:

<code>

Index c1Index = client.getIndexByName(*INDEX\_NAME*);

Constraint c1Constraint = **new** RetrieveColumns(**new** IndexRangeConstraint(

c1Index, **new** CRange("value".getBytes())), c1Index.getPrimaryTable());

// check the first constraint

SeekingCurrentIterator iter = c1Constraint.getResultIterator();

</code>

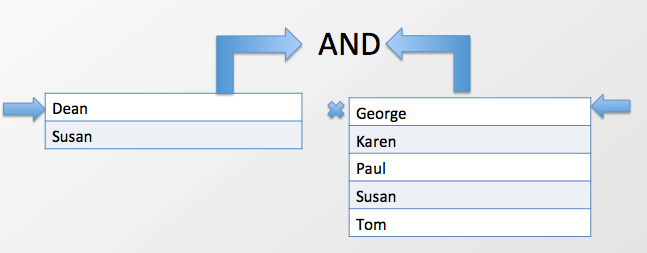
First, we get the index out of the client that you want to use when querying (to make sure you are searching for the right field). Then you build a constraint to use as a query.

That constraint is actually a nested constraint, describing each step in the process. First you scan the index to get the row ids of the field you are looking for (in this example, rows that the have word “value”) – this is the IndexRangeConstraint. You can basically think of this as a ‘WHERE’ clause where we explicitly specify the index to search for that value. This is because for things like the TermBasedIndex, you would be looking for values in different fields, depending on which index you use – you don’t want to look for email sender names in the content field, right? Most queries are going to start with an index range constraint.

Then once you have an all the row ids, you can go and actually get the rows specified using RetrieveColumns – retrieving all the columns associated with that row id. Its just like you would be doing with indexes all ready, just formalized and prebuilt for you. Makes sense, right?

That is the simple case - you just want to pull values of your table that you have indexed.

Now consider a little bit more complex case – doing an AND between the results of two queries. Now the simple, home rolled solution, is that you load all the left side of the AND into memory, then check to see which values from the right side match up. This is actually pretty bad if you pick the wrong side of the AND to load into memory – you will probably blow out memory and crash your client before getting any result. Culvert takes a different approach - each side of the AND is streamed to the client and only matching values are kept around, so you never have more than the *number of matches +2* elements in memory. It looks something like this:



Each side of the AND is streamed back to the AND logic on the client, where we can decide which rows to keep and which rows to discard. Note here that Culvert is leveraging the fact that the BigTable model enforces that from each TableServer or RegionServer will returned ordered results, so all we need to do is make sure we match the right results up. Here is the code to do an index based AND:

<code>

Constraint and = **new** And(c1Constraint, c2Constraint);

iter = and.getResultIterator();

</code>

In the beginning lets assume that you already have the Constraints for each side of the AND based on the index you want to search (just like we did before)

Culvert also supports a variety of other SQL-like constructs OR and JOIN. OR works very similarly to AND, just with slightly different logic. JOIN, on the other hand, can be either naïve – just joining two tables – or index based. However, in both cases if the underlying database supports it, the JOIN is actually implemented as a *server-side join*. This means it is incredibly efficient and powerful. Currently only the HBase adapter supports server-side joins, but Culvert developers are working on extending Accumulo to support this functionality (see ACCUMULO-14 https://issues.apache.org/jira/browse/ACCUMULO-80).

If you don’t want to use straight Java to interact with the index, Culvert also (soon!) works with Hive. It integrates directly with Hive as just another handle (similar to how the HBase-Hive handler works). When you send an HQL query to Hive, Culvert pulls out the predicates that it can handle and then queries the indexes you have specified via configuration to serve out only the results that Hive will actually use. This means you get *huge* speedups using Culvert with Hive.

And that is really all there is to using Culvert!

If you are still excited, please download the code at:

[www.github.com/booz-allen-hamilton/culvert](http://www.github.com/booz-allen-hamilton/culvert)

Also, if you are interested in learning about how it works under the hood, you can see the original talk at Hadoop Summit 2011 here (http://www.slideshare.net/jesse\_yates/culvert-a-robust-framework-for-secondary-indexing-of-structured-and-unstructured-data)