A huge amount of unstructured or semi-structured data is available on the Internet today. NoSQL databases able to dynamically load content in nearly any shape have made putting this data into an accessible form much easier. This tutorial explains how to build or replicate a data set in the eXtensible Business Reporting Language (XBRL) format, the type used by the U.S. Securities and Exchange Commission.

NoSQL databases are all the rage right now, and for good reason. There's a huge amount of unstructured or semi-structured data available on the Internet. While loading this data into a traditional RDBMS was cumbersome and difficult, the ability of NoSQL databases to dynamically load content in nearly any shape makes putting this unstructured or semi-structured data in an accessible form much easier.

In this tutorial, the type of dataset we're interested in is that used by the United States Securities and Exchange Commission. SEC reports, required by law from all US public companies, contain things such as balance sheets, cash flow statements, and other pertinent financial information. Originally mandated after the 1929 stock market crash to increase transparency and, thus, prevent another Great Depression, these reports contain data that is freely available. Here's a portion of IBM's statement of earnings as an example:

Initially, SEC reports were submitted on paper, then later in electronic format. Eventually, starting in 2005, the SEC started to phase in submission of this data in a machine-readable structured format called eXtensible Business Reporting Language (XBRL). XBRL is a type of XML document, and a full explanation of it is well beyond the scope of this tutorial. However, the "Services" number in the previous example looks like this in XBRL:
In this tutorial, you’ll learn how to build your own database, either from scratch or through replication, and deploy your own version of xbrl.mybluemix.net.

Each quarterly or annual report filed to the SEC includes an accompanying XBRL file that provides the same data as the human-readable report. The SEC provides a listing of these files and they are free for anyone to download. While this data is provided in a structured form, it’s not easy to query or enumerate because it consists of hundreds of thousands of distinct XML files.

The app I discuss in this tutorial, xbrl.mybluemix.net, provides a simple interface to query this dataset and display it in an easily understood manner.

As an example, see the following figure. After searching for a company and a financial concept (“AssetsCurrent” in the example), the data is presented in a line graph. Individual points have a hover fly-out that can be “pinned” to provide navigation to the filing on the SEC website where the data was acquired.

Run the app
Get the code

What you’ll need to build your application

- A Bluemix account and a DevOps Services account, both linked to your IBM ID
- Optionally, for local deployment: Node.js runtime

Step 1. Build a database

There are three options for providing data to your application. In decreasing order of difficulty, they are:

a. Build a new database from scratch.
b. Replicate the existing database to your own Cloudant account.
c. Use the existing database.

Option c really means doing nothing, so we'll focus on options a and b. If you want to use the existing database, skip ahead to Step 2. Create a new Project on DevOps Services.

Option 1: Build a new database from scratch

1. Log in to Bluemix and add the Cloudant NoSQL DB service to your Bluemix space.
2. Open the Cloudant dashboard. (See the appendix for instructions on how to access the Cloudant dashboard.) Create a new database.
3. Install the Node.js runtime locally if you haven't done so already.
4. From the XbrlBuilder source repository, export the source tree to an archive file.
5. Extract the archive into a local directory.
6. From the XbrlBuilder folder, start the application by entering

   `node app.js`

   You will need to specify the following environment variables:
   
   a. BASE_DIRECTORY: Where to store the downloaded XBRL files (gzipped).
   b. CLOUDANT_DATABASE: The Cloudant database you created in step 2. It should look like:

   `https://<user>:<pass>@cloudant.com/<db>/`

   These credentials can be found in your Bluemix dashboard. Note that if there is no Bluemix app bound to the Cloudant service, you must create one to see these two credentials.

   The application will iterate over every month that XBRL filings have been accepted. It first downloads the XBRL files to BASE_DIRECTORY, and then uploads them to CLOUDANT_DATABASE. If a failure that causes an unrecoverable error happens while downloading, simply restart the app. Any previously downloaded files will be skipped.

7. Create the necessary views. These are:

   - **factsMainViews/EntityConceptName**
     
     **Map:**
     
     ```javascript
     function getValue(fqn) {
       return fqn.substring(fqn.lastIndexOf('/') + 1, fqn.length);
     }
     
     function(doc) {
       emit([doc['http://www.xbrl.org/2003/instance/Entity'], getValue(doc['http://www.xbrl.org/2003/instance/Concept'])], null);
     }
     ```

     **Reduce:**

     `_count`

   - **factsMainSearchIndexes/EntitySplitConcept**
     
     Search index function:
function unCamelCase (str){
    return str
    // insert a space between lower & upper
    .replace(/([a-z])/g, &apos;$1 $&apos;)
    // space before last upper in a sequence followed by lower
    .replace(/([A-Z]+)(([A-Z])((a-z))/, &apos;$1 $2$3&quot;)
    // uppercase the first character
    .replace(/^./, function(str){ return str.toUpperCase(); });
}

function getValue(fqn)
{
    return fqn.substring(fqn.lastIndexOf(&quot;/&quot;) + 1, fqn.length);
}

function(doc){
    index(&quot;entity&quot;, doc['http://www.xbrl.org/2003/instance/Entity'],
        {&quot;store&quot;: true});
    index(&quot;conceptNameSplit&quot;, unCamelCase(getValue(doc['http://www.xbrl.org/2003/instance/Concept'])),
        {&quot;store&quot;: true, &quot;facet&quot;:true});
}

factsMainSearchIndexes/EntityCipherCompanyName

Search index function:

function rotateText(text, rotation) {
    // Surrogate pair limit
    var bound = 0x10000;

    // Force the rotation an integer and within bounds, just to be safe
    rotation = parseInt(rotation) % bound;

    // Might as well return the text if there's no change
    if(rotation === 0) return text;

    // Create string from character codes
    return String.fromCharCode.apply(null,
        // Turn string to character codes
        text.split('').map(function(v) {
            // Return current character code + rotation
            return (v.charCodeAt() + rotation + bound) % bound;
        }));
}

function getValue(fqn)
{
    return fqn.substring(fqn.lastIndexOf("/") + 1, fqn.length);
}

function(doc){
    index("companyName", rotateText(doc['http://www.xbrl.org/2003/instance/Entity'], 325) + ' ' +
        doc['http://www.sec.gov/Archives/edgar/companyName'], {"store": true, "facet":true});
}

Option 2: Replicate to a new Cloudant database

1. Log in to Bluemix and add the Cloudant NoSQL DB service to your Bluemix space.
2. Open the Cloudant dashboard. (See the appendix for instructions on how to access it.)
3. Go to the replication screen and do the following:
   - Select New Replication.
   - Under SOURCE DATABASE > Remote Database, enter the database URL:
     https://0741ae13-4f99-4ff8-8282-60d27e161c7f-bluemix.cloudant.com/facts
• Under **TARGET DATABASE > New Database > Create a new database locally**, enter a name for the new database. It can be any name you choose. In this example, I use "facts."

![New Replication](image)

4. Click **Replicate**.

**Note:** The facts database is over 100 gigabytes. A smaller subset of the data (approximately 47MB) is available and is called **facts_test**. Use this subset to test your replication. You could even use facts_test to test the whole application, though you would need to create the views shown in the final step of **Option 1: Build a new database from scratch**.

**Step 2. Create a new project on DevOps Services**

1. Log in to **DevOps Services**.
2. Browse to the **XbrlServer source repository**.
3. Click **FORK** on the top menu and follow the instructions to set up a new project.

**Step 3. Deploy the app to Bluemix**

1. Browse to the DevOps project you created in **Step 2**.
2. Modify the code as necessary to point to the correct database:
   a. If you created your own database linked to your app and called it "facts," there is nothing to do here. The Cloudant URL will come from the app environment variables.
b. If you created your own database named something other than "facts," you must modify the `cloudantFactsUri` parameter in the `helpers.coffee` file and the `cloudantUri` parameter in the `elements.coffee`, `companies.coffee`, and `size.coffee` files.

c. If you used the existing database, then modify the `getCloudantUrl` method in the `helpers.coffee` file to return

```
https://0741ae13-4f99-4ffb-8282-60d27e161c7fbluemix.cloudant.com
```

3. Click the root folder, click the drop-down button in the Launch Configuration run bar, and click **CREATE NEW**.

4. Follow the instructions to complete the deployment process.

The deployment process might take a few minutes. A message displays when the process is complete, or you can check the deployment status from the Bluemix dashboard.

You need to set up the deployment only one time. You can deploy again with the same setup by clicking the **PLAY** button beside the Launch Configuration run bar.

**Code Highlights**

**XbrlBuilder**

Here are some of the key files from the XbrlBuilder source repository:

- `app.coffee` is the main entry point.
  - `parseRss()` takes the RSS feeds of submitted files, parses them for relevant XBRL files, and then starts the download/parsing process of those files.
  - `parseInstance()` takes an XBRL instance and transforms it into a stream of JSON documents to be sent to the Cloudant bulk_posts endpoint.
- `streams/CloudantBulkPostStream.coffee` takes a series of objects and serializes them as a JSON array for upload to Cloudant.
- `streams/FactTransformStream.coffee` takes a series of facts from an XBRL document and transforms them to JavaScript objects.
- `streams/InstancePerProcessorTransformStream.coffee` saves information about an instance document from the RSS feed in which it was found to be used in the `FactTransformStream.coffee` file.
- `streams/XmlTransformStream.coffee` is a streaming XML parser.
- `model/xml/Selector.coffee` is a class used to select an element in XML, used by `XmlTransformStream.coffee` (as opposed to a string-based selector such as XLink).
- `model/xml/SelectorSet.coffee` is a series of tests to be performed on XML elements as the parser encounters them to determine if they are to be selected. These are logical or operations. Only one test needs to be successful for an element to be selected.

**XbrlServer**

Here are some of the key files for `xbrl.mybluemix.net`:

- `routes/helpers.coffee` contains a number of helper functions used by the application:
  - `getCloudantURL()` returns the URL of the Cloudant data store; by default, it pulls from `VCAP_SERVICES`. Change this to develop locally or to pull from a database not defined there.
- recursiveCloudantSearch() is not currently used; see Every Result from a Cloudant Search Index for information on how it could be used.
- get_parsed_fact_data() returns the data to populate the chart.
- tickerResolver() allows users to search for ticker symbols as well as text in the company name.
- routes/companies.coffee is used to populate the company search combo box.
- routes/elements.coffee is used to populate the concept search combo box.
- routes/facts.coffee returns the data to populate the chart.
- routes/save.coffee saves data in Cloudant to allow the share feature to work. It requires write access, so using the existing facts database, being read-only, is not an option here.
- model/Fact.coffee allows the JSON objects that come out of Cloudant to be used more effectively.
  - getValue() returns the text following the last "/" in a URI.
  - GetUnitDescription() returns a friendly string description of the fact's unit.
  - GetDimensions() returns an associative array of the axis/member pairs controlling the XBRL dimensionality of the fact.
  - GetHashValue() returns a string that should uniquely identify a fact irrespective of its value. Where two facts have the same HashValue, a decision must be made about which is more "correct," for example, the most recent.
  - GetPeriodDescription() returns a friendly string description of the fact's XBRL dimensionality.
- streams/FactTransformStream.coffee takes fact data from Cloudant and transforms it to a form more suitable for loading into Highcharts (the charting tool used on xbrl.mybluemix.net).
- public/scripts/index.coffee contains all the client-side JavaScript code used by xbrl.mybluemix.net

Lessons learned while working with Cloudant

This section consists of a series of lessons I learned while building xbrl.mybluemix.net. Some of them may sound very simple to Node.js or Cloudant gurus, but as I started with zero knowledge of either of these topics, I had to learn the hard way.

Minimize the number of views in each design document

This is not an official "best practice," it's just a habit I fell into while working with Cloudant and large datasets. There are no "queries" as such when working with a Cloudant database. Instead, you can create secondary or search indexes or "views" against the database. Both are functions written in JavaScript that are executed against every document in a database with the results stored in a B-tree. This makes getting results from large data sets blazingly fast, as all possible results are pre-calculated. However, creating the index can be very slow if there is lots of data or the index function is complex. Each view is part of a design document, and each design document can contain any number of views. As you can see from the Cloudant documentation:

View indexes are rebuilt when a design document is updated. An update to any one of the views causes all the views in the document to be rebuilt.

That second sentence is the important bit. When this tutorial was written, xbrl.mybluemix.net had just under 80 million documents in its main database. Building a simple secondary index against
this data usually takes about 72 hours. If you have an application depending on such a view, it will either show very incomplete data, or no data at all, during this time. There are other tutorials out there that say that each design document conceptually represents an "application" and each view a part of that application. That sounds nice, but if you've got hundreds of gigabytes or terabytes of data, it means that if you want to change one part of your application, or even just add to it, you're going to bring the whole thing down for days. Even if it's just dev and not prod, that's really annoying. So whether it's an official "best practice," I've fallen into the habit of having as few views per design document as possible, usually one or two. It may not be as pretty, but if I want to change one part of an application without taking down all the other parts, it's really nice.

**CamelCase tokenizer for Cloudant search indexes**

When creating a search index over a Cloudant database, you'll find a number of analyzers available for parsing and tokenizing full-text data. When building xbrl.mybluemix.net, I wanted to search on CamelCase strings such as *SalesRevenueNet*. There is no built-in CamelCase tokenizer in Cloudant. It's possible to build your own Lucene.Net tokenizer for CamelCase strings, but this is of no use when using Cloudant. Luckily, the flexibility of creating your search index as a JavaScript function lets you tokenize pretty much any way you want. Simply apply your own tokenizer when creating the index.

```javascript
function unCamelCase (str){
    return str
        .replace(/([a-z])([A-Z])/g, '$1 $2')
        .replace(/(\b[A-Z]+)([A-Z])([a-z])/, '$1 $2$3')
        .replace(/^[A-Z]+/, function(str){ return str.toUpperCase(); });
}

function(doc){
    index("name", unCamelCase(doc.name), {"store": true});
}
```

It's that easy. You don't have to index on a field that already exists; you can create one at "run time." For those with keen eyes, I realize the previous example doesn't take into account numbers, but if that's a concern, you can do that as an exercise. Credit goes to this [StackOverflow thread](https://stackoverflow.com) for providing a JavaScript example of a CamelCase tokenizer.

**"Distinct" results from a Cloudant search index**

Search indexes are a really nice feature of Cloudant. I'd go so far as to say they're essential. Without them, what many would consider routine database tasks would be much more complicated, if not impossible. Let's look at a simplified set of the data available at [xbrl.mybluemix.net](https://xbrl.mybluemix.net), where IBM is reporting some financial information over time:

```json
{ "date": "03/31/2010", "company": "IBM", "concept":"Assets", "value": 100.00}
{ "date": "06/30/2010", "company": "IBM", "concept":"Assets", "value": 110.00}
...repeats...
```

Build an enterprise-scale database of SEC financial data with Bluemix and Cloudant
Nothing too complicated so far. Let's create a search index on the concept field:

```javascript
function(doc) {
    index("default",
          doc.company,
          {"store": true});
    index("concept",
          doc.concept,
          {"store": true});
}
```

So far, everything looks good. To populate the second auto-complete combo box on the main page of xbrl.mybluemix.net, we need to search for all the concepts that a company has filed that contain a certain string. Let's suppose we're looking for all concepts that IBM filed containing the text "Assets," limited to 10 results. The query URI would look something like this:

```
https://myaccount.cloudant.com/facts/_design/factsSearches/_search/nameSearch?q="IBM"%20AND%20name:Assets&limit=10
```

The actual JSON for the results isn't important yet, what's important is the response from this URI would contain the following results:

```
{ "date": "03/31/2010", "company": "IBM",
  "concept": "Assets", "value": 100.00}
{ "date": "06/30/2010", "company": "IBM",
  "concept": "Assets", "value": 110.00}
```

... repeats ...

```
{ "date": "06/32/2012", "company": "IBM",
  "concept": "Assets", "value": 150.00}
```

"Current Assets" is not there! Nor should it be. The search index did exactly what it was asked to do. What we actually wanted was to find distinct concepts with "Assets" in them. In SQL terms, we were looking for:

```
SELECT DISTINCT concept
FROM facts
WHERE company='IBM'
    AND concept LIKE '%Assets%'
```

A naive solution would be to increase the limit, say to 100, then simply process the list on the client or server, keeping only distinct results, before populating the combo box. This would work for our
simple example, but suppose IBM reported its Assets 100 times, or 1000 times? Cloudant imposes a maximum limit of 200 results. If IBM had reported "Assets" 1000 times (note that this is a bit of a contrived example), we'd be out of luck. Because we need only a list of distinct values of a single field (concept), we can use the counts facet. Simply rebuild the index like so:

```javascript
function(doc){
    index("company", 
        doc.company,
        {"store": true});
    index("concept", 
        doc.concept,
        {"store": true, "facet": true});
}
```

Now the query URI looks like this:

```plaintext
https://myaccount.cloudant.com/facts/_design/factsSearches/_search/nameSearch?q="IBM"%20AND%20name:Assets&limit=0&counts=["name"]
```

We use `limit=0` because we aren't really concerned with the actual documents, just the list of distinct values; `counts=["name"]` gives us this. The JSON resulting from this query would look something like:

```json
{"total_rows":17,"bookmark":"g2o","rows":[],"counts":{"name":{"Assets":16,"CurrentAssets":1}}}
```

That final associative array is what we're after, a list of permutations of the name field containing "Assets" and how many times each permutation appears. **Caveat:** There is no "`limit=x`" option when working with the counts facet. So if the query were to be something like all documents where the concept contains "e," the query could take a long time and return an extremely long list of counts.

### Return multiple fields from a "distinct" Cloudant search index

In the previous section, we wanted to get distinct results from a search index. It worked, but it had a drawback in that we could not return more than one field. By that, I mean we could not create SQL like the following:

```sql
SELECT DISTINCT 
    identifier,
    name
FROM 
    companies
WHERE 
    name LIKE 'International%'
```

We get back only the field we searched on. We could simply concatenate two or more fields when creating the index, but there's a serious drawback with this method. We would be indexing all the concatenated fields, rather than indexing one and getting indexing many back.

This isn't a problem if the fields are using strictly different characters spaces. For example, one only-alphabetic-characters and one only-numeric-characters. Even if this isn't the case, we can force it to be by using a substitution cipher. JSON documents are not limited to ASCII characters.
The UTF-8 character space is quite large, and we can simply shift one or more of the fields to a totally different character space when creating the index.

Here’s how the search index would look (credit goes to Tim Severien’s cipher example on GitHub for the rotate function):

```javascript
function rotateText(text, rotation) {
  // Surrogate pair limit
  var bound = 0x10000;

  // Force the rotation an integer and within bounds, just to be safe
  rotation = parseInt(rotation) % bound;

  // Might as well return the text if there's no change
  if(rotation === 0) return text;

  // Create string from character codes
  return String.fromCharCode.apply(null,
    // Turn string to character codes
    text.split('').map(function(v) {
      // Return current character code + rotation
      return (v.charCodeAt() + rotation + bound) % bound;
    })
  );
}

function(doc){
  index("companyName", rotateText(doc.identifier, 500) + ' ' + doc.name, {"store": true, "facet":true});
}
```

We’re creating the index on the concatenated string consisting of the cipher-ed identifier and the original name. This means the pair '0000051143', 'INTERNATIONAL BUSINESS MACHINES CORP' will become the string 'ȤȤȤȤȤȤȤȤȤ INTERNATIONAL BUSINESS MACHINES CORP'.

From here we continue as before, using faceting to return a list of distinct results. It’s then simply a matter of reversing the cipher to get the identifier back (in this example, rotate the text by -500). You can either make the very reasonable assumption that no one will enter characters from the cipher-ed character space into your search field or not allow searches that contain them.

**Every result from a Cloudant search index**

As we discovered earlier, there is a limit of 200 results from any single call to a search index. Suppose, for whatever reason, we want every result. Let’s look at the actual JSON we get back when performing a call against a search index URI:
Notice the first field in the result, "total_rows". Cloudant knows how many results there are. However, it will give them to you only in batches of up to 200. Notice the second field in the result "bookmark". To get the next $n$ results, simply call the same URI with a "bookmark=" parameter, using the bookmark provided from the previous $n$ results. Using the miracles of Node.js, streaming, and recursion, we'll look at a simple method to first stream every result, then get a list of $n$ distinct results. First, we'll need a stream that can extract the bookmark value and store it in a pass-by-reference variable.

```javascript
stream = require('stream')

class exports.JSONArrayTransformStream extends stream.Transform
  constructor: () ->
    @first = true
    super
    objectMode: true

  _transform: (chunk, enc, next) ->
    if (@first)
      @push('[
        @first = false
      ]
    else
      @push(',
    
    @push(JSON.stringify(chunk))
    
    next()

  _flush: (next) ->
    if (@first)
      @push('[
        @first = false
      ]
    else
      @push(']
    next()
```

Next we'll need a stream that takes incoming objects and outputs them as a JSON array. This presumes we're sending the results somewhere, such as a response stream, a file, or the console.

```javascript
stream = require('stream')
```
Finally, we'll use a recursive function to execute each API call in sequence, using the bookmark from the previous call. The JSONStream package is very useful here.

```javascript
stream = require('stream')

class exports.JSONArrayTransformStream extends stream.Transform

    constructor: () ->
        @first = true
        super
        objectMode: true

    _transform: (chunk, enc, next) ->
        if (@first)
            @push('[')
            @first = false
        else
            @push(',
')
        @push(JSON.stringify(chunk))
        next()

    _flush: (next) ->
        if (@first)
            @push('[]]
        else
            @push(']
        next()

If we want distinct results, we're going to need a new stream that accumulates results before outputting at the very end.

```javascript
stream = require('stream')

class exports.DistinctingTransformStream extends stream.Transform

    constructor: () ->
        @first = true
        super
        objectMode: true

    _transform: (chunk, enc, next) ->
        if (@first)
            @push('[')
            @first = false
        else
            @push(',
')
        @push(JSON.stringify(chunk))
        next()

    _flush: (next) ->
        if (@first)
            @push('[]
        else
            @push(']
        next())
Conclusion

In this tutorial, you've learned how to build your own enterprise-scale Cloudant database of financial data by using Node.js. You've also seen how it's possible to build a simple UI over that data in the form of a web charting tool. In addition, we've explored some lessons about dealing with large datasets and finding distinct results in such datasets.

Appendix: Open the Cloudant dashboard

1. Click on your application from the Bluemix dashboard. You should see a screen similar to this:
2. Click on the Cloudant icon.
3. From the next screen, click the **Launch** button.

The Cloudant NoSQL service provides access to a fully managed NoSQL JSON data layer that's always on.

© Copyright IBM Corporation 2015

**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)